NEWS 3X/400's

Desktop Guide to the S/36

By Mel Beckman,
Gary Kratzer,
and Roger Pence

A Division of
DUKE COMMUNICATIONS
INTERNATIONAL

Loveland, Colorado

Library of Congress Cataloging-in-Publication Data

Beckman, Mel, 1955-
NEWS 3X/400's Desktop Guide to the S/36 / by Mel Beckman, Gary Kratzer, and Roger Pence. — 1st ed.
p. cm.
Includes index.
ISBN 0-9628743-5-3 : $99.00
1. IBM System/36 (Computer) I. Kratzer, Gary, 1961- II. Pence, Roger, 1953- . III. Title. IV. Title: Desktop Guide to the S/36.
QA76.8.I107B43 1992
004. 1'45—dc20 92-34281
CIP

Copyright © 1992 by DUKE PRESS
DUKE COMMUNICATIONS INTERNATIONAL
Loveland, Colorado

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

Duke Communications International reserves a compilation copyright on the software contents of the S/36 Desktop Guide Diskettes. A compilation copyright is granted when an organization collects the information in a lawful way, adds value to it, and offers it to others. Separate copyrights are also held by the respective authors whose software is included in this collection. This compilation copyright does not supersede individual ownership rights to any of the software by its original authors. You have the right to non-commercial private use of the software, but you do not have the right to resell, publish, or in any manner commercially exploit the software accompanying this book, or to participate in such reproduction, sale, publication or exploitation by any other person.

It is the reader's responsibility to ensure procedures and techniques used from this book are accurate and appropriate for the user's installation. No warranty is implied or expressed.

This book was printed and bound in the United States of America.
First Edition: December 1992

ISBN 0-9628743-5-3

To Patricia, Michelle, and Susie

// Acknowledgments

A whole slew of people helped bring this book to press. First and foremost are our editors, Dave Bernard and Sharon Hamm, who read (and corrected) every word of the manuscript. With three authors, their job was particularly complex, requiring the untangling of contradictory opinions and the coordination of many, many little bits of text. Dave and Sharon are responsible for melding the literary voices of three rarely coherent nerds into that of one slightly daffy, articulate author. We thank them for sticking with us to the very end. We also thank Chuck Lundgren for his sage advice on content and style, and Mike Patton, who gave us encouragement and the assurance that at least one person would buy the finished product. We greatly appreciate the ingenuity of our favorite Hollander, Edwin Helmus, for authoring the SNAP utility (included on the diskette).

Also instrumental to our success were Trish Frease, who provided early assistance with writing a book over E-mail, and Lynn Riggs and Jan Caufman who, with their trusty Macintoshes, formatted and typeset our text with style and artistry. Thanks also to Robin Strelow for her user-friendly interior and cover design of the book. For technical help we are indebted to Clark Buch of Complete Hardware Services and Jay Brock of A-1 Computer Marketing; they took time from their busy schedules in the present to research obscure S/36 hardware facts from the past. In addition, Ron Elliot and Wayne Madden provided invaluable tips on AS/400 compatibility.

Three people beta tested the diskette and accompanying text: Chuck Lundgren, Mike Derossier, and Joe Medeiros. They were our final quality checks on the part of this book to which you will entrust your valuable machine resources; their expertise, we feel, adds significantly to the value of the whole product.

Finally, in any book of this scope and complexity mistakes are bound to creep in. We only ask that you do not blame these on our esteemed helpers. Any errors are solely due to us, the authors.

Mel Beckman
Ventura, California

Gary Kratzer
Mission Viejo, California

Roger Pence
Marion, Indiana

Table of Contents

Acknowledgments IV
List of Figures XII
Introduction XVII

Section I — System Internals 1

Chapter 1: S/36 Architecture 3
Genealogy 3
Main Storage Processor 6
Control Storage Processor 7
Optional Processors 8
  The Workstation Controller 8
  Data Storage Controller 9
  MLCA and ELCA Processors 10
  Local Area Network Processor 11
IBM's Multiple Processor Advantage 11
The Channel 14
Disk Drives 15
The Bottom Line 16
Where to Learn More 17

Chapter 2: S/36 Memory Management 19
Main Memory Organization 19
Memory Concepts 21
Fragmentation 22
Solving Fragmentation 22
Virtual Memory 25
Page-in, Page-out Mechanisms 26
Peculiarities of S/36 VM 28
System Queue Space 29

Chapter 3: Inside Disk Data Management 33
Disk Mechanics 33
Data Files 39
  Organization and Operation 39
  Primary and Overflow Index Areas 41
The Storage Index 43
  Ripple-Down Adds 46
  Adding Gaps 48
  Alternate Indices and DUPKEY Processing 51
  Keysorting 52

Section II — Hardware Adviser 57

Chapter 4: S/36 Models and Configurations 59
Overview of Models 60

Memory 62
Disk 64
Workstations 66
Communications 70
Local Area Networks 71
Optical Storage 73
Conclusion 75

Chapter 5: The Importance of Memory and Disk Space 77
Just Add Memory 77
Memory Configuration 79
Pricing Memory 81
The Importance of Free Disk Space 83
System Work Areas 83
A Smarter Compress 83
Successful File Extends 84
Proper File Placement 86
Additional Spindles 86
Pricing Disk Drives 87
Is the S/36 Worth Upgrading? 88
Avoid Third-Party Pitfalls 89

Chapter 6: Other Configuration Considerations 91
Model D Upgrade 91
Upgrading to a 5360 92
A Dedicated Development Machine 94
Communications Upgrade 95
Nonswitched Network 97
Switched Network 97
Packet-Switched Network 97
Distributed Data Management 98
Maintenance 102

Section III — External Program Calls 105

Chapter 7: How External Program Calls Work 107
IBM's Little Secret 107
128 MB of Memory, Virtually! 108
Primitive Modules 109
EPCs in Action 110
The Program Invocation Stack 113
Coding EPCs: A Primer 114
EPCs and Disk Files 117
A Practical Example 119
Reaping the Benefits of EPCs 119

Chapter 8: A Comparison of EPC Vendor Offerings 123
The Contest 123
Design and Coding Considerations 124

Run the Good Race ... 128
Testing and Production Considerations ... 130
The Gravy ... 132
Wrap Up ... 132
Product Information ... 133

Chapter 9: Implementing Modular RPG Applications ... 135
The Mysterious Module ... 135
Activation and Invocation ... 136
External Subprogram Deactivation ... 136
Local Names ... 136
Parameter Passing ... 137
Late Binding ... 137
Breaking it Down ... 137
Defining the Problem ... 138
Implementing the Modules ... 139
Change is No Problem ... 142

Section IV — Living With Disk Data Management ... 145

Chapter 10: Using DBLOCK and IBLOCK Effectively ... 147
Blocking Data Records ... 147
Enabling Record Blocking ... 151
Sizing Data Buffers ... 152
Record Blocking Considerations ... 154
Index Blocking ... 155
Sizing Index Buffers ... 156
Index Blocking Considerations ... 159
Where Buffers Live ... 161
Allocating Appropriate Buffer Sizes ... 162
Benchmarks ... 165
Mental Blocking ... 165

Chapter 11: Prescriptions for Healthy DDM ... 167
Is There a Doctor in the House? ... 167
Taking Action ... 170
Keeping Storage Indexes Open ... 171
The Proof is in the Performance ... 177
Avoiding the Deadly Embrace ... 177
Who is the Culprit? ... 181
A Ray of Light ... 187

Chapter 12: A Baker's Dozen DDM Tips and Techniques ... 189
1. Consider alternate indexes as an alternative to #GSORT ... 189
2. Consider replacing indexed files with sequential files and alternate indexes ... 190
3. Keep alternate indexes to a minimum ... 191
4. Spindle placement is more important than file placement ... 191
5. Share if you must ... 193

6. Do some of DDM's work yourself ... 194
7. Take the bypass ... 194
8. Avoid being underextended ... 195
9. Change default file extend value easily ... 196
10. Don't pack 'em in ... 197
11. Reorganize files often — the easy way ... 198
12. Provide lots of disk space for KEYSORT ... 199
13. Put memory to work for system programs ... 201
Recapping the Baker's Dozen ... 202

Section V — Performance Measurement and Tuning ... 205

Chapter 13: Using SMF ... 207
The Good Catch ... 207
The Quick Look ... 209
MSP/CSP Usage Values ... 210
Disk Usage ... 211
Disk Seeks > 1/3 ... 211
User Area Disk Accesses (UADA) ... 212
Translated Calls/Loads Ratio ... 213
Communications Line Usage ... 213
The Art of Tuning ... 215
SMF Cookbook ... 215
Problem 1: High Disk Usage ... 215
Problem 2: Unbalanced Disk Usage ... 216
Problem 3: High CSP Usage ... 217
Problem 4: High MSP Usage ... 218
Problem 5: Sudden, Unexplained Response Time Degradation ... 220
Problem 6: File and Index Blocking Don't Seem to be Helping Performance ... 225
What You Don't Know *Can* Hurt You ... 225
References ... 226
SMF Summary Report Part 1: Summary Usage ... 226
SMF Summary Report Part 2: Summary System Event Counters ... 227

Chapter 14: Do You Need More Memory? ... 231
Where Does it All Go? ... 231
The Warning Signs ... 233
Memory Meter ... 234

Chapter 15: Cashing in On Extra Memory ... 237
How Cache Works ... 237
Starting Out with Cache ... 240
Counting Cache ... 241
Quick Cache ... 243
Dynamically Controlling Cache ... 245
Value-Added Cache ... 247
To Cache or Not to Cache ... 247

Chapter 16: Is Response Time Fast Enough? 249
Response-Time Speedometer 249
Looking for Delay 251
Getting SSP Out of the Loop 257

Section VI — Advanced Topics 261

Chapter 17: Harnessing the Power of Assembler Routines 263
String Handling Functions 264
SUBR$F 265
SUBR$C 266
SUBR$X 267
SUBRAT 268
SUBRBX 269
SUBRCS 269
SUBRUP 270
Library Manipulation Functions 271
SUBRLD 271
SUBRLR 274
SUBRSG 276
File Manipulation Functions 279
SUBRCO 280
SUBRFA 281
SUBRRN 286
System Management Functions 287
SUBRSY 288
SUBRUF 288
SUBRUL 289
SUBRUR 290
SUBRCT 291
SUBRDT 292
Device Control Functions 292
SUBRCP 293
SUBRDU 293
SUBREK 294
SUBRLN 295
SUBRMG 296
SUBRPC 296
SUBRPS 297
SUBRSX 298
SUBRWT 298
RBRIDG 299
Defining RLABLs 299
Making Your Call 299

Chapter 18: Profiling and Advanced Debugging 303
Profile of a Profiler 303
RPG X-Ray Machine 308

Going for the Gold 313
Reasonable Goals 318

Chapter 19: Achieving Upward Compatibility 321
First, a Word of Caution 321
The Two Products 323
The Implementation of Externally Described Files 324
RPG/400 File Operation Codes 328
Other RPG/400 Features 330
Additional Tips for Upward Compatibility 332
Data File Usage on the S/36 332
Code to Avoid Decimal Data Errors 332
Assembler Subroutines 333
The Final Analysis 335

Section VII — Into the Future 337

Chapter 20: The AS/400 339
Three Audiences 339
S/36E Layer 340
Smoke and Mirrors 341
Building and Controlling the S/36E 343
The Programming Environment 344
Free Usability and Performance Improvements 344
Compatibility with the S/36 345
Sizing up Your AS/400 345
Model Selection 346
Configuring Memory 347
Configuring Disk 348
Making Sure 350
All in All 351

Chapter 21: The Unix Alternatives 353
UNIBOL on the RS/6000: A Virtual S/36 353
  Functions and Features 353
  Missing Links 354
  S/36 Look and Feel 354
  Migration Patterns 356
  Passing Muster 356
  User Recommended 358
  Programming with UNIBOL 358
  More than Token Support 361
  Up and Coming 362
Open RS/36: A Load'N'Go Alternative 362
  Look and Feel 364
  See How They Run 365
  Testing, Testing 366
  Load Member Compatible 367

What Doesn't it Do? 367
AIX: An Intense Experience? 368
The Magic Answer? 369
Product Information 370
Machine Mimicry 371

Afterword 375

Index 377

List of Figures

Figure 1.1	System/36 Genealogy	4
Figure 1.2	System/36 Multiple Processors	6
Figure 1.3	Comparison of Features Among S/36 Models	9
Figure 1.4	Overview of S/36 Internal Components	12
Figure 1.5a	Timeline for Single-Processor Task Switching	14
Figure 1.5b	Timeline for S/36 Dual-Processor Task Switching	14
Figure 1.6	Run Length Limited Codes for Disk Error Correction	16
Figure 2.1	Main Storage Contents	20
Figure 2.2	System/36 Real Storage Organization	21
Figure 2.3	How Memory Fragmentation Occurs	23
Figure 2.4	How the System/36 Solves Fragmentation	23
Figure 2.5	Example of Generating a Real Address From a Translated Address	25
Figure 2.6	Segmented and Demand Paging Comparison	27
Figure 2.7	System Queue Space Requirements for Common Control Blocks	31
Figure 3.1	Hard Disk Anatomy	34
Figure 3.2	Hard Disk Physical Units	35
Figure 3.3	Sectors and Tracks	35
Figure 3.4	Heads, Sectors, and Cylinders	36
Figure 3.5	Comparison of Operating Parameters for Some S/36 Disk Drives	38
Figure 3.6	Index and Data Record Area of a Disk File	39
Figure 3.7	Records Fitting Evenly Within a Sector	40
Figure 3.8	Records Spanning a Sector	40
Figure 3.9	Logical Representation of a File Index	41
Figure 3.10	Logical View of One Sector Full of Keys	42
Figure 3.11	Primary and Overflow Index Areas	43
Figure 3.12	Sequential Phone Book Look-Up	44
Figure 3.13	Phone Book Look-Up the Storage Index Way	45
Figure 3.14	Keys Added to Index Area in Key Sequence	46
Figure 3.15	Index Area with Overflow Added	47
Figure 3.16	Index Area with Overflow Containing Many Entries	47
Figure 3.17	Index Area Including Overflow with Gaps	48
Figure 3.18	Index Area Including Overflow with Gaps — Big Bird Added	49
Figure 3.19	Index and Overflow Areas Full	50
Figure 3.20	Overflow Area Requiring Total Ripple Down of Entries	51
Figure 3.21	Fragment of Overflow Area That Support Duplicate Keys	53
Figure 3.22	Adding New Record to Overflow Area with Duplicate Keys	53
Figure 3.23	Formula to Calculate Disk Space Required for Workfile KEYSORT	55

Figure 4.1	Models and Features Overview	60
Figure 4.2	S/36 CPU Relative Computational Performance	62
Figure 4.3	Memory Configurations	63
Figure 4.4	Disk Configurations	65
Figure 4.5	Disk Drive Characteristics	65
Figure 4.6	Workstation Configurations	67
Figure 4.7	Twinax Daisy-Chain Local Workstation Network	67
Figure 4.8	Twisted-Pair Star Network Topology	69
Figure 4.9	RWS Support Using Communications Lines and Remote Workstation Controllers	69
Figure 4.10	Communications Configurations	71
Figure 4.11	S/36 LAN Interfaces	72
Figure 4.12	Token-Ring LAN Configurations	74
Figure 4.13	Optical Storage Configurations	76
Figure 5.1a	User Area of Memory Available without Cache	78
Figure 5.1b	User Area of Memory Available with Cache	78
Figure 5.2	More Memory Means More System Programs Resident	80
Figure 5.3	CNFIGSSP Screen 17.0	81
Figure 5.4	Maximum Memory Per S/36 Model	81
Figure 5.5	Disk Spindle After COMPRESS FREEHIGH and After COMPRESS FREELOW	84
Figure 5.6	Disk Spindle After "Smart" COMPRESS	84
Figure 5.7	"Smart" COMPRESS for Spindle A1	85
Figure 5.8	"Smart" COMPRESS for Spindles A1 and A2	85
Figure 5.9	Extended a File Once Successfully — But Not Again	86
Figure 5.10	Maximum Disk Capacity for S/36 Models	87
Figure 6.1	Recommended Communications Adapters	96
Figure 6.2	Overview of DDM Operations	100
Figure 7.1a	Job with Three Active Programs	111
Figure 7.1b	EPC Program Invocation	111
Figure 7.1c	Program Invocation Stack	113
Figure 7.2	Program Fragments with EPC Opcodes	115
Figure 7.3a	OCL for Program File Sharing	118
Figure 7.3b	RPG File Sharing Between Programs	118
Figure 7.4	Modular Application Design	120
Figure 8.1	OCL for a Single File Used by Two Subprograms	125
Figure 8.2	RPG II½ and 400RPG Coding for Subprograms PROGA and PROGB Referencing a Common File	126
Figure 8.3	IBM RPG Coding for Subprograms PROGA and PROGB Referencing a Common File	127
Figure 8.4	Performance Test of 10,000 CALLS Passing 128 Bytes of Parameters	129
Figure 8.5	Using a Library List to Exercise Test Modules in a Production Environment	131

Figure 9.1 Functional Decomposition of Appointment Scheduling Application 139
Figure 9.2 Modules Required to Add "Reschedule an Appointment" Function 142

Figure 10.1 Application Program with Unblocked Data Buffer 148
Figure 10.2 Application Program with 100-Record Block Data Buffer 148
Figure 10.3 A Minimum Disk Data Buffer for a File with 64-Byte Records 149
Figure 10.4 A Minimum Disk Data Buffer for a File with 56-Byte Records 150
Figure 10.5 The Data Buffer Shown in Figure 10.4 After Record 37 is Read 151
Figure 10.6 Enabling Record Blocking in RPG F-Spec 152
Figure 10.7 Specifying Record Blocking with the DBLOCK Keyword 152
Figure 10.8 DBLOCK Factors for Desired Data Buffer Sizes 153
Figure 10.9 One Sector Full of Keys 157
Figure 10.10 OCL Showing IBLOCK Keyword 157
Figure 10.11 IBLOCK Factors for Desired Index Buffer Size 158
Figure 10.12 18 K Program with Appended Buffers 162
Figure 10.13 26 K Program with Unappended Buffers 163
Figure 10.14 Status Users Screen 164

Figure 11.1 Index Doctor Report 169
Figure 11.2a One Program Using One File 173
Figure 11.2b Two Programs Using One File 173
Figure 11.3a Procedure KEEPOPEN 174
Figure 11.3b Procedure KOPENF 174
Figure 11.3c MRT Procedure KPOPEN2 174
Figure 11.3d MRT-NEP Program KPOPEN 175
Figure 11.4 KEEPOPEN Performance Benchmarks 178
Figure 11.5 Diagram of a Deadly Embrace 179
Figure 11.6 An Algorithm Using the NITU Strategy 181
Figure 11.7 RPG Code Incorporating the NITU Strategy 182
Figure 11.8 SHOWUR Screen 186

Figure 12.1 Changed Duplicate Key Value Causing Ripple-Down Add 192
Figure 12.2 Sequence of Events for an Extended File 196
Figure 12.3a REORGX Prompt Screen #1 200
Figure 12.3b REORGX Prompt Screen #2 200
Figure 12.4 REORGX's Syntax When Calling from a Procedure 201

Figure 13.1 Key SMF Measurements 210
Figure 13.2 File Placement 217
Figure 13.3 Sample SNAP Utility Display 219
Figure 13.4 I/O Counters Summary 221
Figure 13.5a File Access Counters by File from SMFPRINT ALL 223
Figure 13.5b File Access Counters by Task from SMFPRINT ALL 223

Figure 14.1 Main Storage Contents 232
Figure 14.2 Memory Meter Screen 235

Figure 15.1 Disk Accesses with and without Cache ... 238
Figure 15.2 How Disk Cache Takes Advantage of Locality ... 239
Figure 15.3 SMF Summary System Event Counters for Cache Evaluation ... 242
Figure 15.4 CACHIQ Display ... 244
Figure 15.5 CACHIQ Logfile Record Format ... 245
Figure 15.6 Sample Procedure to Automatically Change Cache Configuration on a Predetermined Schedule ... 246
Figure 15.7 Sample Procedure to Automatically Change Cache in Response to Changing Environment ... 246

Figure 16.1 Sample Detail Report from Utility RTIMER ... 251
Figure 16.2 Sample Summary Report from Utility RTIMER ... 252
Figure 16.3 Typical All-in-One Order Entry Panel ... 254
Figure 16.4a Customer Name and Address Information Screen ... 254
Figure 16.4b Customer Credit History Screen ... 255
Figure 16.4c Ship-To Address Screen ... 255
Figure 16.4d Customer Line-Item Order Screen ... 256
Figure 16.4e Summary Order Information Screen ... 256
Figure 16.5 Traditional Program Linkage Using OCL and LDA ... 258
Figure 16.6 Disk Overhead for Terminating/Initiating Task Sequence ... 258
Figure 16.7 Program Linkage Using External Program Calls ... 259

Figure 17.1 Example of Coding an RLABL Definition List ... 300
Figure 17.2 Example of Coding CALL Statements for RBRIDG ... 300
Figure 17.3 Sample COBOL Program Using RBRIDG ... 301

Figure 18.1 Sample Profiled Source Program Listing ... 305
Figure 18.2 OCL Showing //FILE Statement for P#SAMP Counter File ... 308
Figure 18.3 Execution-Time Cost Multipliers for Various RPG Operations ... 308
Figure 18.4 Sample Output from Utility RPGDUMP ... 310
Figure 18.5 Symbol Table Creation Prompt Screen ... 311
Figure 18.6 Sample Symbol Table Source Member ... 311
Figure 18.7 Example of RPG Code Using SUBRTD ... 312
Figure 18.8 RPG Dump Formatter Prompt Screen ... 312

Figure 19.1a RPG Program Using Externally Described Files ... 325
Figure 19.1b Code Generated by 400RPG ... 326
Figure 19.2 400RPG External File Description Member for BANK File ... 328
Figure 19.3a Using RPG II to Describe a Multipart Key ... 329
Figure 19.3b Using a Data Structure to Describe a Multipart Key ... 329
Figure 19.3c Using KLIST to Assemble Fields of a Multipart Key ... 329
Figure 19.4a Indicator-Laden RPG II Code ... 331
Figure 19.4b 400RPG Code Using AND/OR Operations ... 331
Figure 19.5a Sample S/36E Program ... 334
Figure 19.5b Native Program Called by Program in Figure 19.5a ... 334
Figure 19.5c Mapping Parameters with a Data Structure ... 334

Figure 20.1 How S/36 Objects Relate to Their AS/400 Equivalents ... 342

Figure 20.2 Internal Organization of the S/36 Environment 343
Figure 20.3 Equivalent S/36 and AS/400 Models 347
Figure 20.4 AS/400 Memory Requirements Worksheet 347
Figure 20.5 AS/400 Model Capacity Chart (MB) 348
Figure 20.6 AS/400 Disk Requirements Worksheet 349
Figure 20.7 IBM Program Product Space Requirements 349

Figure 21.1 Comparison of UNIBOL and Open RS/36 Features 355
Figure 21.2 Comparison of UNIBOL and Open RS/36 Facilities 357
Figure 21.3 UNIBOL Versus S/36 Performance Benchmarks 359
Figure 21.4 Screen from UNIBOL's Programmer Interface Environment (PIE) 360
Figure 21.5 UNIBOL Pricing 363
Figure A S/36 Machine Language Instruction Set 372

Introduction

THE SYSTEM/36 IS DEAD! That, at least, was the claim made by a columnist in the December 1986 *System User* magazine, when the S/36 was barely three years old. Needless to say, the announcement was a little premature. Six vigorous years later, more than 300,000 S/36's are running worldwide — with new CPUs still being sold by IBM and its resellers. As the S/36 closes out its tenth year of profitable computing, nobody is yet quite sure when the system is going to finally lie down and die. The AS/400, certainly a sexier and more powerful machine, has somehow failed to pull cheerfully persistent S/36 users away from what they see as a working, *paid-for*, business solution. This S/36 community seems determined to remain active for several more years.

This book was written for that still-thriving community. Whether you're a system operator, programmer, MIS director, or consultant, if your job includes managing a S/36, this book will help you. We call it the *Desktop Guide* because it's designed to stay off the shelf and by your side — ready to provide solutions to the kinds of problems that crop up regularly in an active S/36 shop. The material is arranged logically into seven sections: system internals, hardware, external program calls, disk data management, performance tuning, advanced topics, and future directions. But because we expect you to access this book randomly, like an indexed file, we've incorporated a number of special reference tools. First, a task guide (see "How to Use This Book") provides a road map to follow for visiting chapters of interest in common problem situations. Second, *performance notes* in the margins highlight useful tips and techniques, to let you get the "good stuff" by skimming. Third, embedded *technical notes* amplify details that might otherwise get swept under the rug. Fourth and finally, the index — an afterthought in some other books — has been prepared with special attention to detail, to help you easily ferret out obscure facts.

However, we want to provide more than just an encyclopedia. To that end, we've included a diskette chock-full of useful utilities that can make your S/36 life easier. These aren't just reruns of programs published in *NEWS 3X/400* (all right, a *few* are reruns, but they're *good* reruns!). Most of the utilities are brand new, original tools appearing here for the first time. We created them to fill in some of the remaining gaps left by IBM in its headlong dash to the AS/400. A directory of these utilities, along with page references to the text describing them, appears at the end of this section (see "How to Use the Diskette").

From the beginning, we tried to write a book that would answer as many questions as possible, explain as much valuable detail as possible, and avoid as much pablum as possible. We don't expect anybody to use everything in this book; but if it solves some of *your* problems, we've succeeded.

HOW TO USE THIS BOOK

"Would you tell me, please, which way I ought to go from here?"
"That depends a good deal on where you want to get to," said the Cat.

— Lewis Carroll
Alice's Adventures in Wonderland

There are many ways to approach S/36 problems and solutions; likewise, there are many ways to approach this book. Here are a few suggestions. If you want to:

Make the machine run faster right now

Read Chapter 5, which will tell you to go buy as much memory as you can stuff into your CPU. Then go to Chapter 4 to determine how much memory that is for your model S/36 (*Warning*: you might find yourself upgrading models before you're done). You can't make your machine faster without plenty of memory; these chapters show you why.

Purchase hardware upgrades

Go directly to Chapter 4, the definitive catalog of S/36 hardware options. If you weren't planning to buy memory, read Chapter 5. Before writing the check for your upgrade, review the upgrade considerations in Chapter 6; you could save a bundle.

Do serious performance tuning

Skip to Chapter 13, which explains the all-important prerequisite to performance tuning, good measurements. If you haven't thought twice about memory yet, read Chapters 14 and 5 next. If you're looking for program-level tuning, study Chapter 3 to understand Disk Data Management, then read Chapters 11 and 12 for tips on reducing the disk bottleneck. Chapter 18 explains profiling, which lets you analyze program performance at the source statement level.

Improve your software development environment

Check out Chapter 6, which enumerates ideas for making software development more efficient through smarter use of hardware. Chapters 18 and 19 provide information about better development and debugging tools (and the diskette includes some tools you can use right away). We're assuming that you already have a good full-screen editor — either FSEDIT from IBM's POP product or the Blue Iris editor from Iris Software (916-893-4747, $475). We consider POP (Programmer/Operator Productivity Aid) an essential prerequisite tool — it greatly assists system management and software development. If you don't have it, get it (IBM program number 5799-BRJ, $500).

Write better, faster programs

Read Chapter 7, an introduction to External Program Calls (EPC). EPC is the single most important technology you can apply to application design today. With it you can achieve subsecond response time with programs that are easier to understand and maintain. If you're already using EPC, scan the comparison of EPC products in Chapter 8 — you may decide to change EPC vendors. If you're sold on EPC, but need a jump-start on modular design and implementation concepts, check out Chapter 9, which presents a case study and rules to live by.

Get new programs up and running sooner

Chapter 18 explains how to shorten the most unpredictable phase of software development: testing and debugging. Two tools included on diskette — a profiler and a dump debugger — help you ensure test coverage is complete and track down elusive intermittent bugs. Don't miss the "Golden Rules" of software testing, a wealth of testing knowledge from acknowledged industry experts.

Do things beyond the ability of RPG and COBOL

Go directly to Chapter 17, a ready-to-roll library of assembly language subroutines that open up areas of the S/36 you thought were IBM's private property. If you think that assembler routines are a bad move for portability reasons, review the discussion on portability in this chapter and the rules for upward compatibility in Chapter 19. It is possible to use assembler without painting yourself into a corner. *Special note to COBOL users*: Be sure to read about the COBOL Bridge at the end of Chapter 17 — it makes most RPG-oriented assembler routines accessible to your COBOL programs.

Use CACHE effectively

Read Chapter 15 to get the definitive scoop on CACHE operation, including a utility that lets you monitor CACHE performance without all the fuss and bother of learning SMF. You may need more memory to take advantage of CACHE; look at Chapters 5 and 14 to find out.

Improve disk performance

Look through Chapters 11 and 12 for specific procedures that reduce disk time and space overhead. Chapter 10 explains blocking and how to use it to best advantage without modifying your programs. Chapter 3 lays out Disk Data Management theory you'll need to really understand disk performance.

Improve response time

Read Chapter 16 carefully. It explains how to measure interactive response time using a utility provided on diskette, so you'll know when response time has improved. The chapter goes on to show you where you can change

applications to reduce response time, and points out how External Program Calls can be the big lever for speeding user interaction.

Make Disk Data Management sit up and bark

We mean, of course, wringing every drop of performance from DDM. To do that, you must understand DDM internals intimately. Fortunately, it's not all that hard, as you'll discover after reading Chapter 3. Chapters 10, 11, and 12 will then show you all the tricks, and a host of utilities on diskette will have you leading DDM around by its, uh, node.

Do the right thing for future migration

Skip straight to Chapter 19, which gives important guidelines for making your applications portable, whatever your future platform. You might be surprised at what the real issues are (for example, you probably think assembler language programs are a big problem, but they're not).

Learn about future platform alternatives

The Big Question: Where do we go from here? Chapter 20 covers the AS/400 from a S/36 user's perspective. It will give you some mental scaffolding to understand what you're evaluating, and all the details you'll need to pick an AS/400 of appropriate size and mass for your needs. Chapter 21 examines the new midrange contender: the RS/6000. You'll read about the two primary Unix migration alternatives and what pros and cons you should consider in your decision.

Be a S/36 guru

It's still fashionable. And not too difficult. Read Chapters 1 and 2 (which we included to give our book intellectual heft), buy a pocket protector, and sign on to *NEWS 3X/400*'s electronic bulletin board, NEWSLINK, every day.

HOW TO USE THE DISKETTE

The diskette packaged with this book contains source and object code for the utilities and subroutines described in the text. We've divided the machine-readable material into three libraries: DT36X contains ready-to-run object code and procedures for all the standalone utilities; DT36R contains the R-modules for assembly language subroutines; DT36S contains all non-assembly language source code. The use of three separate libraries lets you install only as much machine-readable code as you want to use. If you just want to use the standalone utilities, install only the DT36X library; if you want to use assembly language subroutines in your RPG or COBOL programs, install DT36R; if you want to modify any of the utilities, install DT36S.

To install a library, simply insert the Desktop Guide diskette into the diskette slot (slot S1 if you have a magazine drive) and type the command:

```
RESTLIBR library
```

where *library* is DT36X, DT36R, or DT36S. SSP will then restore the library on your hard drive. You can use any of the standalone utilities by making DT36X your current library using the `SLIB DT36X` command. Then simply key in the name of the utility you wish to run (e.g., MMETER). Alternatively, you could copy the contents of DT36X to your #LIBRARY by typing:

```
LIBRLIBR DT36X,#LIBRARY,LIBR,ALL
```

Now the utilities will be available from any terminal no matter what library is current.

To use the assembly language routines once you've loaded them, specify DT36R for the *subroutine library* name parameter on the RPGC or COBOLC statements. For RPGC this is parameter 16:

```
RPGC MYPROG,MYLIB,,,,,,,,,,,,,,DT36R
```

For COBOLC, you also specify DT36R in parameter 16:

```
COBOLC MYPROG,MYLIB,,,,,,,,,,,,,,DT36R
```

Remember that for COBOL you must use the RBRIDGE routine to access assembly language routines written for RPG (see Chapter 17, page 300, for an explanation of the RBRIDGE routine).

To locate any of the utilities or subroutines described in the text, refer to the following alphabetical library directories, which list the name and brief description of the utilities or subroutines contained in each of the libraries and provide a page number to turn to for detailed usage instructions.

DISKETTE CONTENTS

Library DT36X: Executable Standalone Utilities

Name	Page	Description
CACHIQ	243	Displays real-time cache utilization and performance statistics
FILEXTND	196	Changes or retrieves a file's extend value without re-creating or copying the file
INDEXDR	167	Index Doctor. Analyzes indexed files and alternate indexes and prints a report describing index health
KEEPOPEN	173	Keeps large indexed files open to maintain the storage index in main storage, speeding program initiation

Name	Page	Description
MMETER	234	Displays real-time memory utilization statistics for the entire system
PROFRPG	304	Performs statement-count profiling on RPG program
REORGX	198	Reorganizes a file and automatically re-creates all of its alternate indexes
RPGDUMP	308	Prints formatted contents of an RPG dump file
RTIMER	249	Measures and reports response-time values for all workstations
SETCACHE	245	Dynamically changes cache values from evoked or JOBQ jobs
SHOWUR	181	Shows which programs are using records in a given file, which records are being held for update, or which records are waiting for another program to finish an update
SLOWKS	199	Reports whether or not a given keysort is being performed using a disk work file (fast) or entirely in-memory (slow) due to lack of available disk space
SNAP	219	Displays real-time CPU and device usage statistics for executing programs

Library DT36R: Assembly Language Subroutines

Name	Page	Description
RBRIDG	300	COBOL bridge to RPG assembler routines
SUBR#D	308	Subroutine used internally by RPGDUMP utility
SUBR$C	265	String concatenate
SUBR$F	265	String search within a field
SUBR$S	234	Subroutine used internally by MMETER utility
SUBR$X	267	Substring extract from a field
SUBRAT	268	String adjust: left, right or center
SUBRBX	269	Convert between binary and hex
SUBRCO	280	File close and open
SUBRCP	293	Return cursor position for a WORKSTN file
SUBRCQ	243	Subroutine used internally by CACHIQ utility
SUBRCS	269	Convert between upper and lower case
SUBRCT	291	Change system time or date without IPL

Name	Page	Description
SUBRDT	292	Retrieve SYSTEM and SESSION date formats
SUBRDU	293	Replace DUP key save area for a workstation
SUBREK	294	Dynamically enable or disable command keys
SUBRFA	281	File access, full procedural
SUBRLD	271	Library directory retrieval
SUBRLN	295	PRINTER file current line number retrieval
SUBRLR	274	Library member read by sector
SUBRMG	296	Send an MSG message from within a program
SUBRRN	286	File relative record number retrieve
SUBRPC	296	Position the cursor for a WORKSTN file
SUBRPS	297	Print the current screen
SUBRSG	276	Library source member get
SUBRSM	219	Subroutine used internally by SNAP utility
SUBRSX	298	Return the spool ID for a PRINTER file
SUBRSY	288	System information retrieval
SUBRUF	288	Retrieve users of a file
SUBRUL	289	Retrieve users of a library
SUBRUP	270	Pack or unpack a field
SUBRUR	290	Retrieve records-in-use for a file
SUBRWT	298	Wait for a specified time interval

Library DT36S: Source Code

Name	Page	Description
CACHIQ	243	Cache IQ source code
KEEPOPEN	173	KEEPOPEN source code
MMETER	234	Memory Meter source code
PROFRPG	304	Profiler source code
REORGX	198	REORGX source code
RPGDUMP	308	RPG dump formatter source code
RTIMER	249	Response time measurement utility source code
SHOWUR	181	SHOWUR source code
SNAP	219	SNAP source code

Section I

System Internals

"Where shall I begin, please your Majesty?" he asked. "Begin at the beginning," the King said, gravely, "and go on till you come to the end: then stop."

—Lewis Carroll
Alice's Adventures in Wonderland

Understanding the inner workings of your computer isn't essential to running the machine any more than understanding engines is essential to driving a car. It's a sure thing, though, that at least a passing familiarity with hardware and software architecture will help you in the long run. If your immediate goal is to improve response time or to write applications, you can safely skip the material in this section — nothing here will make or break your ability to tune the system or write good application programs. But if you want a good understanding of the fundamental mechanisms that underly the S/36, you'll find it right here.

Chapter 1 is the definitive description of S/36 hardware architecture: an under-the-covers look at CPU internals not found in any public IBM publication. Chapter 2 presents an equally detailed dissection of S/36 memory management that will serve you well as you dig through pages of SMF reports, hot on the trail of improved performance. Chapter 3 unravels the mysteries of Disk Data Management — the information pump through which all your application's data must pass — and dispels a good many old programmers' tales in the process.

All of this material might be considered above and beyond the call of duty for the average programmer. But then, you aren't average — you bought this book, didn't you?

Chapter 1

Introduction to S/36 Architecture

Computers, like skyscrapers, are built from the basement up; sometimes it takes years to reach the top floor. Most existing commercial computers have risen slowly from established architectures, and the System/36 is no exception. It stands on the foundation wrought by the IBM System/3, System/32, and System/34. But System/36 users should not fear that their trusted systems will become lost in the rubble. An investigation of the history and architecture of the System/36 reveals that System/36 reliability and cost efficiency ensure several more years of life, despite the availability of newer systems such as the AS/400.

Examining the S/36's inner parts reveals its evolution. Its unique parallel processors and modular components speak of several design phases. There is much to learn. But interesting as this knowledge might be, of what practical use is it? Why should anyone be interested in how a machine works internally? Certainly the S/36 can be programmed and operated without such detailed knowledge, just as a car can be driven without understanding what goes on under the hood. However, for the driver, a knowledge of auto mechanics comes in handy when comparison shopping for a new car or when "tuning up" the old car. Knowing something about the inner workings of your computer provides the same kind of benefit.

Understanding what happens under the computer's covers is especially helpful when planning for the inevitable upgrade to a newer platform: you can take steps today to make your applications more portable while retaining good performance. And programmers who understand the basic operation of a computer can design programs that take advantage of the computer architecture's strong points.

Genealogy

A discussion of present-day S/36 architecture must begin with a look at the architecture of its predecessors. Figure 1.1 summarizes the hardware developments contributed by each generation in the S/36 ancestry.

The S/36 architectural line began in 1969 with the S/3, one of the first computers for small businesses. The S/3 was revolutionary in one sense: it offered a fast commercial instruction set at a time when most computers relied on a scientific instruction set. A computer's instruction set consists of the lowest level machine language orders it can carry out. "Add binary," "load register," and "branch to address" exemplify machine language instructions. The scientific machine instruction set used by most computers of that era

Figure 1.1
System/36 Genealogy

Machine	Major architectural contributions
System/3	• Commercial machine language instruction set, including: multibyte, memory-to-memory instructions (up to 256 bytes per instruction); variable-length decimal; and arithmetic no hardware multiply/divide • Address Translation Register (ATR) technique for multitask memory management.
System/32	• Separate processor to handle I/O (Control Storage Processor) • Microprogram emulation of System/3 processor (Main Storage Processor) • Single program execution only • 27 MB disk drive technology (Gulliver)
System/34	• Hardwired implementation of System/3 processor • Task switching and address translation handled by CSP • Significant operating system functions implemented as Supervisor Calls • Scientific instruction set (for BASIC and FORTRAN) in microcode • Four-line communication controller (MLCA) to handle polling and teleprocessing I/O • 64 MB disk drive technology (Picollo)
System/36	• Faster main storage and control storage processors • 8 MB real main storage addressability • 192 K translated (region) main storage addressability • 128 K control storage (double that of the System/34) • Two-byte wide, bidirectional channel • Improved overlap of CSP and MSP operation • Fast task-switching hardware and multiple ATR groups • Virtual memory management for system transient routines • New MSP instructions for ease of programming • W/S controller optional to reduce cost of entry level machines • Word Processing Text Mode support for workstations • Data Storage Controller to handle disk/tape/diskette data transfers • Eight-line communication controller (ELCA) to handle polling and TP I/O • RLL/ECC disk drive technology to eliminate write verifies

performed calculations on binary numbers in registers (high-speed, scratchpad memory). For scientific programs this type of instruction set provided much needed speed. Programs computing Laplace transforms or orbital velocities ran like the wind.

Business programs, however, have little to do with binary numbers or registers; they work instead with decimal numbers and variable length fields. A traditional scientific instruction set was ill-equipped to deal with these factors because it was constantly converting commercial formats to binary formats and vice versa. Consequently, payroll and inventory applications run on

computers with a scientific instruction set used up much processing time on data conversion and ran like glaciers.

In contrast to many of its contemporaries, the S/3 commercial instruction set could perform arithmetic operations directly on decimal numbers in memory — no intermediary registers were necessary. It also could, in a single instruction, manipulate a field of data up to 256 characters long (no more byte-by-byte translation to binary). With the S/3's multi-byte, memory-to-memory instructions, S/3 programs did not spend much time converting data, and business application performance improved.

The speed of the S/3 processor was carried over to the S/32 (sometimes called the "bionic desk" because it was an all-in-one unit), which was introduced in 1973. With the S/32, IBM heralded a new hardware technology that allowed two CPUs to function side-by-side in one machine.

One CPU, or processor, was a microprogrammed version of the S/3. This processor wasn't as fast as a S/3, but it executed the same instruction set, which meant IBM could reuse much previously developed system software. The other processor, given the job of handling all contact with peripheral devices and the outside world, was a real innovation. This second processor ran its own dedicated program using its own dedicated memory. Although in the early 1970s the S/32's dual processors represented a hardware advancement, this system was limited by its capacity to run only one program at a time.

Four years later, IBM announced the S/34, which continued the S/32 philosophy of using two processors. However, the microprogrammed processor was replaced by a much faster "hardwired" version. This processor, now faster than the fastest S/3, could run multiple programs simultaneously. The second processor, also a holdover from the S/32, took on the job of managing memory and dispatching tasks for the hardwired processor. (IBM included in this processor a scientific instruction set, emulated by microcode, which allowed faster execution of BASIC and FORTRAN programs.) Local terminal management was relegated to yet a third processor. A fourth processor was available as an option to support outside communications — the first time a processor was offered optionally. Extensive use was made of another development, the ATOM (A Tiny Optimized Microprocessor), to directly control the system printer and MICR (Magnetic Ink Character Recognition) devices.

These developments led to the S/36, which appeared in 1983. In many ways, it is a radical departure from the S/34. For example, memory addressing was re-engineered for the first time since the S/3. The two main processors also were improved and the selection of optional processors expanded.

Internally, the S/36 supports several different processors. Figure 1.2 summarizes their names, functions, and characteristics. The Main Storage Processor (MSP) is really a hardwired S/3 CPU with a few new instructions. The Control Storage Processor (CSP) controls the overall operation of the entire

Figure 1.2
System/36 Multiple Processors

Function of Processor	Internal Technology	Instruction Set	Execution Speed in MIPS	Address Space	Memory Access Time (ns)
Main Storage Processor	MSP	Enhanced System/3	0.36	8 MB	200
Control Storage Processor	CSP	Register-to-Register	1.6	128 K	200
Workstation Controller	CSP/I	Register-to-Register	1.6	128 K	200
Data Storage Controller	CSP/I	Register-to-Register	1.6	128 K	200
Eight-Line Comm Adapter	CSP/I	Register-to-Register	1.6	128 K	200
Printer Controller	ATOM	Register-to-Register	1.1	128 K	200
Magnetic Ink Character Recognition	ATOM	Register-to-Register	1.1	128 K	200
Local Area Network	PC/AT	Intel 80286	1.2	640 K	150

machine. It runs a dedicated program in its own memory (control storage) which may be either 64 K or 128 K. A third kind of processor has the same instruction set and organization as the CSP, but it is used as a dedicated controller for certain input/output (I/0) operations. IBM has designated it the CSP/I; one each is found in the Workstation Controller, Multi-Line Communications Adapter (MLCA), Eight-Line Communications Adapter (ELCA), and Data Storage Controller. If a 3262 printer or Magnetic Character Reader is attached to the system, ATOMs will control these devices. To better understand the function of each of these processors, let's first examine the two main processors (MSP and CSP) and then discuss the optional processors.

Main Storage Processor

The S/36 Main Storage Processor (MSP) runs SSP programs and user applications through the S/3-based commercial (memory-to-memory) instruction set. That is all the MSP does. It has no control over which programs are executed. It has no direct contact with the outside world. (When the MSP must perform I/O operations, it submits a request to the CSP, which handles contact with

the outside world.) And the MSP executes only .36 million instructions per second (MIPS). This rate might seem slow when compared with other computers, but because the MSP doesn't concern itself with I/O or task management, it is free to concentrate on the job at hand. This freedom makes up for the MSP's apparent lack of horsepower.

The MSP in the current S/36 can address up to 8 MB of memory, or main storage — seemingly small by today's standards, where an average PS/2 might be configured with up to 16 MB. However, the S/36's implementation of virtual memory (VM) lets you run up to 128 MB of applications simultaneously. Chapter 2 describes S/36 memory architecture, including VM, in detail.

Control Storage Processor

The Control Storage Processor (CSP) interfaces with peripheral devices, manages MSP memory and swapping, and controls the execution of the MSP. The CSP also provides special computational services to the MSP, including high-level operating system operations such as queue management and intertask communications. Through judicious task and memory management, the CSP tries to keep the MSP operating at maximum efficiency. Because the CSP is not working on business programs, it uses a more applicable register-to-register instruction set, which allows the CSP, running at speeds of 1.3 to 1.9 MIPS, to juggle many jobs at once. The services provided by the CSP simplify the programming involved in the SSP and take advantage of the four-fold speed advantage CSP has over MSP for time-critical functions.

There are three versions of the S/36 CSP. Machines shipped before October 1984 contain a Stage 1 CSP, which runs at 1.3 MIPS. Machines shipped after October 1984, including all 5362 and 5364 processors, contain a Stage 2 CSP, running at 1.6 MIPS. CPUs on the 5360 model D, and all 5363s, use a Stage 3 CSP, running at 1.9 MIPS. On small machines the performance difference between Stage 1 and Stage 2 processors is insignificant because the CSP is rarely running at anywhere near its rated capacity. However, on large 5360s running many workstations or DisplayWrite/36 jobs, the CSP may be fully utilized, and the Stage 3 processor improves performance significantly.

The S/36 CSP has a number of enhancements over the S/34 version. In addition to being faster, the S/36 CSP processes more requests in parallel with the MSP than did its predecessor. It also recognizes many new Supervisor Call (SVC) instructions, which perform operating system functions for the MSP. Included in these new SVC instructions is a "storage mapping" service, which allows SSP programs (e.g., data management) easier access to buffers in a user application.

The S/36 CSP also contains a larger control storage area than the S/34 CSP. Because the S/34 CSP contained only 64 K of control storage, control storage programs that couldn't fit in this space were read in from disk as "transients"

when required. The S/36 CSP can contain either 64 K or 128 K of control storage. (The extra storage is used to contain Workstation Controller (WSC) code if the WSC function is inboard, or to simply keep more CSP routines resident if the WSC function is outboard.) In addition to the extra control storage, the S/36 CSP offers a new Virtual Address Facility in its memory management function. This facility allows any number of MSP transients to run from the user area instead of bottle-necking in a single transient area as they did on the S/34.

An interesting and useful service provided by the CSP is the Alter/Display facility. When the MSP STOP button is pressed on the service panel, a special menu appears at the system console. This menu allows a programmer or service technician to examine and modify any location on disk, in main storage, or in control storage. This kind of tool, when it is available at all on single-processor machines, is usually implemented as a large and complex control panel. The S/36's "soft" control panel is much easier to use and provides a wider range of functions. For example, an Address Compare Stop feature can be used to stop the MSP when a certain disk or memory address is referenced or changed to a specified value. The MSP is in a suspended state while Alter/Display is being used; processing resumes at the point of interruption after exiting the Alter/Display menu. This capability is invaluable for tracking down difficult system bugs.

Optional Processors

As options, you may install other processors that take care of additional tasks. A Workstation Controller (WSC) processor deals with local workstation input/output; a Data Storage Controller (DSC) processor mediates data transfers between disk and slower devices such as diskette and tape; an MLCA (Multi-Line Communications Adapter) or ELCA (Eight-line Communications Adapter) processor handles polling and protocol for multiple communication lines; a Local Area Network processor supports IBM's Token-Ring LAN.

The Workstation Controller. An interesting difference between the S/34 and the S/36 is in the workstation controller. Every S/34 had a dedicated CSP/I with 32 K of control storage to poll workstations, process keystrokes and handle field attributes like right-adjust, zero fill, and check digits. The workstation expansion feature to support more than eight devices was simply a memory expansion of the WSC to 64 K.

Not every S/36, however, has a dedicated WSC. On all 5364 models, and on 5362 models without the workstation expansion feature, the WSC function is performed "inboard" by the CSP. Because the CSP and WSC both use identical processors, adding WSC tasks to the CSP's workload wasn't hard to do, and it allowed IBM to produce machines with full S/36 functionality at a lower price. (For a comparison of features among S/36 models, see Figure 1.3.) These models do not suffer a performance loss because the CSP has

Figure 1.3
Comparison of Features Among S/36 Models

Model	Disk Configurations	Access Time	Memory Capacity	Workstation Controller	Data Storage Controller	Comm Controller	Diskette	Tape**
5364	40,80 MB up to 2 spindles	60 ms	1 MB	N/A	N/A	PC (1 line)	5.25"	6157
5363	68 to 425 MB up to 2 spindles	35 ms	2 MB	N/A	Optional	SLCA (2 lines)	5.25"	6157
5362	30 to 520 MB up to 4 spindles	35 ms	2 MB	Optional	N/A	SLCA, MLCA (4 lines)	8"	6157
5360	30 to 1438 MB up to 4 spindles	35 ms	7 MB	Dedicated (optional 2nd avail)	Optional with tape attachment	SLCA, MLCA, ELCA (8 lines)	8"	6157 8809
9402*	160 to 640 MB up to 4 spindles	35 ms	2 MB	N/A	Optional	SLCA (2 lines)	5.25"	6157

* Also known as the AS/400 model 9402, but actually is a S/36. IBM marketed this machine for a short time as the AS/Entry.

** The 6157 tape drive is a streaming cartridge unit with 60 MB capacity. The 8809 tape drive is a reel-to-reel unit for mainframe data exchange.

enough additional capacity to easily take on the extra load. Larger 5362s and all 5360s have an "outboard" WSC that relieves the CSP of handling more extensive local networks.

Both the inboard and outboard WSC implementations support the new "word processing mode" for local workstations. This mode adds such functions as indentation, margin control, tab entry, and word wrap — functions used by DisplayWrite/36 to provide a user interface better adapted for word processing than the fixed-field format of data processing mode. Because these features are under the direct control of a CSP or CSP/I, they have consistently fast response time, regardless of the load on the MSP.

Remote devices, such as the 5251 Model 12 workstation and the 5294 control unit, also contain workstation controllers. In the 5251 Model 12, the WSC program is fixed in Read Only Memory (ROM) and cannot be changed. It is unable to support word processing mode. The 5294 does not contain a fixed WSC program. Instead, the host S/36 downloads WSC microcode when the 5294 goes on-line. Thus, word processing mode functions are available, unlike remote WSC that have fixed micro code.

Data Storage Controller. The S/36 supports up to two tape drives —

something unavailable on the S/34. Tape drives are attached to the system through the Data Storage Controller (DSC), which can autonomously transfer files from disk to tape without the intervention of either the MSP or CSP. In fact, the DSC also can mediate transfers between disk and diskette, diskette and tape, or disk and disk.

When a S/36 does not have a DSC, data is transferred between devices on an internal two-byte-wide path called the "channel." This same path is used for intercommunication between the MSP, CSP, and other processors. With all these devices competing for use of the channel, a sudden high-volume transfer of data can result in a logjam of information, degrading system performance significantly. The DSC operates "below" the channel, communicating directly with the devices over its own private data path. This capability reduces access contention on the main channel and eliminates the degradation that normally occurs with large file transfers. The S/34 experienced tremendous response-time degradation when transferring files between diskette and disk, or disk and disk.

To operate efficiently, the DSC contains two 16 K buffers. It initially fills both buffers; then, after one buffer is written to the output device, it starts refilling it while the second buffer is being written. This double-buffering improves the output transfer rate significantly and allows the tape drive to run in streaming (high speed, nonstop) mode. When a DSC transfer is requested, the CSP notifies the DSC where the source and destination files are and the DSC takes over, interrupting the CSP only when a diskette or tape must be changed.

Because the DSC only relieves congestion on the main channel (it doesn't actually move the data faster), no appreciable performance improvement will be noticed unless the system is heavily loaded. The DSC can only make response times more consistent. The DSC also is limited to performing one device-to-device transfer at a time. If the DSC is engaged in a transfer and the MSP requests another transfer, the second request will be queued until the DSC is free. The one exception to this rule is if the DSC is processing a tape transfer and a request for a diskette transfer is made. Because the tape transfer could require a long time (especially if the operator doesn't change reels when prompted), the diskette request is processed immediately using the main I/O channel.

MLCA and ELCA Processors: A S/36 with one or two communications lines (Single-Line Communications Adapter — SLCA) uses the CSP to poll the lines, handle bottom-layer protocol, and buffer data transfers. One line presents no problem, but two lines can put an unwieldy burden on the CSP, which is forced to drop everything it's doing to service the high-priority communication interrupts. When more than two lines are installed, the MLCA (now available only on the 5362) and ELCA processors do the dirty work. These processors are essentially identical — the ELCA is more recent and the

only product currently available on newer 5360s.

Because both communications processors are a dedicated CSP/I, they support data rates much faster than the SLCA. They also assume the responsibility for polling terminals, processing protocol messages, computing checksums, retransmitting buffers, and for the lower layers of SDLC protocol. A machine with MLCA or ELCA installed will experience much less degradation than a machine using SLCA.

Local Area Network Processor. The S/36 supports IBM's Token-Ring Local Area Network (LAN) through a specially attached PC/AT. The Token-Ring network ports appear as communications lines 9 and 10, and the Token Ring runs only at the 4 million bits per second (mbps) data rate; the 16 mbps Token Ring isn't supported.

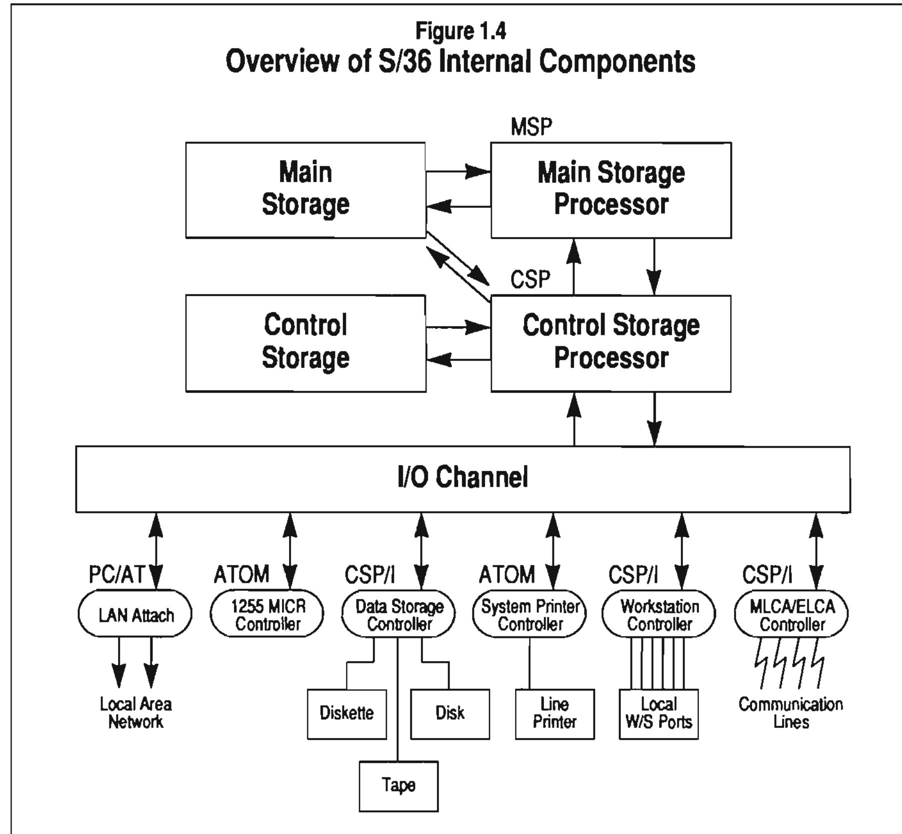
IBM's Multiple Processor Advantage

A computer architecture, such as the S/36's, that uses multiple processors faces a significant problem: how to use the processors efficiently. The goal is to get some degree of parallel operation without unnecessarily holding up the execution of any one task. Traditional approaches to the problem treat all processors equally either by running separate application programs on each processor or by interleaving instruction execution among processors. IBM has taken a different tack with the S/36 by assigning each processor a specific, dedicated job and by designing for each a unique instruction set optimized for the tasks at hand. Figure 1.4 diagrams the major components inside the S/36 and shows how they are interconnected.

The usefulness of the multiprocessor architecture is demonstrated in the analogy that single-processor computers suffer from the same problems as single-engine airplanes: a shortage of options. When the engine quits on a single-engine airplane, there are no options from which to choose. The important decision has been made for you by the engine: the aircraft is going down. The engine in a single-processor machine can stop, too, when an invalid instruction is encountered, or when a hardware error occurs. When such an event happens, the computer often is headed in the same direction as the airplane.

The multiprocessor S/36, like a multi-engine aircraft, recovers somewhat more gracefully from serious system failures. If the MSP tries to do something crazy, the CSP gets control and executes an error-recovery procedure. Often, the error-recovery program needs only to cancel the offending task before resuming the work in progress. Sometimes even this step is not necessary because the problem can be corrected while the MSP waits. For example, if the MSP runs into a parity check (memory failure) in a main storage memory card, the task is canceled and the 2 K page of memory is taken off-line to protect other tasks from the damaged memory. Likewise, if a disk sector is found to be unusable, the CSP automatically assigns a spare from a special supply of

Figure 1.4
Overview of S/36 Internal Components



extra disk sectors, then lets the MSP proceed as if nothing happened. The CSP also keeps a detailed log — the Error Recovery Analysis Report (ERAP) — of any problems it detects for later perusal by a customer engineer.

The interdependence between the CSP and the MSP is especially important because during a typical processing day several programs compete simultaneously for use of the MSP. Competition for the MSP means the CSP must make many decisions about when to run which program. The process of allowing a program (task) to run, then stopping it and starting up another program, is called "task-switching."

A typical task-switching scenario might proceed as follows: when the MSP must perform some I/O operation, it makes a request to the CSP via a Supervisor Call instruction. The program that requested the I/O must now wait. The CSP selects another program that is ready to run and starts it, then schedules the I/O operation for the first task, thus "switching" the tasks. The steps that take place when a task switch occurs bear examining because a

major advantage of the S/36 over single-processor systems hinges on how these steps are carried out.

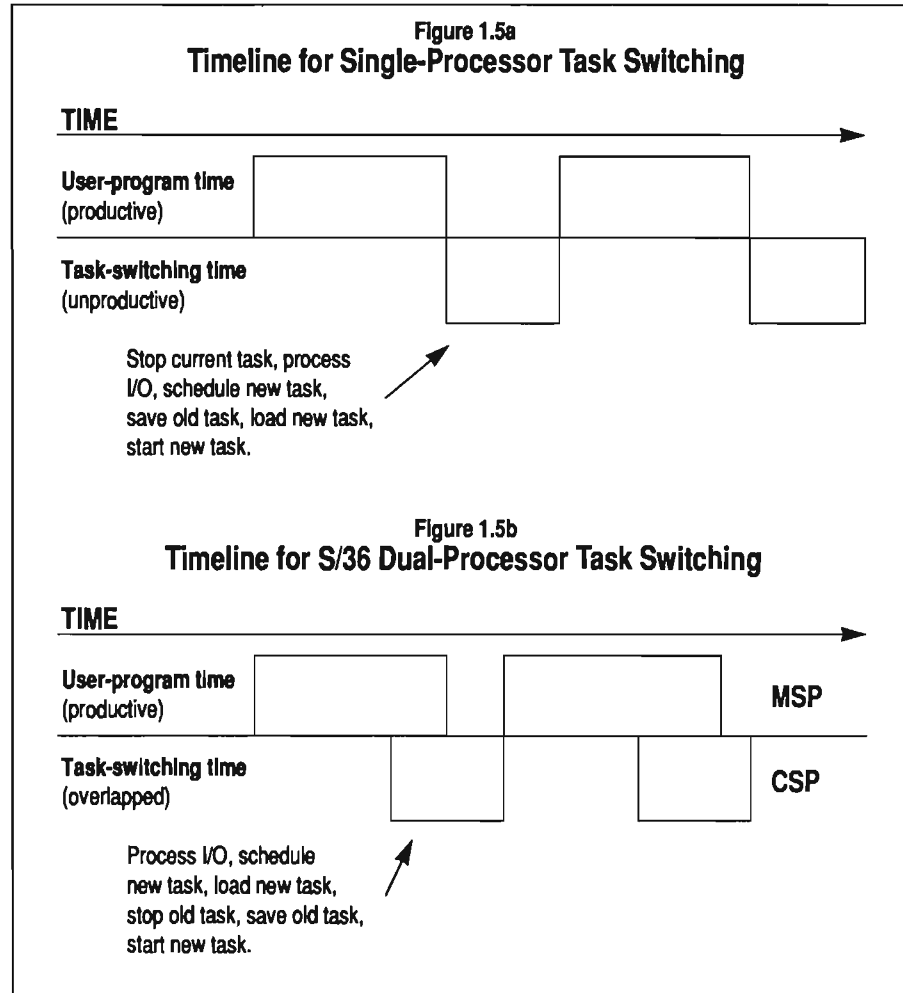
Many computers, including the S/36, use an "I/O-driven" mechanism for switching tasks. That is, when the execution of one task is interrupted to perform an input/output operation, the machine switches to another task. This switch makes sense because most I/O operations are quite slow when compared with the speed of the processor. For example, a disk read requires about 40 milliseconds; in the same amount of time the MSP could execute nearly 15,000 instructions. Because the task that requested the I/O operation is going to wait anyway, running another task in its stead overlaps the operation of the processor and I/O.

However, task switching is not an instantaneous event. There is a general sequence of events that must occur. First, the I/O operations that caused the interruption must be dealt with: transferring the data and controlling the device. Then the computer must determine which of several tasks should run next and maintain the various queues used to make this decision. After a new task has been selected, the environment of the old task (instruction pointer and registers) must be stored. If the new task has been swapped out to disk, it must be brought into memory. Finally, the environment for the new task must be loaded and execution started at the point of previous interruption.

Figure 1.5a shows the timeline of events for a conventional single-processor computer. Because only one processor is available, when an interrupt occurs, everything else must stop while the task switch is done. If two tasks are run together, chances are that the total amount of time to run them will be longer than if the tasks were run one after the other: more time is required to switch between tasks. When many tasks are running at the same time, task switching can account for an appreciable portion of the total execution time. In fact, as the task load increases, a point eventually will be reached where more time is used up doing task switches than running the tasks themselves. To the user, it appears as though system performance degrades rapidly, out of proportion to the number of tasks. This situation is clearly unhealthy; in most forms of accounting it is referred to as a net loss.

The same timeline for a S/36 is shown in Figure 1.5b. Here, when a task switch must be made, only the CSP is interrupted — the MSP keeps running. The CSP then sets up everything for the task switch. It takes care of I/O handling, determines which task will run next, swaps the task into storage if necessary, and then switches tasks. All time-consuming operations are performed in parallel by the CSP while the MSP continues to process user program instructions. However, the S/36 contains special "fast task-switch" hardware that allows it to save and load the MSP registers quickly, which, in turn, makes the task switch nearly instantaneous.

The MSP is not involved in the details of shifting gears and loses little



time between tasks. Within the space of a few MSP instructions, the old task is stopped and the new task is started. As new tasks are added, the MSP concentrates on running those tasks, and there are fewer abrupt changes in system performance.

The Channel

The I/O Channel is the data path used by the MSP, CSP, other processors, and peripheral devices to transfer data inside the machine. When a disk record is read it is transferred byte by byte down the channel to main storage. The MSP and CSP continue to run during this transfer, but the channel "steals" a cycle

from the MSP when it needs to access main storage. Other transfers work in a similar manner. The S/34 channel was one byte wide, meaning that, for each cycle, one byte could be transferred from one component to another inside the machine. The S/36 channel is bidirectional and two bytes wide. It can transfer two bytes at a time between components or one byte simultaneously in each direction. Because the S/36 channel can transfer twice as much data as the S/34 version, it "steals" only half as many cycles from the MSP, which improves the performance of user programs and SSP functions. Generally, any channel activity takes only half as long as it did on the S/34, greatly reducing internal traffic congestion.

The S/36 channel is actually an intelligent device, not just a data bus. It incorporates a primitive channel processor that executes a limited range of instructions specifically geared to moving data on the bus. Although these instructions are simple, it gives the channel some degree of autonomy: the CSP can issue commands to the channel and then go do other work while the commands are carried out.

Disk Drives

The hard disk technology used on the S/36 is a major advance over previous devices. For this discussion, only the 10SR 200 MB drive will be examined. But other disks (the 30/60 MB used in the 5362 and the 40/80 MB used in the 5364) are similar in operation. All S/36 disks use data encoding to increase reliability and decrease access time.

To better understand S/36 disk drives, let's again contrast the S/36 with the S/34. The S/34 Gulliver (27 MB) and Picollo (64 MB) devices required that, after every write operation, the data be re-read to make sure it was recorded correctly. While this step was handled automatically by the CSP, it was time-consuming: after writing a record, the CSP had to wait for the disk to spin around again to the starting point before the record could be re-read for verification. Thus, write operations were more than twice as long as read operations. Also, while the S/34 re-read technique provided a high level of reliability when the data was written, it provided no recourse if the data was damaged after writing (random damage). Experience with S/34 drives demonstrated that the most common random disk error was a single-bit failure within a byte. Double-bit errors within a byte also occurred but much less frequently.

The 10SR (STAR) uses a data encoding technique called Run Length Limited (RLL) encoding, which eliminates the re-read requirement and achieves reliability by detecting and correcting single- and double-bit errors at read time. The S/36 technique recognizes that bad data could be put on the disk at write time, but that most problems will be single-bit errors. On the S/36, data is not written on disk as a series of fixed-length bytes, as it is on the S/34. Instead, the bytes are encoded into variable-length bit strings containing

Figure 1.6
Run Length Limited Codes for Disk Error Correction

Input bit string	Resulting RLL code
10	0100
11	1000
000	000100
011	001000
010	100100
0010	00100100
0011	00001000

twice as many bits as originally input (Figure 1.6). Six Error Correcting Code bytes are also written for each 256-byte sector. On a S/36, then, because information is being stored redundantly, it is often possible to repair damaged bytes at read time. When a record is read, the encoded data is decoded and an error detection/correction algorithm executed on the result. The mathematics of the algorithm guarantee that any single or double-bit errors can be detected, and that single-bit errors can be corrected. The net effect is that records can be safely written to disk without re-reading for verification.

The Bottom Line

The S/36 is a prime example of building on pre-existing technology effectively. Through extensions to an established architecture, it has the ability to coexist in a distributed environment with other IBM midrange systems. The S/36's modular and general-purpose internal components let you effectively trade off performance, capacity, and cost. The architecture of the S/36 proves that IBM has acted to preserve the history of engineering, software development, manufacturing knowledge, and technical support invested in the S/36.

Where to Learn More

An excellent overview of the System/36 can be found in the IBM technical bulletin *S/36 Internals* (G361009). It outlines general concepts of both software and hardware architectures, provides a lucid explanation of memory addressing, and presents details about MSP/CSP interfaces. This volume is actually one of a series of six "Rochester Technical Bulletins" — the other five cover specific SSP topics:

- S/36 8809 Tape Support (G360-1005)
- *S/36 Performance Monitoring and Tuning* (G360-1006)
- *S/36 Query/36 Design Guide* (G360-1007)
- *S/36 Data Dictionary System Design Guide* (G360-1008)
- *S/36 Advanced Disk Data Management* (G360-1008)

Assembler language programmers will find the following two IBM volumes useful: *Programming with Assembler* (SC21-7908) and *Functions Reference Manual* (SA21-9436). The first book is provided as part of the IBM Basic Assembler Language program product and covers everything a programmer would need to know to write simple assembler programs. A more complete description of the machine, from the programmer's perspective, is found in the *Functions Reference Manual*. Machine addressing modes, instruction formats, and supervisor calls are examined in excruciating detail. The programming characteristics of every device (disk, diskette, tape, printer, display, and communications) also are set forth. Programmers who intend to write special subroutines that access input/output devices directly will be interested in this level of detail.

Technical references useful to system programmers are contained in the trilogy:

- *S/36 Program Service Information* (LY21-0590)
- *S/36 System Data Areas* (LY21-0592)
- *S/36 Program Problem Diagnosis and Diagnostic Aids* (LY21-0593)

These books are available for a charge to any licensed user of SSP. Those who plan to do serious programming in assembler language should have these manuals; they cover debugging facilities, SSP component operation, memory, and disk organization, and the formats of internal SSP data areas.

The manual, *IBM S/36 Control Storage Service Information* (LY31-0650), describes the detailed operation of control storage processor programs. It explains how the CSP communicates with and controls the MSP. The concepts are well illustrated, and an appendix contains several step-by-step examples of CSP/MSP interaction.

For hardcore hardware details, turn to the *S/36 Theory of Operation* manual, which covers detailed internal computer operations at a circuit board level. This manual is one of the large-format customer engineering books shipped with every 5360 system unit. Smaller versions of the S/36 (the 5362 and 5364) are not supplied with this manual.

Chapter 2

S/36 Memory Management

You hear conflicting stories when people discuss how the System/36 manages memory. Some people maintain that the System/36 is a swapping machine; others say it's a virtual machine. Many data processing managers believe System/36 memory architecture is simply a copy of the System/34 with minor changes; likewise, many programmers believe the System/36 limits tasks to 64 K because the System/34 has this limitation. Misconceptions arise from the lack of complete, understandable System/36 memory management information available to busy DP managers and programmers.

Although understanding the low-level details of S/36 memory management isn't essential, it helps you determine whether you have enough memory and whether you are using it effectively. And as you learn more about S/36 memory and how the system manages it, you can design S/36 programs that use memory efficiently.

Main Memory Organization

To develop a picture of S/36 memory, look at a diagram of S/36 main memory (Figure 2.1). Main memory is organized as eight-bit bytes and varies in size from 128 K to 7,168 K, depending on the machine model. Main memory consists of hundreds of integrated circuit "chips" and represents one of the most finite resources of the S/36. Figure 2.1 shows the three areas that comprise the contents of main memory: the fixed nucleus, the variable nucleus, and the user area.

The *fixed nucleus*, which occupies the first 4 K of main memory, contains variables and data structures needed by all components of the S/36's operating system, the System Support Program (SSP). The S/36's dual processors — the Main Storage Processor (MSP) and the Control Storage Processor (CSP) — also use the fixed nucleus to communicate with each other. Because the fixed nucleus is permanently set to the same size and content for all S/36 machines, a programmer or DP manager can do little to influence its effect on performance. However, an assembler language programmer can use the data stored in the fixed nucleus when writing special-purpose performance measurement tools (see MMETER Utility, chapter 14.)

The *variable nucleus* includes the transient area, virtual page table, resident routines, and system queue space. The transient area is 4 K of memory set aside for the very few SSP programs that must run in the variable nucleus. These programs are the task attach and detach, disk file open, diskette

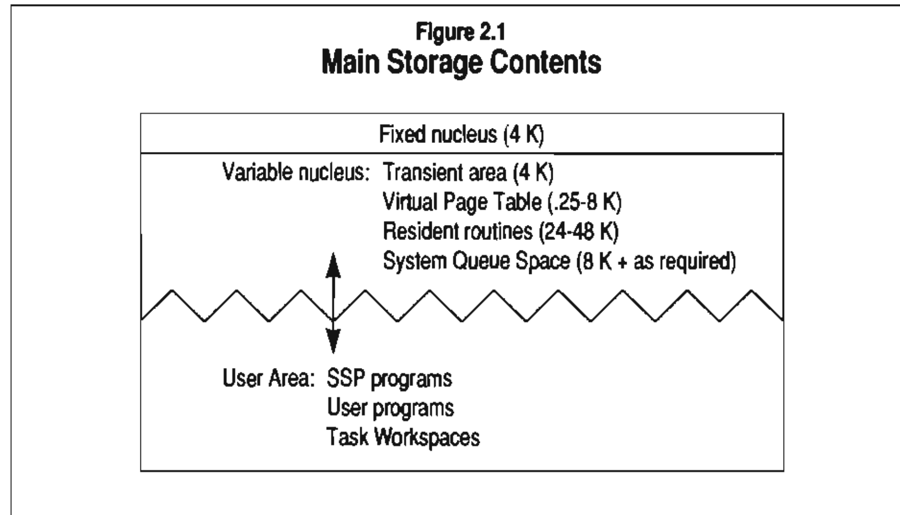


Figure 2.1
Main Storage Contents

> **Performance Tip**
>
> The SSP automatically queues up requests for the transient area, but a high volume of such requests can slow performance significantly by causing many jobs to wait for the transient area. You can reduce transient area contention by designing your applications to minimize new jobsteps (e.g., by using external program calls), thus reducing the need for task initiation/termination and file open/close. Avoiding DDM situations that result in exceptions also helps minimize transient area contention (see Chapter 3).

open, and disk data management exception routines, which run infrequently enough so that contention for the transient area does not slow performance. The virtual page table is used by the S/36 virtual memory (VM) mechanism (described later) to keep the system operating even when memory is overcommitted (i.e., when more programs are running than can fit in memory at one time). Resident routines are a few special SSP programs (disk data management and frequently used parts of workstation data management) that, for performance reasons, are always kept in main memory. System queue space (SQS) is a "pool" of memory set aside for dedicated use by SSP data structures needed to control the system.

> **Technical Note**
>
> Only one system program at a time can run in the transient area. Because file open/close, task attach/detach, and disk data management (DDM) exception handling all run in the transient area, SSP must perform these functions serially. For example, while a task such as // LOAD jobstep is being started, no files may be opened or closed. Similarly, when DDM exceptions occur (e.g., update of a key) no files may be opened or closed, or tasks initiated or terminated, until the exception is handled and the transient area becomes free.

The name "variable nucleus" implies the nature of this region: it varies in size with the amount of work performed by the system. The first three components of the variable nucleus don't actually change size while the machine

is running; the amount of memory they occupy depends on the hardware and software configuration at IPL. Only the last area, system queue space (SQS), ebbs and flows with the varying system load. Because the first three components are "out of your hands," nothing more need be said about their function. On the other hand, your program design and scheduling *do* affect SQS, so a detailed knowledge of the SQS helps you make decisions that improve overall system performance. Later, we'll look at characteristics of SQS that are important from a performance standpoint.

The last, and usually largest, area of main memory is the *user area.* User programs, most SSP programs, file buffers, screen formats, and other objects reside here. One truism applied to computers in general, and the user area in particular, is: "You can't have too much main memory."

Effective memory management rests on your understanding of a few fundamental concepts: real memory, translated memory, and virtual memory. To grasp these ideas, let's look at main memory from a different angle.

Performance Tip

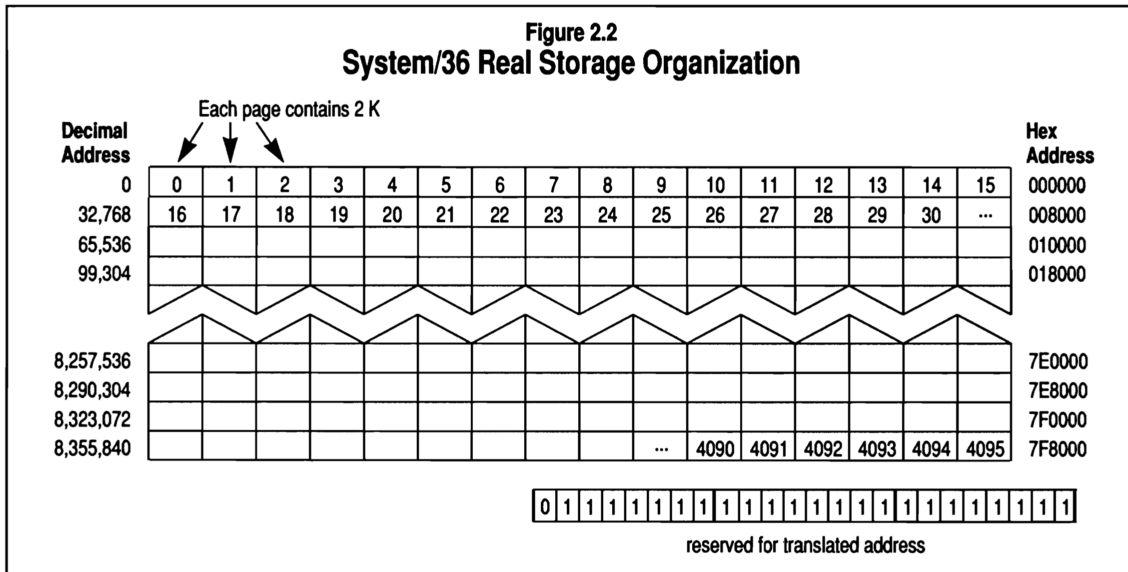
Adding additional memory often drastically reduces interactive response times, making it the easiest and cheapest way to boost performance, especially given the availability of inexpensive used memory (see Chapter 5).

Memory Concepts

Figure 2.2 depicts the main memory address space for the S/36. Memory addressing is the practice of assigning to each location of computer memory a unique address, and using that address when referring to the contents of that location. The S/36 follows the popular convention of dividing memory into eight-bit bytes, each with a unique numeric address starting with zero. The number of bytes that byte-addressable memory may contain depends on the

Figure 2.2

System/36 Real Storage Organization



Performance Tip

The 5360 Model D is generally regarded as having a 7 MB memory limit. However, the *real address space* of the Model D supports an 8th MB. See Chapter 5 for information about adding an 8th MB of memory to your 5360 Model D.

size of the largest allowable address. On the S/36, a memory address is three bytes long, or 24 bits. The first bit of every address is set aside for a special purpose, leaving 23 bits to contain the address. The largest number that can be represented by 23 binary bits is 8,388,607, so the S/36 can theoretically have 8,388,608 bytes in its memory (remember, the first address is 0, not 1). These locations, or bytes, of memory, each with its unique address, make up *real memory* — all the memory that physically exists. The range of addresses, from 0 through 8,388,607, is the *real address space*: the entire set of unique memory locations available to the machine. The term real memory is used to differentiate between memory that is available on the hardware and memory that programs and programmers are *led to believe* is available (more on this type of memory later in the discussion of virtual memory).

Fragmentation

When you try to apply the real memory viewpoint in a multitasking system, problems arise, the worst of which is *fragmentation*. Figure 2.3 shows how this problem develops.

In Figure 2.3a, a hypothetical computer with 128 K of memory is running four programs that consume a total of 124 K. After programs B and D finish running, they release their memory, leaving two 24 K "holes" in the 128 K address space (Figure 2.3b). Later, the computer tries to run program E, which requires 32 K of memory (Figure 2.3c). Although a total of 48 K is available, the program is unable to run because available memory is split into two 24 K pieces. Program E must wait for either program A or C to end before it can obtain enough unbroken, or *contiguous*, memory. Because the usable address space is fragmented, program execution is delayed and perfectly good memory is wasted. Memory fragmentation worsens quickly in a busy computer system, causing system performance to drop off dramatically.

Solving Fragmentation

One solution to fragmentation is allowing programs to run in *noncontiguous* blocks of memory. The S/36 accomplishes this using addresses that do not directly correspond to real memory addresses but must be translated by special hardware into real addresses, hence the term *translated addressing*. Translated addresses appear to the executing program to represent contiguous memory locations.

Here's how address translation works. The S/36 groups bytes of real memory into 2 K units called *pages* (as you saw in Figure 2.2), each with a unique number from 0 to 4095. Figure 2.4 shows the program PROG A broken up into pages. Note that in main memory PROG A is not just fragmented — its logical pages are out of order. The fourth logical page of PROGA physically appears before the first logical page. Before such a fragmented program

Figure 2.3
How Memory Fragmentation Occurs

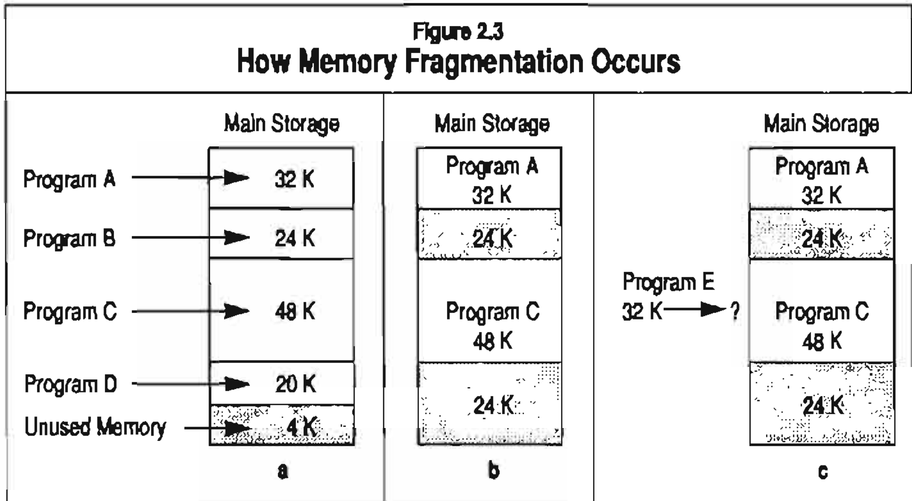
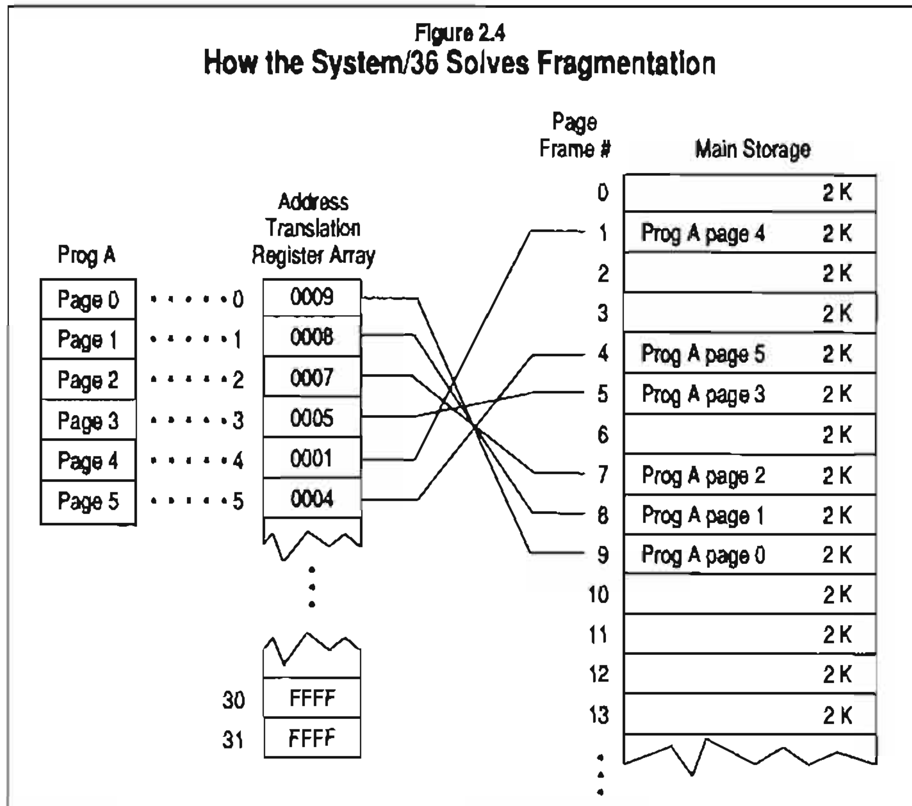


Figure 2.4
How the System/36 Solves Fragmentation



can run, a mechanism must rearrange the physical pages into their correct logical order.

The S/36 contains a special array of hardware registers, called *address translation registers* (ATRs). While a program is executing, each ATR contains the number of a real memory page occupied by the program. The program "sees" these pages as contiguous because the program only uses translated addresses to refer to the pages. Every time the program references a memory location, address translation hardware uses the array of ATRs to generate the correct real memory address for that location. A close look at the translation process reveals some important details about S/36 memory management.

Figure 2.5 shows what happens during real address generation. The translated address is three bytes long, but only the last two bytes contain address information. The first byte is always set to "10000000," where the high-order "1" identifies this address to the MSP hardware as one requiring translation. The seven bits following the "1" are ignored in a translated address. The sixteen remaining address bits provide an address space of 65,536 bytes, or 64 K. (This limit of using only two bytes for address information is the origin of the infamous 64 K region-size limitation.)

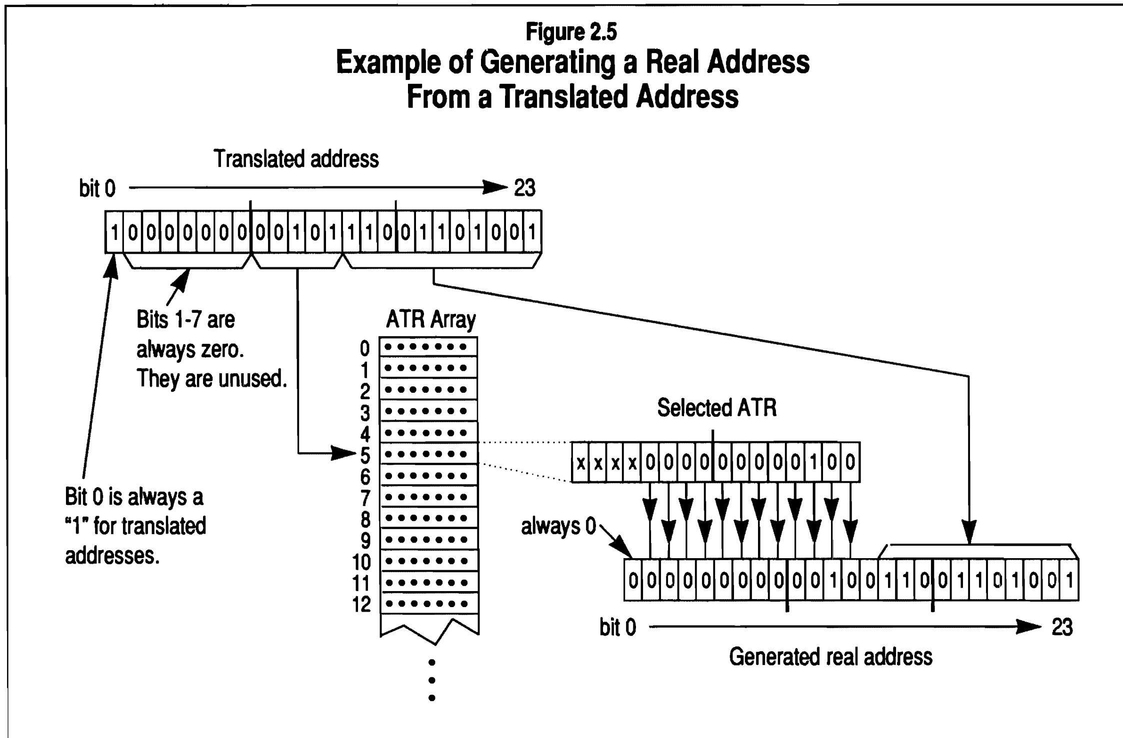
Because the last eleven bits of the translated address always fall within the boundaries of one logical page (the largest number represented by 11 bits is 2,047), these eleven bits are copied directly into the corresponding eleven bits of the generated real address.

The first five bits of the sixteen-bit translated address represent the number of the ATR containing the real memory page frame address. In the example, these bits contain "00101," or five, causing ATR #5 to be selected. Each ATR is sixteen bits long, but only twelve of those bits are used. Those twelve bits are copied to bits 1 through 12 of the generated real address. Bit 0 of the generated real address is forced to a value of zero, which as previously mentioned, designates a real address. This "generated" real address is the actual location of the data in real memory.

The S/36 performs the address translation process automatically for every machine instruction. When a machine instruction references several translated addresses, each address is individually translated as it is needed. For example, if the instruction resides in translated memory (as is usually the case), the instruction address is translated just before the instruction is fetched. If the instruction then references operands in translated memory, each operand address is translated individually before the operand is used by the instruction. Because the translation is carried out in hardware, the process does not add significant time to program execution.

The S/36 address translation mechanism not only solves the memory fragmentation problem, it also lets program pages reside in memory in any order. In fact, the system can even change the page order in memory, provided

Figure 2.5
Example of Generating a Real Address From a Translated Address



the ATR array for the moved pages is updated to reflect their new location. The useful ability to change page order without affecting the programs involved makes possible the next feature of S/36 memory management: virtual memory.

Virtual Memory

The fact that two levels of storage — primary and secondary — exist in most computer systems points up an ongoing compromise in computer technology. High-speed primary storage (such as the S/36's solid-state main memory) is too expensive and volatile for permanent data retention, so permanent information is stored on less expensive, but slower, secondary storage (usually disk). Primary storage contains only the data and programs the computer currently needs. However, the computer often works on several programs simultaneously — perhaps more than can fit in main memory at one time. When the number of currently executing programs exceeds the capacity of main memory, main memory is *overcommitted.* One way to handle overcommitment is to hide the true size of main memory from programs, letting them believe that there is much more memory than actually exists. The memory that programs use during execution — but that may not actually be available on

the system — is called *virtual memory* (VM). The range of "imaginary" addresses is the *virtual address space.*

There are two popular ways to implement VM: segmented paging and demand paging. Figure 2.6 compares some features of segmented paging used by the S/36 with demand paging used by the S/38 and the AS/400. As you can see from the chart, neither technique is new. Both techniques originated in the early sixties and both share three important characteristics: memory organization, backup storage method, and address translation method. Both techniques also make tradeoffs involving expense, performance, and efficiency.

Page-in, Page-out Mechanisms

With demand paging, programs can reference any location in the virtual address space directly, although only some of the pages of the virtual address space actually reside in real memory at any one time. When a program tries to reference a location in a page not currently resident in real memory, special hardware detects the condition and generates a page fault interrupt. The page fault interrupt invokes a special operating system routine or hardware device to locate the requested page on secondary storage and read it into real memory, a process called *paging in.*

As part of the page-in process, the page fault handler updates a table used to generate real addresses during program execution — a process similar to S/36 address translation. To make room for the page to be read in, the page fault handler also may need to select a less important page in real memory and write it to secondary storage, a process called *paging out.* Usually, the paged-out page is chosen using an algorithm that finds the least-recently-used page in real memory.

The term *demand paging* comes from the fact that paging is driven by program references, or demands, to virtual memory. If a program never asks to "see" any location on a particular page, the page is never brought into real memory.

Segmented paging does not allow programs direct addressability to all locations in the virtual address space. Instead, programs have access only to a segment of virtual memory — 64 K in the case of the S/36. Instead of waiting for a program to reference a location in a nonresident page, the operating system keeps a list of pages currently being used by each executing program. When control switches from one program to another, the operating system compares the list of pages the next program requires with a list of pages currently in real memory (the virtual page table). If any pages are missing, the operating system retrieves them from secondary storage. If there are no free pages in real memory, the operating system writes some least-recently-used pages to secondary memory to free up enough pages for the next program to run.

The primary advantage of segmented paging — inexpensive imple-

Figure 2.6

Segmented and Demand Paging Comparison

	Segmented paging (S/36)	Demand paging (S/38 and AS/400)
First implemented	Burroughs B5000, 1961	Atlas, 1962
Memory organization	Fixed-length pages (2,048 bytes on the S/36)	Fixed-length pages (512 bytes on the S/38; 4096 bytes on the AS/400)
Backing store	Secondary disk storage	Secondary disk storage
Address translation	Dynamically with dedicated hardware.	Dynamically with dedicated hardware.
Page-in mechanism	Operating system knows program requirements and brings in required pages before giving program control.	Hardware detects program request for nonexistent page and generates a "page fault" to bring page in before task resumes execution.
Page-out mechanism	Pages for the lowest priority tasks are written out until enough pages are available for the program waiting for storage.	The least-recently-used page is written out and used to satisfy the page fault request.
Real memory usage	All pages for which a program has addressability must be in real storage before the program can run, regardless whether the program actually needs data in those pages now.	Only pages actually referenced by a program need be kept in real storage. Unused pages eventually are moved to secondary storage, freeing real storage for other programs.
Implementation	Mostly software. Address translation is assisted by special hardware.	Mostly hardware. Page faults and content management have special hardware assistance. Address translation is performed entirely in hardware.
Best features	Simplicity; lack of specialized hardware makes implementation less expensive; performance does not depend upon program behavior.	Hardware implementation improves both time and space efficiency; because only referenced pages are resident, memory utilization is good.
Worst features	Lack of hardware assistance means greater execution overhead; large programs tend to squander memory because unneeded pages are kept resident.	Hardware implementation is expensive; certain kinds of program behavior can cause repeated paging, known as "thrashing," which degrades performance.

mentation — comes from the fact that less complex address translation hardware is required. On the S/36, the address translation mechanism already is in place, making it easy to move pages in and out of real memory and rearrange them when necessary.

However, the inexpensive implementation exacts a price in performance. *All* the pages used by a program must be brought in before the program can resume execution, so some pages probably are not needed, and are wasted. Also, the special hardware used by demand paging to detect missing

pages usually is much faster than the software-implemented virtual page table the S/36 uses for segmented paging.

On the S/36, this performance loss is mitigated to some extent, because the CSP can perform VM management chores while the MSP is working on user programs. But segmented paging also imposes a restriction on programmers: programs cannot exceed the size of one segment. On the S/36, the hardware-limited, 64 K segment size is uncomfortably small. Some systems other than the S/36 use a segmented paging approach that allows a program to use more than one segment, thus alleviating the S/36 restriction.

VM does, however, achieve its purpose. It theoretically can manage a virtual address space of 128 MB — 16 times larger than the maximum real address space of 8 MB. And it can manage this large virtual space efficiently. Many S/36 installations use external program calls to activate *all* of their frequently run programs for each user at the beginning of the day — hundreds of simultaneously active program segments amounting to 20 MB or more of VM. Because paging is much faster than reinitiating programs and reopening files, this technique eliminates redundant program initiation, reduces file open and close overhead, and improves response time dramatically.

Peculiarities of S/36 VM

The S/36 VM mechanism has a few unusual, and potentially confusing, twists. One common misconception is that the 64 K segment-size limitation, which also limits program size, limits *task* size. A task can contain one or more programs, each of which can be up to 64 K and must be executed individually. Because the number of programs that can be contained in a task on the S/36 is unlimited, the size of a task is also unlimited (up to the size of virtual address space).

Performance Tip

Using external program calls (EPC) greatly reduces task initiation and file open overhead, improving interactive response time. Changing your existing applications to use EPC rather than OCL to interconnect programs is not very difficult, making EPCs one of the best tools in your kit for improving performance (see Section III).

The S/36 contains a built-in external program call mechanism that lets one program invoke another separately compiled program, and then regain control when the called program returns. In addition, any number of called programs contained within a task may be simultaneously active. Active programs retain their internal state (values of variables and open files) from invocation to invocation.

Another oddity of the S/36 virtual implementation is the concept of *workspaces*, virtual segments that contain data instead of program code. Workspaces hold data buffers, screen formats, and various system-related tables and work areas, helping you get around the limitations of 64 K per program. An example of a workspace familiar to RPG programmers is the *disk file workspace*, which is created automatically when the 64 K segment for an RPG program has no room for disk file physical I/O buffers.

When a program needs to access data in a workspace, it calls on the operating system *map* facility, which gives the program addressability to the

workspace by giving up some addressability to the program's virtual segment. Mapping, however, takes time and may result in paging activity, so the increased flexibility gained using workspaces is purchased with reduced performance.

A third unusual S/36 VM artifact is encountered only by installations that use a large amount of VM. On the S/36 the secondary storage used for paging is called the Task Work Area (TWA). The TWA is contained in a special system file called #SYSTASK that must reside on drive A1.

Initially, the maximum size for #SYSTASK is 6553 blocks (16 MB). This maximum is only about twice the maximum real memory size of 8 MB — not a very efficient overcommitment ratio. When the TWA is full, the SSP automatically extends the TWA by 400 blocks. When the TWA fills again, SSP doubles the extension to 800 blocks. Each time the TWA fills up, the size of the extension is doubled, allowing the TWA to grow to a very large size.

Unfortunately, each TWA extension requires contiguous space on drive A1. Drive A1 is also the default drive the system uses when allocating new files and work areas, which results in disk space fragmentation that may prevent the TWA from extending. Thus, the difficulty of obtaining disk space for paging can result in a much lower virtual address space limit than the 128 MB architectural maximum, unless the user takes steps to force TWA expansion before the A1 disk space becomes fragmented.

Performance Tip

One way to reduce TWA expansion problems is to "pre-allocate" the TWA by activating all your programs in advance — usually immediately after IPL (see Chapter 7).

System Queue Space

Now that you understand real, translated, and virtual memory, you can appreciate the effort undertaken by the S/36 to administer memory usage efficiently. Although address translation and segmented paging improve memory use by effectively reusing a limited resource, not everything in real memory can be moved about with abandon. Only objects in the user area accommodate this manipulation. A certain amount of real memory — the fixed and variable nuclei — must remain resident and can be accessed only through real addressing.

All of the fixed nucleus and most of the variable nucleus is static (unmoving) — beyond your control. As mentioned earlier, programming techniques directly affect only one part of the variable nucleus: system queue space. Knowing how your application design decisions impact SQS use helps you make educated compromises between performance and simplicity.

SQS is an expandable "pool" of memory used by the SSP and the CSP to hold dynamically allocated data structures, called *control blocks*, critical to the operation of the system. Once a control block is created in SQS, it remains resident in real memory at the same location until explicitly destroyed. Because each control block must occupy contiguous memory locations, SQS can become fragmented.

Control blocks range in size from 16 bytes to 2,048 bytes, in 16-byte increments. They can be categorized by their life spans: short, medium, and long. A short-lived control block's life span is only a few milliseconds. The SSP creates short-lived control blocks for the duration of certain brief chores (e.g., a disk file operation) and destroys them when the chore is complete. Medium-lived control blocks last a relatively long time — for the duration of a job, for instance. Long-lived control blocks (usually created when the system is started) are the very few that become permanent until the next IPL.

The system keeps a modest reserve of SQS available (about 2000 to 4000 bytes) to satisfy most control block creation requests quickly. When this reserve is consumed, the system takes a 2 K page away from the user area and adds it to SQS. (Because a control block cannot be larger than 2 K bytes, the newly acquired page can be obtained from anywhere in real memory.) The system continues to take 2 K pages from the user area as needed. When more than about 4000 bytes accumulates in the SQS reserve area (due to control blocks being freed), the system returns a 2 K page to the user area. Thus, the logical "boundary" between SQS and the user area fluctuates constantly to meet the needs of the system.

Of the three classes of control blocks, only one is of concern to you. Short-lived control blocks have minimal impact on system performance, and long-lived control blocks are beyond your control. Only medium-lived control blocks have a controllable impact on system performance; most medium-lived control blocks are a direct result of the kinds of programs you design. The table in Figure 2.7 summarizes the space requirements for the most common control blocks and the program activities that create them.

The table also will help you determine the amount of SQS a given program or device needs to run. Computing the SQS requirements for an entire job mix lets you estimate the total amount of real memory that will be dedicated to SQS, and therefore will be unavailable in the user area. For example, an interactive job with ten indexed files, a printed report, and five subprograms requires 9,088 bytes of SQS:

- 192 bytes for the workstation session control block
- 256 bytes for the job control block
- 96 bytes for the task control block
- 320 bytes for the active programs (64 bytes each)
- 96 bytes for one level of subprogram invocation
- 64 bytes for a disk file workspace
- 688 bytes for the opened print file
- 1,600 bytes for disk file VTOC entries (160 bytes each)
- 1,680 bytes for other file-related control block (file specification block, file buffer block, disk buffer block, allocation queue element, record queue

Figure 2.7
System Queue Space Requirements for Common Control Blocks

SSP entity	Control Block	Total SQS Bytes Used
Each local workstation session	Terminal Unit Block (192 bytes)	192
Each printer	Printer Unit Block (96 bytes)	96
Each remote device	RWS Device Unit Block (80 bytes)	80
Each job	Job Control Block (256 bytes)	256
Each task	Task Block (96 bytes)	96
Each active program or subprogram	Program Block (64 bytes)	64
Each invoked program or subprogram	Request Block (64-2048 bytes)	96 (avg)
Each workspace	Storage Block (64 bytes)	64
Each user of an opened file	File Specification Block (64 bytes) File Buffer Block (24 bytes) Disk Buffer Block (16 bytes)	104
Additional overhead for first user to open a file or use a library	Format-1, or VTOC entry (160 bytes)	160
Additional overhead for each user of a shared file	Allocation Queue Element (32 bytes) Record Queue Block (16 bytes)	48
Additional overhead for each user of an indexed file	Index Control Block (16 bytes)	16
Each storage indexed file storage index	Depends on the size of a storage index (the storage for a file is shared by all users of the file)	varies
Each opened print file being spooled	Printer Specification Block (64 bytes) Spool File Descriptor (112 bytes) Spool intercept buffer (256-2048 bytes)	688 (avg)
Each active spool writer	Writer Descriptor Block (48 bytes) Task Block (96 bytes) Spool print buffer (256-2048 bytes)	1168 (avg)

block, and index control block) , and
- 4,096 bytes for storage indexes (estimated)

If you plan to run the program from nine workstations simultaneously, the additional eight workstations require 3,392 bytes of SQS each (the VTOC control blocks and storage indexes are counted only for the first user), resulting in a grand total of 36,224 bytes of SQS. Remember that SQS use reduces the amount of memory available in the user area for virtual use, thereby increasing the "swap rate" (level of paging activity), and possibly degrading

system performance. If you run these programs on a 512 K system, you might find installing another 256 K memory board a cost-effective way of maintaining acceptable response time.

Considering all aspects of S/36 memory management, you can see why misconceptions abound. But the S/36 loses its mystique once you master the secrets of its memory. You can use this knowledge to help plan future expansion of your S/36 and to evaluate its place in the midrange system market. Careful evaluation of memory requirements lets you predict the effect of additional memory more accurately. And, of course, the better you understand your S/36, the better you can take advantage of its features to improve performance.

Chapter 3

Inside Disk Data Management

Disk I/O is the most common performance bottleneck on the S/36. Unlike the other major components of the S/36, disk I/O is mechanical (it requires moving parts), making it the slowest thing on the system. So anytime you reduce disk accesses, you increase performance.

Traditionally, disk I/O has been interpreted as reading and writing application files. In fact, disk tuning the S/36 requires knowledge at both the application and architectural levels. We will take an in-depth look at techniques you can use at the application level to improve performance through better disk I/O management in Section IV. In this chapter, we focus on the architectural level. You can dramatically improve your applications' performance by learning just a little about the S/36's disk data management (DDM) *architecture* and the silent performance killers that lurk within. An understanding of S/36 disk data management also will make many of the concepts to follow later in this book more clear.

At the architectural level, your applications are often doing things "behind the scenes" that cause extra disk I/O. In fact, many of these events affect performance just as significantly as application data file operations do; but they occur without warning or explicit symptom — other than a sluggish system — that something is amiss. With an understanding of S/36 disk data management architecture, you can avoid these additional low-level disk accesses — many times with very few coding changes.

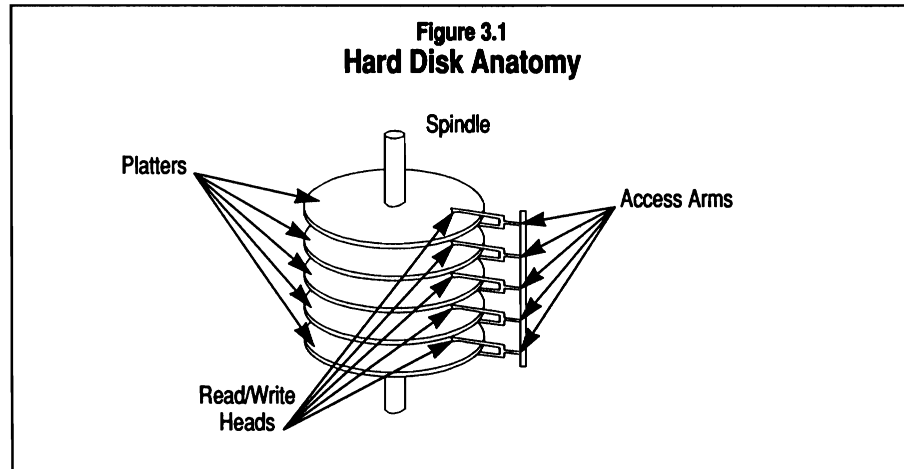
Many performance-enhancing opportunities exist at the architectural level when you understand data files (organization, operation, and index areas) and the Indexed Sequential Access Method — ISAM — (including storage indexes, ripple-down add, alternate indices, DUPKEY processing, and keysorting). But first, let's review the basic concepts related to disk mechanics.

Disk Mechanics

Understanding disk mechanics gives you a better appreciation for the importance of minimizing disk I/O. An overview of S/36 disk hardware anatomy shows how disk access time — and therefore performance — is related to the physical movement required to locate and transfer requested data from the disk.

A disk drive is generally not one disk at all, but rather several disks, or platters, stacked on a spindle. Depending on the drive capacity, a S/36 spindle may have up to eight platters (Figure 3.1). A recording surface is on each side of each platter except for the top one, where only the inward-facing

Figure 3.1
Hard Disk Anatomy



surface is used for data storage. The top surface holds a special set of "servo" tracks that provide control signals used to position the read/write head for each data-recordable surface (Figure 3.2). These heads transfer data to and from the hard disk. On a drive with eight platters, there are 15 read/write heads — two heads each for the lower seven platters and one servo head for the top surface. Except for the diskette drive, the access arms and disk platters are the only moving parts inside the S/36. The key to getting the most out of your S/36 is to eliminate as much of this physical movement as possible. In the time it takes the S/36 to perform one disk access, it can perform as many as 35,000 machine instructions. Every time you eliminate disk I/O, you increase performance.

Technical Note

In the time it takes the S/36 to perform one disk access, it can perform as many as 35,000 machine instructions! *Every* disk I/O you eliminate improves performance.

Each platter's surface comprises concentric circles called tracks, and each track is divided into segments called sectors (Figure 3.3). The number of sectors in a track — and the number of tracks on a surface — depends on the disk drive model. On the S/36, there are 256 bytes in each sector, which is the smallest amount of disk storage that can be read or written in a single disk operation. The system addresses the disk by relative sector number, where the first sector number is numbered 0, the next 1, and so on through all the tracks on all drives. Any given vertical stack of like-numbered tracks is called a cylinder. Data

Figure 3.2
Hard Disk Physical Units

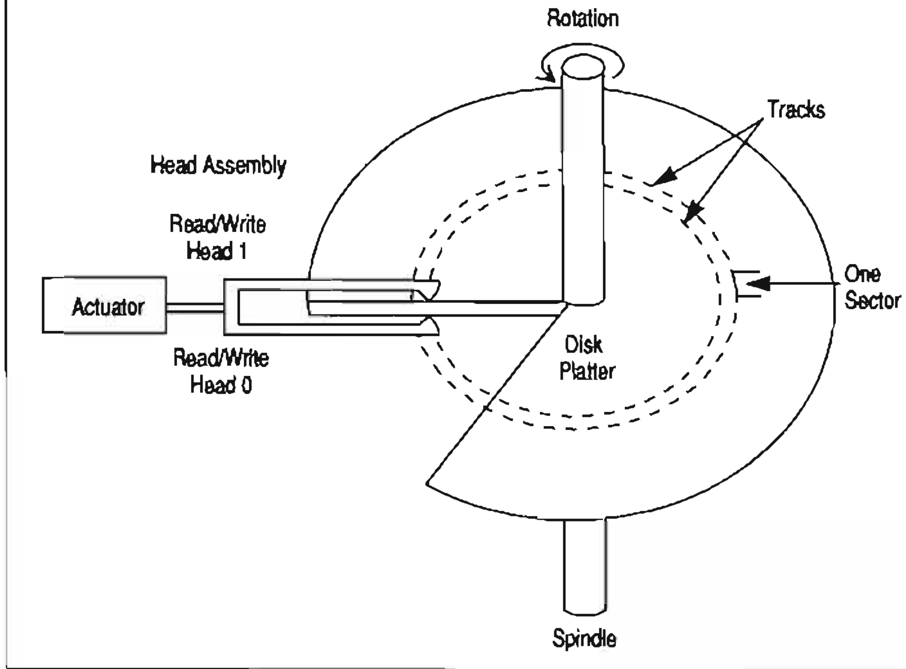


Figure 3.3
Sectors and Tracks

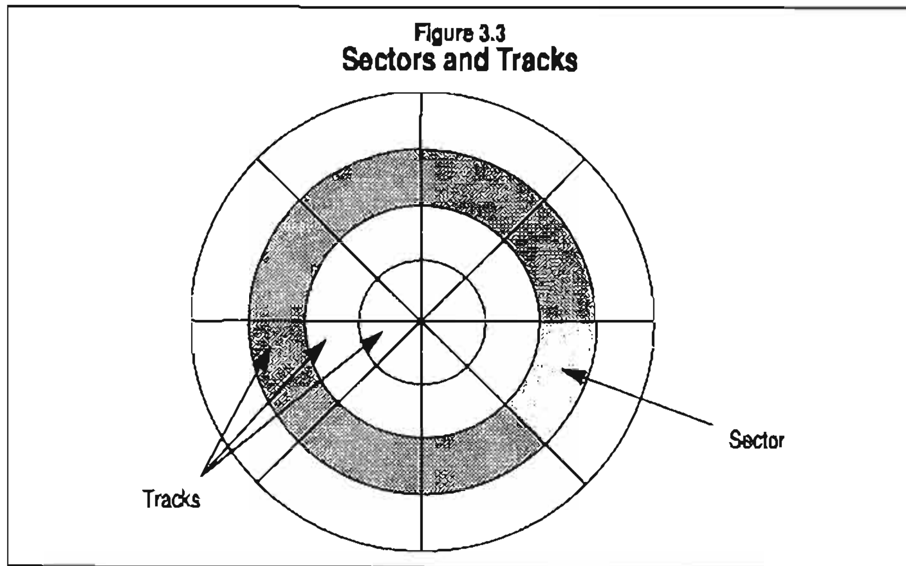
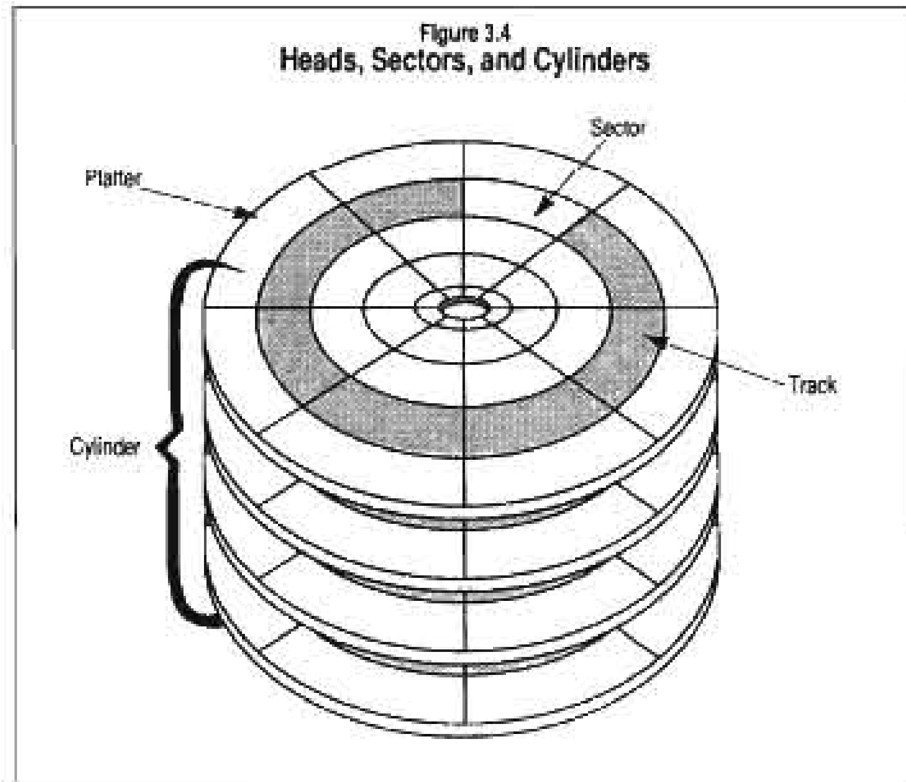


Figure 3.4
Heads, Sectors, and Cylinders



is stored vertically in these cylinders (Figure 3.4), so that like numbered tracks within a given cylinder contain sequentially stored data. Storing data in cylinders maximizes the amount of sequential data that can be read without moving the disk arm.

Technical Note

Each drive is generally made up of several platters. Platter surfaces are divided into concentric circles called tracks. Each track comprises many 256-byte size chunks called sectors. A sector is the smallest amount of disk storage the S/36 can read or write in a single disk operation.

As an application programmer, you're familiar with high-level language (HLL) file operations such as *read sequential*, *read-random-by-key*, *add*, and *update*. Collectively, these application-level functions are called

logical operations, because S/36 disk hardware can't directly execute them. When an application program requests a logical file operation, DDM translates it into one or more *physical* operations that are then directly executed by the hardware. The S/36 supports three such operations: *read*, *write* and *scan*.

The read operation transfers one or more sectors from a given disk address into a main memory buffer. Similarly, the write operation transfers one or more sectors from a memory buffer to a specified disk address. The scan operation, which is somewhat unusual in the world of commercial computer systems, searches one or more sectors for a particular data pattern. By combining these three physical disk operations in various ways, DDM can carry out any HLL logical-file operation. Knowing how much time each physical operation requires lets you estimate the time required for various logical operations. Knowing how DDM maps logical-file operations to physical operations lets you predict situations where DDM performs poorly.

The mechanics of disk operation dictate three time factors to every physical operation. First, the time required to move the access arm to the cylinder containing the data. Second, the delay while the rotating disk brings the requested sector under the read/write head. Third, the rate at which data transfers between the disk and memory. These are called, respectively, seek time, rotational delay, and data-transfer rate.

Seek time is the largest of these factors; it is proportional to the distance the access arm must move. However, a large part of the seek time is spent just starting and stopping arm motion, so seek *distance* isn't as significant as you might think. S/36 drives have average seek times ranging from 12 to 40 milliseconds (a millisecond (MS) is one-thousandth of a second). Average seek time is based on spanning one-third of the disk — some 400 cylinders. But moving the access arm just one cylinder takes as much as 10 milliseconds, making seek distance a minor factor. Clearly, *eliminating* seeks — not shortening seek distance — is the key to controlling seek time!

The second largest time factor is rotational delay, which averages between 7 and 10 milliseconds for S/36 drives. Sector starting points are offset between adjacent cylinders, so that reading sequentially across cylinders doesn't require wasting one rotation because the next sector went by during the seek operation. Average rotational delay is the time required for one half of a disk revolution.

The smallest time factor is the data-transfer rate, which runs between 0.8 and 5.7 megabytes per second, depending on the drive model. Even on slower drives, however, transferring a single sector takes only 0.2 milliseconds — a tiny fraction of the total disk operation time. You could transfer 10 times that amount and still only spend 2 milliseconds. The key point here is that the quantity of data transferred in a physical operation has minimal effect on total operation time.

Figure 3.5

Comparison of Operating Parameters for Some S/36 Disk Drives

Drive Model	Capacity	Sectors /Track	Tracks /Cyl	# of Cyls	Average Seek Time	Rotational Delay	Transfer Rate
10SR	200 MB	100	14	572	25 ms	10.1 ms	1.2 MB/s
10SR	359 MB	100	14	1024	25 ms	10.1 ms	1.2 MB/s
21ED	30 MB	70	4	445	40 ms	9.5 ms	0.9 MB/s
21ED	60 MB	70	4	888	35 ms	9.5 ms	0.9 MB/s
0065	40 MB	32	7	733	40 ms	8.3 ms	0.6 MB/s
9332	200 MB	148	4	1349	19 ms	9.6 ms	5.7 MB/s
9402	160 MB	48	14	946	12 ms	6.9 ms	0.8 MB/s

Performance Tip

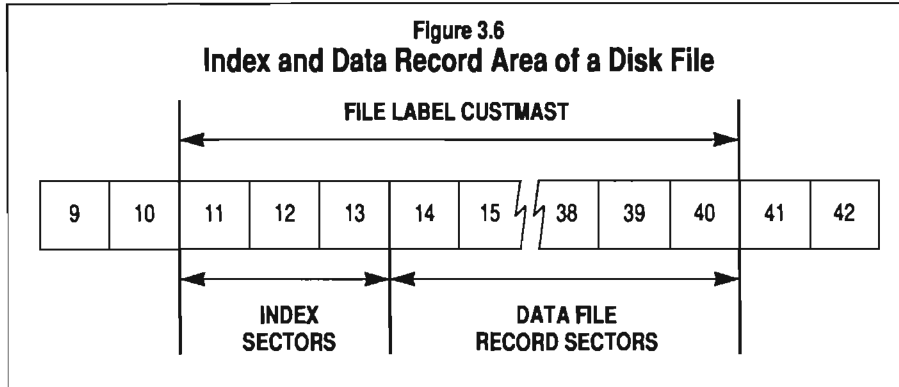
All of the disk drives available for the S/36 offer a range of data-access times (18 to 50 milliseconds). However, transfer rates vary dramatically from model to model. The 21ED and the 10SR drives transfer data at 1.25 and 1.5 MB per second, respectively; the 9332 disk drive transfers data at 5.7 MB per second — a four-fold improvement. For applications that copy lots of files, the 9332 will be noticeably faster.

These timing factors reveal where most I/O time is spent: moving the disk arm. Knowing that the amount of data transferred won't significantly affect I/O timing also gives you insight into data blocking performance: Sensible blocking factors won't penalize you much in increased I/O time, even if the I/O requests don't often fall within a block. Knowing the actual timing values for different disk devices also helps you estimate data throughput times for various S/36 configurations. Figure 3.5 summarizes the timing factors for various disk drive models. (Refer to Section IV for an in-depth discussion of blocking.)

You might think that DDM translates such HLL file operations as READ and UPDATE directly into physical read and write operations. However, as you've seen, the time taken by physical operations depends on locating the data to be read or written and not on the amount of data transferred. Reading or writing 10 or 20 sectors takes essentially the same amount of time as one sector. So DDM uses memory buffers that can hold more data than a single logical operation usually requires, eliminating the need for some physical operations and improving performance.

HLL operations such as the RPG CHAIN for an indexed file actually perform two functions and thus require more than one physical operation. First, the record to be read must be located by searching an index; second, the data in the record must be read into memory. The physical scan operation helps greatly here. Rather than searching an index by reading chunks of it into a buffer and then searching for the key in the buffer, the scan operation searches the entire index for the requested key "on the fly," stopping only upon finding the target key or reaching the end of the index. The scan operation can read and compare fast enough to keep up with the disk's rotation speed, taking much less time than the read-into-buffer approach. The S/36 also uses the scan operation for other search functions, such as locating a file

Figure 3.6
Index and Data Record Area of a Disk File



in the VTOC, locating a library member in a directory, and locating a screen format in a screen format load member.

Data Files

With an understanding of physical disk organization and operation under your belt, you're now ready to examine the logical organization and operation of data files. You'll see that DDM leaves you many opportunities to capitalize on your knowledge of disk operation to improve system performance.

Organization and Operation. The S/36 stores data files on disk in contiguous sectors, using as many sectors as necessary to accommodate the entire file, even if the file is empty or only partially full. Sequential and direct files contain only a data area, and alternate index files contain only an index area. Indexed files contain an index area and a data area, with the index area preceding the files' data records. Alternate indexed files only have an index area — which refers to the data area of the parent file. Physically, data files are stored and accessed vertically in like-numbered tracks on a drive's many platters — via the drive's cylinders. Logically, though, a data file is just a contiguous group of sectors, without regard for track, cylinder, platter, or even drive boundaries. Figure 3.6 shows the logical layout for an indexed file named CUSTMAST. CUSTMAST has three index sectors (sectors 11, 12 and 13) and 27 data sectors (sectors 14 through 40). Sector 41 is not part of CUSTMAST and is available for the system to allocate for other disk storage.

Some operating systems support noncontiguous file storage. MS-DOS and Unix, for example, dynamically allocate disk space as needed for a data file. As records are added to a file, the disk space needed to accommodate new records is automatically allocated and maintained. The S/36, however, requires that data files be on disk in pre-allocated contiguous sectors. Therefore, the total number of records that a S/36 data file can contain is always fixed — it must be specified when the file is created. Even when using the EXTEND keyword with

Figure 3.7
Records Fitting Evenly within a Sector

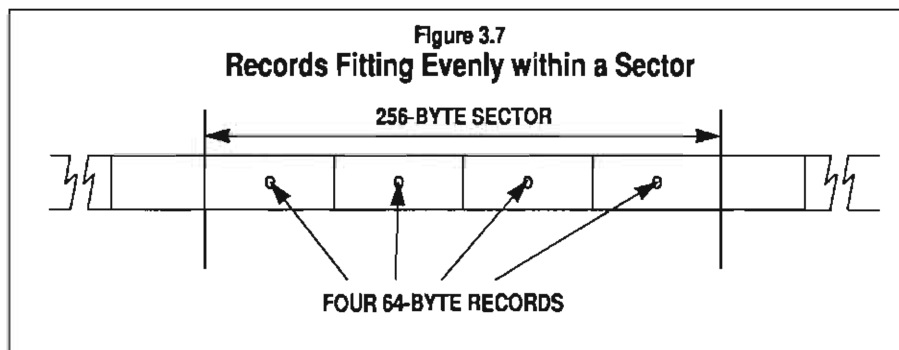
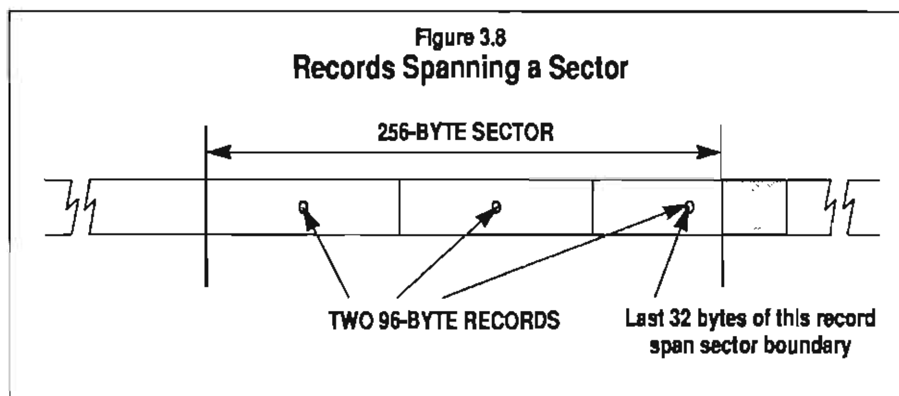


Figure 3.8
Records Spanning a Sector



the // FILE statement, a file is not actually extended when it gets full, but rather is copied to another area on disk with a larger allocation.

SSP stores data records making up a disk file in contiguous sectors on disk. Data records *do* span sector boundaries when necessary. As Figure 3.7 shows, record lengths that are submultiples of 256 fit exactly within a sector. However, when the record length is not a submultiple of 256, data records span sector boundaries. Figure 3.8 shows the sector layout for a file with 96-byte records. Here, two complete records and 64 bytes of another record are stored in one sector. The last 32 bytes of the third record are stored in the next sector. A 512-byte record length requires two contiguous sectors for each record. Much has been written about S/36 record lengths and their impact on performance. Because the S/36 can only read and write data in 256-byte chunks, it is widely believed that S/36 data records should only be multiples or submultiples of 256. As this chapter will later show, because of the S/36's extremely fast data-transfer rate, data record length has little, if any, impact on performance. Let your application dictate record size — not the myth that records should always be submultiples or multiples of 256.

Figure 3.9

Logical Representation of a File Index

Key Value	Relative Record Number
AA0112345345	567
AA0123234835	131
AA0123555235	234
AA0132434063	455
AA0145567720	835
AA0157758385	521
AA0162476410	212
AA0169556693	134
AA0176345952	546

Technical Note

It's widely believed that data-file record length substantially affects performance. Given the fast transfer rate of even the slowest S/36 drives, record length has little bearing on performance. Let your application dictate record length — not the myth that records should always be submultiples or multiples of 256.

Primary and Overflow Index Areas. Indexed files have an index area on disk — an area that physically precedes the file's data. The index contains record keys and corresponding relative record numbers for each data record (Figure 3.9). The relative record number is stored in the index as a 3-byte binary value. This pairing of key values and relative record numbers is used by ISAM to randomly retrieve data records. When an indexed record is requested, ISAM performs a key-value search in a similar fashion to look-up tables in RPG: ISAM scans the index for the key; if the key is found, ISAM uses the key value's corresponding relative record number to retrieve the data record.

Like data file storage, keys are stored in contiguous sectors. Unlike data records, though, keys and their corresponding relative record numbers *do not* span sector boundaries. Figure 3.10 shows one sector of an indexed file with 18-byte keys. Each index entry requires 18 bytes in the index for the key value, plus 3 bytes for the corresponding relative record number, making the total index entry length 21 bytes (the 18-byte key value plus the 3-byte RRN). A 256-byte sector can contain 12 of these 21-byte index entries (12 x 21 = 252). Therefore, for each sector in the index, four bytes per sector are wasted. Disk real estate is generally not at such a premium that you need to worry if

Figure 3.10
Logical View of One Sector Full of Keys

Key	RRN	Key	RRN	Key	RRN	W
Key	RRN	Key	RRN	Key	RRN	a
Key	RRN	Key	RRN	Key	RRN	s t
Key	RRN	Key	RRN	Key	RRN	e

Twelve 18-byte keys (12 x 21 = 252) 4 bytes in sector wasted.

some (or even all) of your indexed files allow some of this waste. The important thing to understand as you look at Figure 3.10 is that index entries are stored in sectors — pairing key values with corresponding data relative record numbers — and that index entries *do not* span sector boundaries. (For more on indexed files and their relationship to performance, see Chapter 10, "Using DBLOCK and IBLOCK Effectively.")

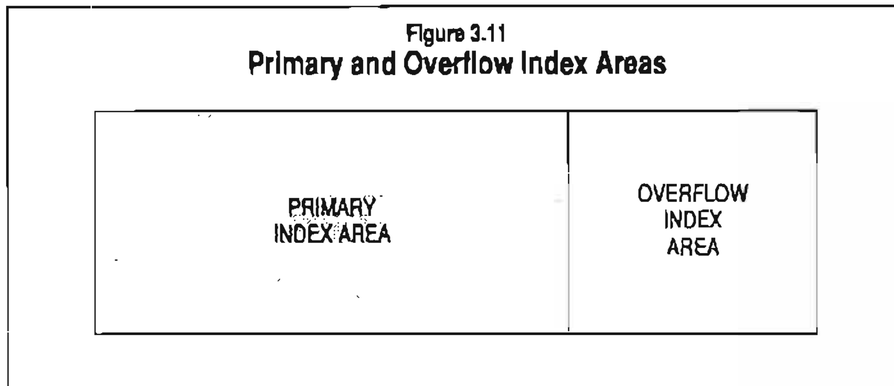
Technical Note

Alternate indices are made up entirely of an index area and do not physically have any data records in them. The relative record numbers stored in an alternate index refer to record locations in the parent file.

Every file index, for both parent and alternate indices, has a primary index area and an overflow index area (Figure 3.11). The primary index area contains index entries for the records in the file in ascending order by key. The overflow area contains index entries for records recently added to the file. Entries stay in the overflow area until the file is *keysorted*. As records are added to a file, new index entries are added to the index overflow area. In most cases, as index entries are added to the overflow area, disk data management automatically keeps them ordered in key sequence.

There are, however, a couple of exceptions to disk data management automatically keeping recently added index entries in key sequence. One example is when a program is creating a new indexed file. Because no other program can read records in a new file until the creating program closes the file, disk data management knows it doesn't need to keep the overflow area in order

Figure 3.11
Primary and Overflow Index Areas



and will simply append the new index entries, in arrival sequence, to the end of the overflow area. DDM sorts the keys into sequence after the program ends.

Consider another case: adding records to an indexed or sequential file that has an alternate index built over it, but the alternate is not currently open as records are added to the parent file. In this case, disk data management creates a 256-byte buffer to temporarily store index-entry additions to the unopened alternate index file. Each time this 256-byte buffer gets full, its contents are appended, in arrival sequence — not in key sequence — to the overflow area of the alternate index. The overflow area must be in ascending key sequence for ISAM to sequentially search the index, so the next program to open this alternate index with its "stale" overflow area must wait while disk data management keysorts the overflow area. This process is called *delayed index maintenance*. Delaying the ordering of keys in the overflow area dramatically improves performance as records are being added to the file. However, this performance increase must be paid for the next time the file is opened, when disk data management transparently keysorts the overflow area without issuing any messages to the operator. If many records were added to the file, the overflow keysort could take a *long* time, delaying program initiation.

The Storage Index

Recall that index entries are stored in sectors and that many sectors are stored in a track (Figure 3.3). When an indexed file is first opened, if its index occupies more than one track, the system creates a *storage index* in the system queue space for that file if a storage index does not already exist. The storage index is an in-memory index to the disk tracks that comprise a disk file's index — a table containing the beginning key value for each track and its corresponding track number. As long as the file remains open by any job in the system, the file's storage index will persist and is available to any application that needs it.

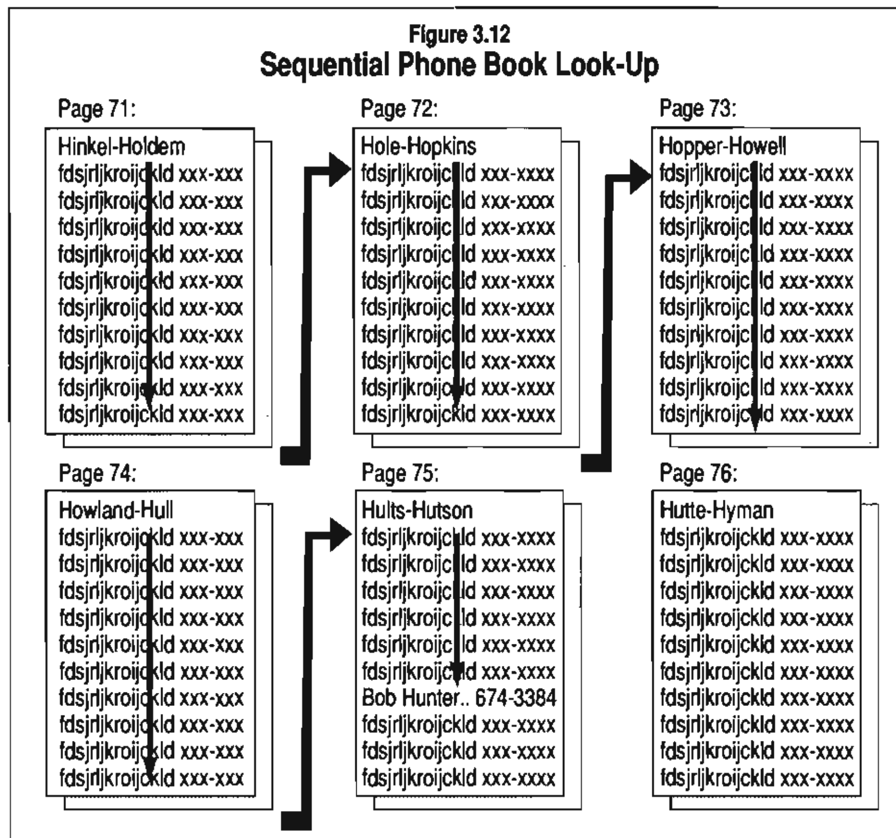


Figure 3.12
Sequential Phone Book Look-Up

The storage index is like the "finder" entries at the top of each page of a telephone book. If you wanted to look up Bob Hunter's phone number in the phone book, you would not sequentially scan every page of the phone book looking for Bob Hunter (Figure 3.12). Rather, you would scan across the top of each page until you found the page with the Hunters on it. After finding this page, you would sequentially scan it looking for Bob Hunter (Figure 3.13). Each page of the telephone book is like a track full of index entries. By using the top of each page as an index to the pages — like a storage index — you would quickly zero in on the page with Bob Hunter on it.

S/36 indexed files use the ISAM method of retrieving random records, and ISAM's storage index works much the same way as the above example. To locate a given key in the index, disk data management first determines, by searching the storage index, which disk track contains the key. Knowing this track number limits the hardware scan for the key value to exactly one disk track (like knowing which page number in the phone book Bob Hunter's

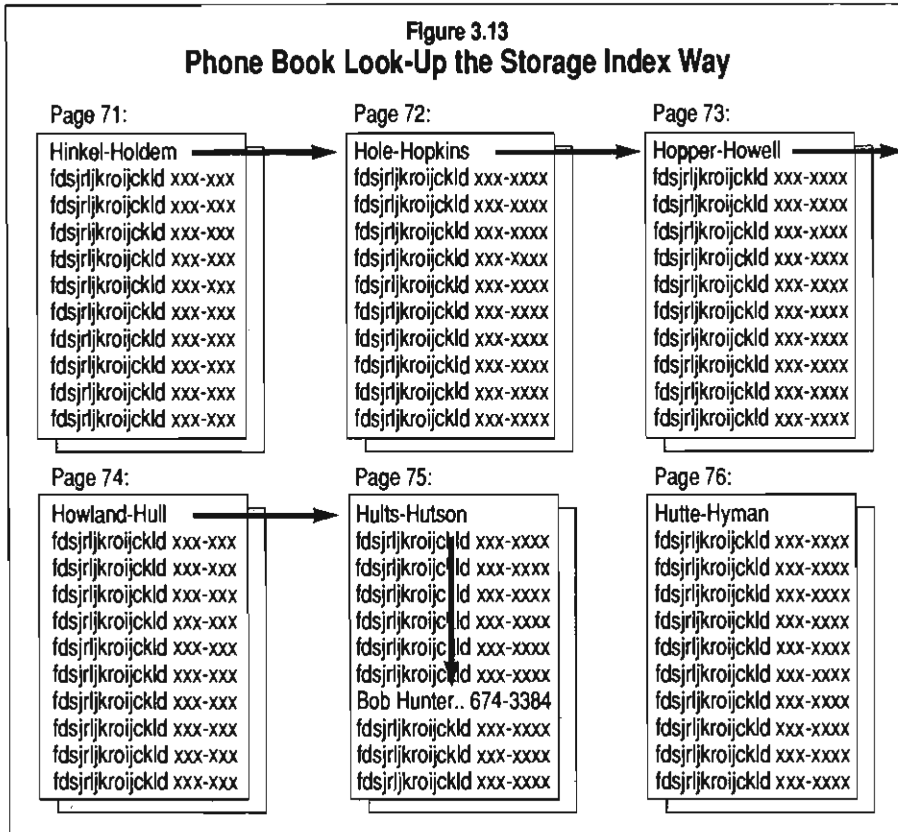


Figure 3.13
Phone Book Look-Up the Storage Index Way

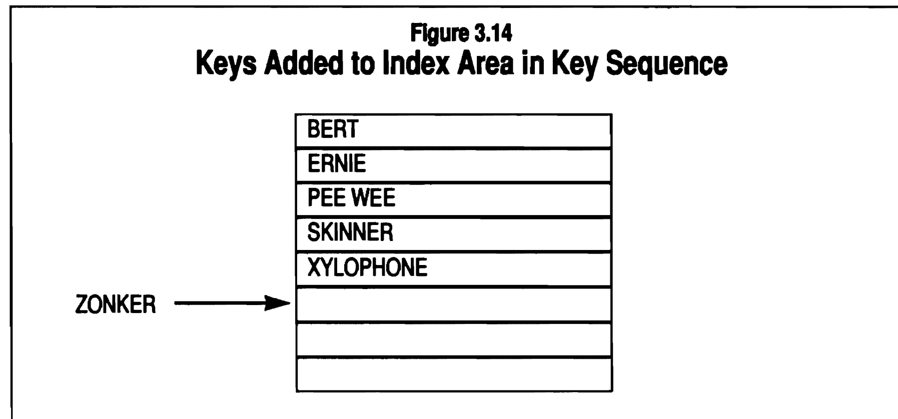
phone number is on). If a storage index doesn't exist for a file, either because the index is too small or because your OCL explicitly inhibited its creation, disk data management must scan every track sequentially looking for the key entry.

Technical Note

The storage index is an index to the index — containing the beginning key value of each track and its corresponding track number. For additional information on the storage index, see Chapter 11.

The overflow area is never included in the storage index. So even if a storage index is created for a file, it always takes longer to fetch a record whose index entry is in the overflow area, because only after searching the

Figure 3.14
Keys Added to Index Area in Key Sequence



prime index does DDM know to look in the overflow area. Thus, an extra scan is required — to search the overflow area. If the overflow area spans several tracks, the scan will take much longer then the prime index scan.

Ripple-Down Adds. S/36 indexed files require ongoing internal maintenance of the index overflow area to provide access to updated and new records. Disk data management must maintain the index for files shared by two or more programs. Note that even during those times when delayed index maintenance is used (as in the case of adding records to a parent with a closed alternate), the overflow area must be sorted before any other job can use the indexed file.

Because of the importance of maintaining the overflow area in key sequence, the S/36 goes through some pretty extensive gymnastics to maintain this order. The worst of these gyrations is the *ripple-down add* technique that DDM uses to keep keys in the overflow in ascending order. To see how these index gymnastics affect performance, let's take a look at how records are added to an indexed file on the S/36 and the heartbreak of ripple-down adds.

Let's start with understanding why the overflow index is necessary. Figure 3.14 shows that for indexed files created using only presorted input (as might have been the case years ago with presorted punched cards), the overflow area isn't necessary. As long as records are being added in key sequence, the keys fall nicely into the index area in ascending order.

However, what happens when the input records aren't presorted? Figure 3.15 shows a situation where the key GROVER must be added to the index, but the previous key added was ZONKER. Rather than try to move all the primary area keys down to make room for GROVER, ISAM simply adds the GROVER key to the end of the overflow area. Because disk data management knows to look in the overflow area when it can't find a key entry in the primary area, this works fine — without much impact on anything as long as

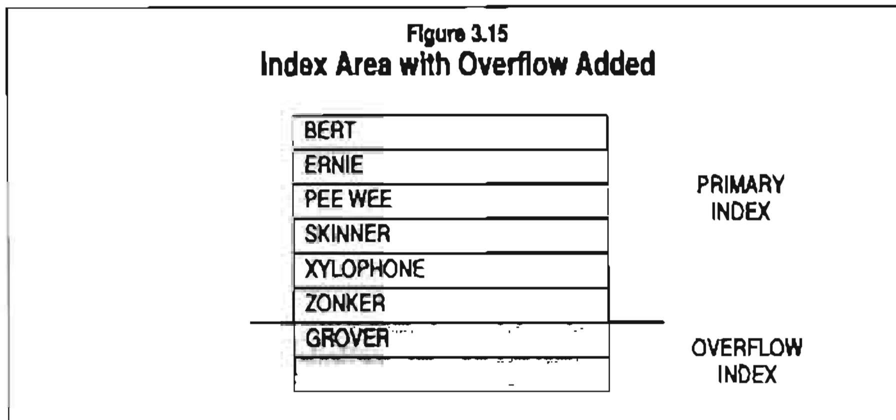


Figure 3.15
Index Area with Overflow Added

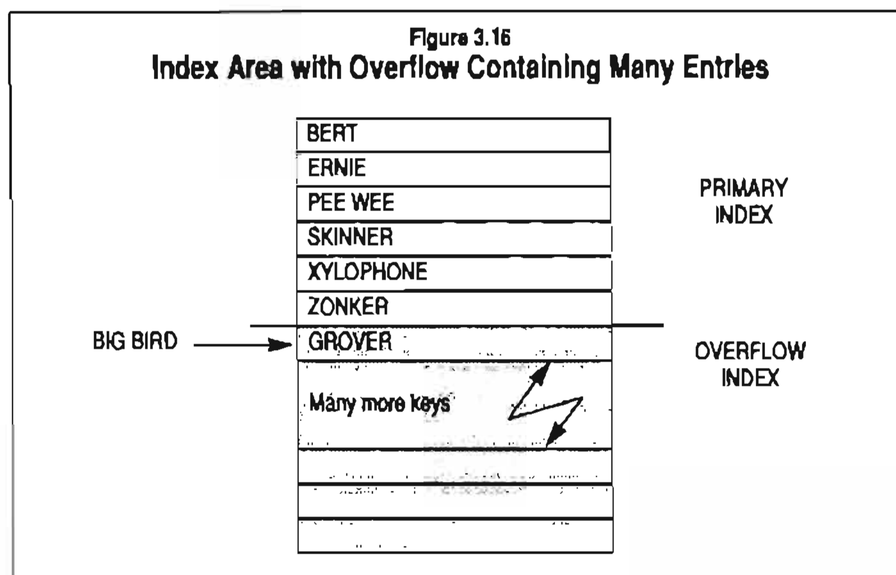
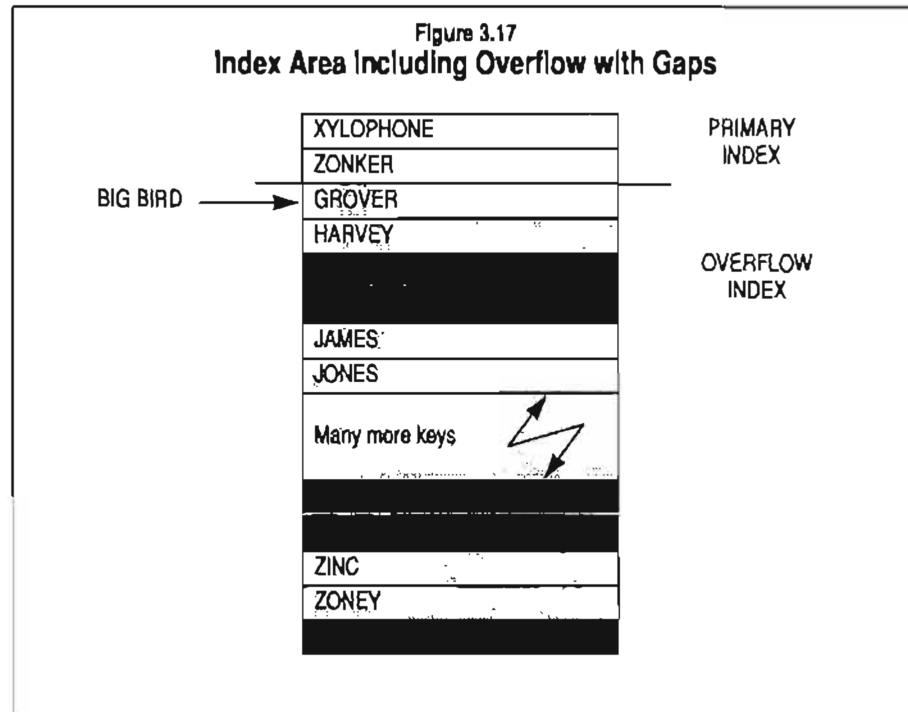


Figure 3.16
Index Area with Overflow Containing Many Entries

the overflow area remains small. It will take disk data management a little longer to realize GROVER'S key isn't in the primary area. Remember: It takes an additional scan and seek to fetch a data record when its key entry is in the overflow area; but what's a few milliseconds among friends?

Figure 3.16 starts to reveal the real problem. Look what happens when you need to add BIG BIRD's key to the index, when the file is opened for keyed input as well as add. Because disk data management requires that the overflow area be kept in key sequence for keyed input access, GROVER must be moved down a notch to make room for BIG BIRD. Moving

Figure 3.17
Index Area Including Overflow with Gaps

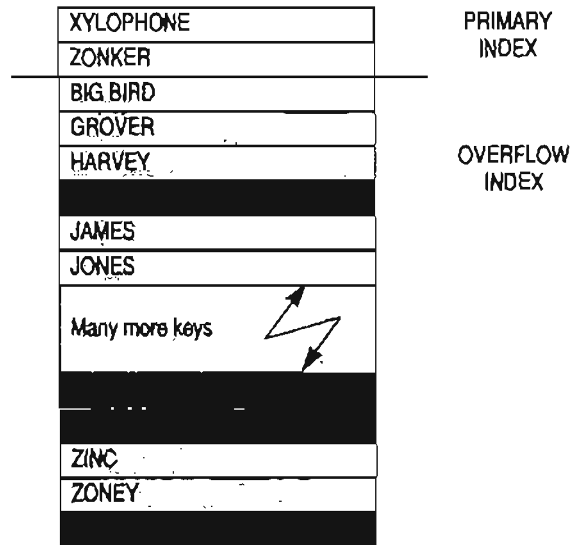


GROVER's index entry down one isn't that big of a deal, because the overflow area is (hopefully) small. However, consider what would happen if there were many keys, hundreds or thousands, to move. When BIG BIRD wants in, the only solution seems to be moving many, perhaps hundreds or thousands, of keys to make room for inserting BIG BIRD's key in the correct sequence.

Adding Gaps. DDM is stuck. It needs to add BIG BIRD to the index but the cost of the numerous disk I/Os to "ripple-down" the high-value keys to make room for BIG BIRD renders the ripple-down add strategy all but worthless. To solve the problem, DDM kludges something together called index *gaps*. As index entries are added to the overflow area, empty sectors (gaps) are left intermittently between index entries. Figures 3.17 and 3.18 show that with gaps, adding BIG BIRD to the overflow area will only cause the ripple down of two index entries. *Usually* there are enough evenly distributed gaps in an index to require moving only a few hundred keys or so.

Adding gaps to the overflow area seems like just the relief ripple-down add needs. That is, until enough keys are added after the last gap that there isn't any room left at the bottom of the overflow area for new keys. In Figure 3.19, ZEVON wants in; but because many gaps have previously been inserted, there isn't a slot available between ZEVON's insertion point and the

Figure 3.18
Index Area Including Overflow with Gaps — Big Bird Added

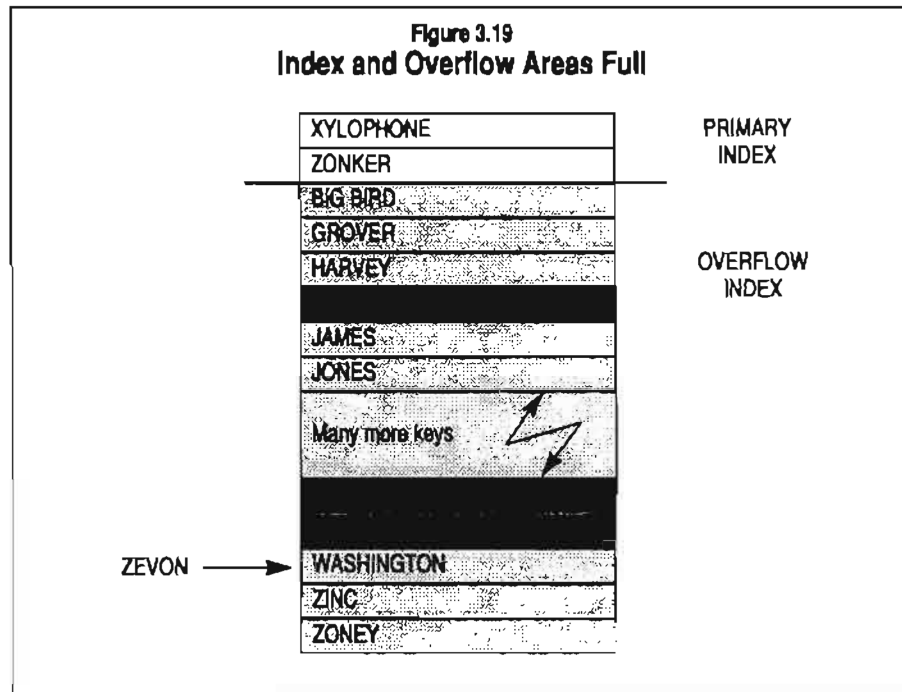


end of the index. Now DDM is in real trouble. However, ISAM was engineered not to fail, and it's not going to. Here's what disk data management does to let ZEVON in:

- Locks the file so that no other tasks using it can run.
- Calls KEYSORT to perform a special "degap" operation — collecting all the available gaps and moving them to the end of the overflow area.
- Issues no other messages or other explicit warnings (disk data management isn't all that proud of this mess). Your S/36 just "nods off" for no apparent reason.

Hopefully, by the time the user calls Level One support wondering why the S/36 went into a coma, the degap operation will have completed and the S/36 will be shaking its groggy head back to life. If Level One's phone is busy and the impatient user re-IPLs the S/36, file rebuild will come along and finish the job.

If you've ever had your S/36 simply take a little nap, especially late in the afternoon, ZEVON wanted in. There isn't anything you can do about a degapping delay once its happened, and you won't be told it's happening.



Your only option is to let your S/36 zen its way through degapping the index — which might take a *long, long* time. Eventually, the index will be degapped and your S/36 will come out of its coma. But here's the really bad news. After degapping the index and letting ZEVON in, we're back where we started from. Every record added for the rest of this job will require ripple-down adds for all keys higher than the key being added. In Figure 3.20, AARDVARK is the next key to be added. With all the gaps now collected at the end of the overflow area, *every* index entry must be rippled down to let AARDVARK in. This ripple-down will continue for every record left to be added. Just as the S/36 was starting to shake off the effects of the degapping processing, the disk I/O required to ripple-down add the remaining index entries puts it right back into its coma.

Here are a few defensive things you can do to avoid index degapping:

- Substantially overallocate large indexed files that sustain frequent index-add operations. For example, consider a 300,000 record file to which 10,000 records are added daily, and which is currently allocated at 400,000 records (enough for adding 10 days' business, after which some records are removed from the file). Even though you don't think there will ever be more than 400,000 records in this file, overallocating this file to 500,000 or even 600,000 records will

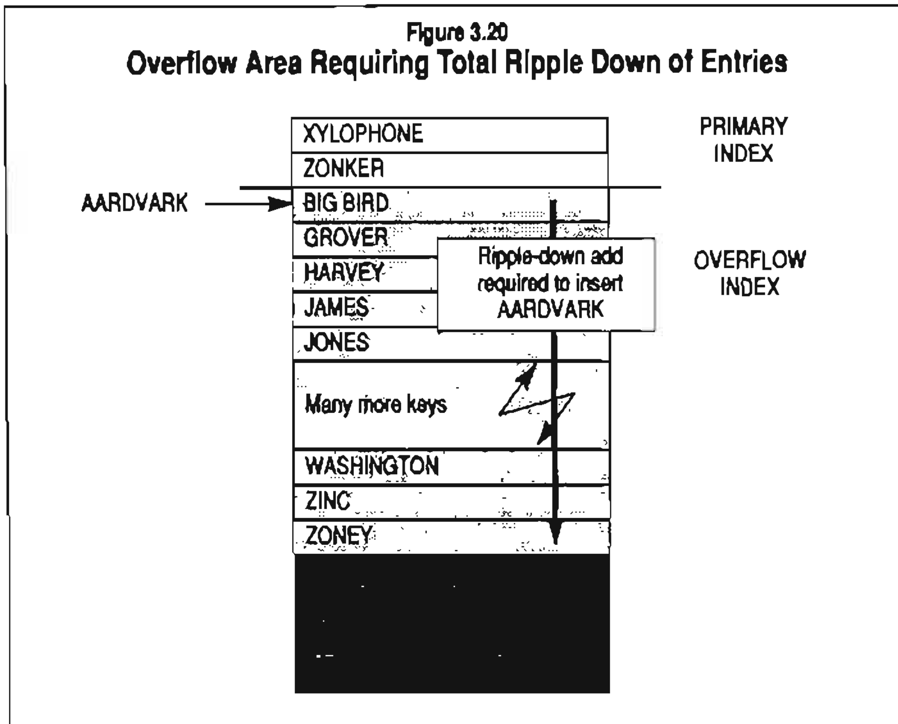


Figure 3.20
Overflow Area Requiring Total Ripple Down of Entries

permit more gaps in the overflow area, reducing the likelihood of degapping ever becoming necessary. Keep in mind that for alternate indices, updating a key field is really a delete followed by an add.

- Presort input files in key order before adding them to an indexed file. This shifts some of the work that disk data management would have had to do to #GSORT — and the results often are astounding.
- If your organization uses very large files and you have reason to believe you're often the victim of index degapping, consider purchasing or leasing ASNA's ACCELER8, a B-tree index implementation that doesn't require key overflow areas and their resulting maintenance.
- KEYSORT your files often. A section later in this chapter discusses KEYSORTing more thoroughly.
- Schedule processes that add many records to indexed files for late at night, when the workload of ripple-down adds is less significant.

Alternate Indices and DUPKEY Processing. Alternate indices use primary and overflow index areas, just as regular indexed files do. Therefore, alternate indices are at risk to the same kinds of ripple-down and index-gapping problems as regular indexed files. Alternate indices with duplicate keys are especially

Performance Tip

Consider substantially over-allocating the size of indexed files to which many records are frequently added.

prone to the ripple-down nightmare. This problem is particularly pervasive because alternate indices very often are likely to have duplicate keys. Consider these possibilities for duplicate keys in an alternate index:

Zip code
Area code
Birthdate
Blanks in the key field (a perennial favorite)

When maintaining alternate indices with duplicate keys, disk data management must insert new duplicate key values in *RRN order*. This point is significant, because ***updating*** keys may result in key insertions in the middle of the duplicate key string. Figure 3.21 shows a fragment of the index overflow area of an alternate index that supports duplicate keys. Duplicate keys are sorted in the overflow area by key value first, then by relative record number. As noted earlier, this does not necessarily mean arrival sequence, because a previous key value could have been changed to 46953, moving its index entry from where it was to its current location with the other 46953 keys.

Disk data management doesn't allow any gaps within a series of duplicate keys. Therefore, adding a new duplicate key or changing an existing key value to that of other existing duplicate keys will quite likely require the rippling-down of many keys in the overflow index. Figure 3.22 shows a new 46953 record being added to the file. To keep the overflow index in order, DDM must insert this index entry between keys 45953 rrn #3 and 46953 rrn #67. This duplicate key is a zip-code field; and with many records in the file with the same zip code, many keys will need to ripple down to make room for the new key. Inserting a duplicate key value in a long string of duplicate keys could take minutes to hours! And because alternate indices can allow key updates, you can encounter this problem simply by changing a key value.

Performance Tip

Avoid duplicate keys in alternate indices. Inserting a duplicate key value in a long series of duplicate keys could take minutes to hours!

To avoid the problem, avoid duplicate keys. That's generally easier said than done, because duplicate keys are so handy. But consider appending a field or another part of a field to the key value to help make it unique. For example, in the case of needing to create an alternate index over a zip-code field, append the telephone number suffix to the zip code and index the file on that value. Leave the file duplicate-key-capable for the rare time that two customers living in the same zip code will have the same telephone suffix. The likelihood of this file now containing duplicate keys is greatly diminished.

Keysorting. Now that you understand how important the index overflow area is to DDM performance, you can see how keeping indices sorted helps reduce ISAM performance bottlenecks. Keysorting keeps the overflow area of the index small because it merges index entries in the overflow area with those in the primary index area. A large, unmerged overflow area degrades random access, future record adds, and alternate-index maintenance.

Figure 3.21
Fragment of Overflow Area that Supports Duplicate Keys

Key	RRN
46952	117
46952	131
46953	3
46953	67
46953	149
46953	177
46953	191
Many other 46953 duplicate key values	

Figure 3.22
Adding New Record to Overflow Area with Duplicate Keys

Key	RRN
46953	51

Key	RRN
38621	117
38621	131
46953	3
46953	67
46953	149
46953	177
46953	191
Many other 46953 duplicate key values	

The SSP invokes KEYSORT automatically during IPL, or when the system operator enters a STOP SYSTEM, or when any of the SSP procedures RESTORE, COPYDATA, TRANSFER and BLDINDEX are called. There also are times when disk data management calls KEYSORT internally because it has determined that an index's overflow area needs sorting (after records have

been added to an indexed, non-shared file, for example). And finally, the KEYSORT procedure can be called explicitly from the command line or a user-written procedure.

The problem with KEYSORT is that many times when it is called it often does not really KEYSORT the file — it just acts like it does. KEYSORT compares the number of records in the overflow area with the total number of records in the file and *will not* really keysort the file if the ratio of new records to total records is under a certain limit (about 7 percent). This means that for large indexed files, a large number of index entries have to be in the overflow area before a KEYSORT will really be performed. This applies no matter how KEYSORT is initiated. Even at IPL time when the SSP says "Sorting keys for file xxxxx," it might not be.

Technical Note

Force a *real* keysort by using "KEYSORT filename,,,CHKDUP".

Fortunately, you can force a real keysort with the KEYSORT procedure by calling KEYSORT with "KEYSORT filename,,,CHKDUP". The CHKDUP parameter forces KEYSORT to really keysort the file (so that it can find duplicate keys). If the file contains duplicate keys (even if duplicates are allowed), you will receive a duplicate key message that you can bypass by responding with a 1 to the second message. As long as you reply with a 1 the file will remain keysorted.

An important consideration with KEYSORT is having enough disk space available. KEYSORT is just like #GSORT in that it uses both a disk work area and main storage. More of either will speed KEYSORT up. However, unlike #GSORT, KEYSORT is designed not to fail in the absence of enough disk space for a work file. Therefore, there are two kinds of KEYSORT:

- A work-file keysort, used when enough disk space is available.
- An in-memory keysort, invoked when enough *contiguous* disk workspace is not available for the keysort.

An in-memory keysort can take *days*! When it happens, no messages are issued; your S/36 just nods off as it thinks itself into oblivion keysorting the index without a disk work file. The only way out of an in-memory keysort is to IPL, bypassing file rebuild; then clear off some disk space and re-IPL with file rebuild.

To avoid the in-memory keysort disaster, you should make sure your system has enough disk space for the work-file keysort method when keysorting large files. Use the formula in Figure 3.23 to determine the disk space

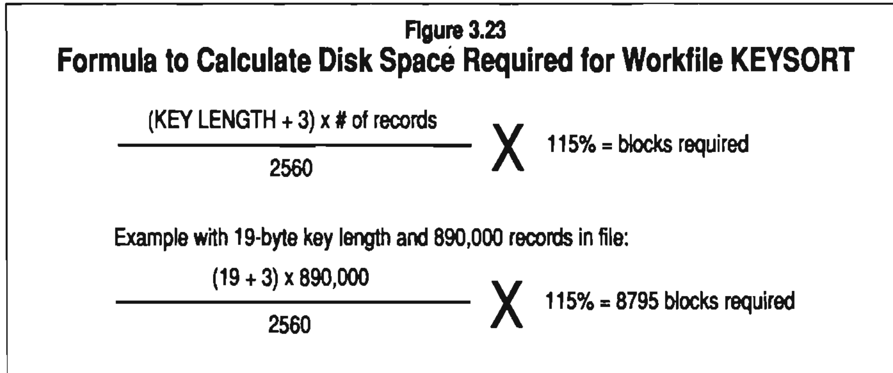


Figure 3.23
Formula to Calculate Disk Space Required for Workfile KEYSORT

required for a given file. Use this formula with appropriate values from your largest file to determine if your S/36 has enough disk space to perform the much faster work-file KEYSORT.

Disk data management's silent performance killers are lurking in your S/36 right now, waiting for just the right time to bring it to its knees. This chapter has exposed all the under-the-hood things you need to know about disk data management and how to avoid those otherwise invisible assassins. The tips and tools presented here should give you all you need to design and modify your applications to work within the constraints and limitations of the S/36's disk data management system.

Performance Tip

Don't assume there is enough disk space on your S/36 for work-file KEYSORTs. Use the formula in Figure 3.23 to be sure. (Also see the SLOWKS Keysort Alert Utility in Chapter 12.)

Section II

Hardware Adviser

"Parts is parts."

—Fast food commercial

One manager of a large S/36 installed base said "Keeping a System/36 alive and well today is one part inspiration, two parts perspiration, and five parts telephone calls." Sure, the S/36 is reliable, but when you need to fix or upgrade your hardware, you quickly discover that just finding out what's available takes most of your time. Gone are the days when you could call your friendly IBM SE and get quick answers to configuration, capacity or performance questions. Gone are the days when you could easily slide past management the cost of a hardware upgrade. Gone are the days when your SE would drop by and take you out to lunch. Today you have to scrounge for information, pay through the nose for consulting services, beg for budget dollars, and buy your own lunch.

This section should make your life easier, at least in terms of access to information and the cost of consulting fees. Think of it as your own personal Systems-Engineer-In-A-Box. Chapter 4 is a catalog of S/36 models, memory and disk capabilities, options, configuration charts, and forgotten hardware lore. If IBM sold it, we list it and tell you what it's good for. Chapter 5 covers the two commodity items in the S/36 hardware market — memory and disk — explaining how the S/36 uses them and how much of each you should have. Reading this chapter probably will send you shopping for one or both. And if the advice in these two chapters doesn't keep you busy enough, Chapter 6 is a gold mine of relevant information, from CPU upgrade tips to running applications across two machines to getting cheap hardware maintenance.

Let this section be your in-house expert on keeping S/36 hardware alive and productive. You're on your own for lunch.

Chapter 4

S/36 Models and Configurations

Although IBM still sells new System/36 hardware through its reseller channels, you will be hard pressed to hear any IBMer recommend the S/36 as a solution. Understandably, IBM strives to migrate customers to the official S/36 "replacement," the AS/400. However, moving to the AS/400 is anything but a foregone conclusion for many S/36 users. The advent of Unix- and PC-based solutions, and the high initial cost of migrating to the AS/400, have led most of the S/36 installed base to wait and watch as alternative platforms mature. (See Section VII, "Into the Future," for more details about alternative successors to the S/36). If you're among the waiting masses, you may continually face the need to expand your existing system, or even to install additional systems.

Most S/36 systems have considerable room for expansion. According to a survey by Elms Technical Communications, the average S/36 not already migrating to another platform has 1.6 MB of memory (out of a possible 8 MB), 409 MB of disk (out of a possible 1,400 MB), 18 local devices (out of a possible 72), 2.5 remote devices (out of a possible 64), and 2.2 communications lines (out of a possible 8). However, not all S/36 models accommodate the maximum configuration values just quoted; thus not all users can expand their existing systems. Further, IBM announced many capacity enhancements in recent years but made little attempt to keep its installed base apprised of these enhancements, leaving many users in the dark about expansion possibilities that could help them extend the life of their system.

Deciding to upgrade and actually figuring out how to do it are two different things. IBM doesn't offer a single document explaining all the S/36 configuration and feature options — and getting verbal quotes from IBM about upgrading an existing CPU, or (shudder) buying additional CPUs, isn't easy when the sales force is geared toward moving AS/400s. IBM's latest product information tools (e.g., the IBM Fax Information Service), don't even mention the S/36. Third-party vendors have better information, but many won't tell you about options not in stock, or that the vendors themselves are not equipped to install.

What S/36 users need is a concise, complete guide to S/36 hardware options. This chapter provides that guide. Starting with an overview of available models and progressing through memory, disk, workstation, communications, and other features, this chapter gives you a recap of hardware options, accompanied by "cheat sheets" useful for later reference. You may discover that your system has room for additional memory and disk that can lead to

Figure 4.1
Models and Features Overview

Feature	5360	5362	5363	9402	5364
Main storage (min/max K)	128-8192	128-2048	1024-2048	1024-2048	256-1024
Disk capacity (min/max MB)	30-1438	30-660	65-1256	160-760	40-130
Diskette drive	8" (magazine*)	8"	5 1/4"	5 1/4"*	5 1/4"
Tape drive (e = external, i = internal)	8809e* 6157e*	6157e*	6157i	6157i*	6157e*
Local workstations (basic/extended)	6/36/72	6/28	16/28	14/28	6/16
Remote workstations (max)	64	64	16	64	16
Communications lines (max)	8	4	2	3	2
LAN attachments (4/16 token ring)	2	2	2	2	1
Other peripherals	MICR	n/a	Optical Disk	Optical Disk	n/a

(* = optional)

improved performance. Or you might find that the S/36's little-known Token-Ring LAN or optical storage capabilities can fill a technological gap in your shop. Whatever your situation, if you have a S/36, this concise guide to hardware options is essential reading before making an upgrade decision.

Overview of Models

Figure 4.1 itemizes various S/36 models and their respective maximum configurations. A brief look at the history of these models can help you put their capabilities in perspective.

The Maytag-sized 5360, the big brother of the line, has unique features. It is the only model that supports a diskette magazine drive (automatically processing up to 23 diskettes at one time), the 8809 (reel-to-reel) tape drive, more than 2 MB of memory, or more than four communications lines. It's also the only model that accommodates more than 28 local workstations. If you're anticipating installing another S/36, and you need any of these capabilities, the 5360 is your only choice. On the other hand, the 5360 is the bulkiest and most expensive S/36 to maintain and upgrade. Considering it has only modestly better performance and disk capacity than its younger siblings, you should try hard to live within the limitations of other models. Four 5360 variations — the A, B, C and D models — represent increasing performance and capacity. The A model was limited to 128 K memory and 60 MB of disk and is obsolete. The C model was only available from IBM for a limited time to permit use of 358 MB disk drives before the D model arrived. In the real world, you're likely only to encounter B and D models. However, the performance and capacity differences

between the B and D models — the D is twice as fast, has two times the disk, and has four times the memory capacity — dictate considering only D models for upgrades. Rather than buying a 5360 B, you're better off choosing a 5362 or 5363, which equal or exceed the B model's capacity and performance.

The deskside 5362, about the size of a two-drawer file cabinet, represents the best buy on today's used market. The 5362 is the only model, besides the 5360, that supports 8-inch diskettes and accommodates up to 64 remote workstations (the number of local workstations is limited to 28). It has the same CPU performance and memory capacity as a 5360 B, but only half the disk capacity. The 5362 also supports a UPS power connector, which when connected to an Uninteruptable Power Supply (UPS), sends a console message requesting shutdown when low battery power is detected. Where disk capacity isn't critical, as in a dedicated development machine or communications hub, the 5362 can be ideal.

The desktop 5364 requires an attached PC/AT and special interface hardware and, like the 5360 A models, is obsolete. Not only is the 5364 slow and of limited capacity, but many users also report reliability problems. Don't consider putting one of these in service unless you receive it as a donation!

The most recent (and probably final) additions to the S/36 family are the identically performing 5363 and 9402 "Y10" models. Both are deskside units about half the width of the 5362, but with greatly increased performance and disk capacity. The CPU, in fact, is identical in performance to the 5360 C model. Both the 5363 and 9402 can have an internally integrated PC/AT (the *processor expansion* feature) that accepts communications and LAN attachment cards; this integrated PC makes these models better LAN platforms than the 5360. The only real differences between the 5363 and Y10 are in name and packaging. IBM announced the 5363 first, then later re-released the machine as the "AS/Entry" — officially claiming it as a member of the AS/400 family. IBM subsequently announced the 9402 Y10, which is a S/36 in an AS/400 cabinet that you can upgrade to a genuine AS/400.

Although the 9402 Y10 is a S/36 through and through, it incorporates a few AS/400 packaging features: the AS/400 cabinet and front panel; an optional five-minute battery backup unit; a battery-backed, time-of-day clock; and optional remote power-on via telephone. You can upgrade a Y10 to the AS/400 9402 E04/E06, although the upgrade is currently impractically expensive — you would be better off buying a used AS/400 9402 E06 and keeping the Y10. The Y10 also limits you to a total of 760 MB of disk with a slower data transfer rate than the 5363. This radical constraint on disk capacity, compared to the 5363, implies that IBM wants to "encourage" Y10 owners to move on to the AS/400. Unless you really need the unique Y10 features, or you have access to a cheap Y10-to-AS/400 upgrade, avoid it in favor of the 5363, which supports nearly twice as much disk storage and provides better performance.

Figure 4.2
S/36 CPU Relative Computational Performance

CPU Model	Relative Performance Factor
5364	1.0
5362	2.4
5363	2.5
5360 B	2.6
5360 D	4.3

When choosing a replacement CPU, remember that in most environments, disk, not CPU speed, constrains performance. Given the multiprocessor nature of the S/36, it's not always easy to determine relative performance differences between various models. Figure 4.2 shows average relative performance comparisons for computational loads based on IBM tests using representative business calculations, with the 5364 given a weight of 1.0. Keep in mind, though, that a faster CPU won't improve performance much if disk accesses are the constraining factor. Thus, even though a 5360 D CPU, with a relative rating of 4.3, seems like it should be nearly twice as fast as the 2.5-rated 5363, in practice the two perform alike because of the 5363's disk speed advantage.

No matter which model S/36 you have presently, don't rule out purchasing a different CPU outright. It's often cheaper to buy and upgrade a used late-model S/36 than it is to update a more obsolete model — and you would still have the old machine for backup or other purposes. One interesting side effect of the S/36's ongoing popularity is that used 5363s often have a higher residual value than equivalent early model AS/400s, even though the AS/400 is newer and supposedly better!

Memory

If there is one thing you should learn from this book, it's that you can never put too much memory in your S/36. Because SSP automatically takes advantage of additional memory to keep frequently used system programs resident, and because you can use that memory — via cache, storage indexes, and resident screen formats — to further improve performance, installing the maximum memory your machine can handle is nearly always a good move. (For more information about using memory effectively, see Chapter 5, "The Importance of Memory and Disk Space.")

Figure 4.3
Memory Configurations

Memory card size	Number of Cards				
	5360	5362	5363	9402	5364
128 K	4	4			2
256 K	4	4			
512 K	4				
1024 K	4	2	2	2	1
2048 K	4				

How much memory you need to buy to fully configure your machine depends upon how much you already have and how it's organized. S/36 memory comes as plug-in cards of various capacities, and each S/36 model has a fixed number of card slots. To increase your machine's memory, you first may have to remove some or all of the existing memory cards (Figure 4.3 itemizes memory card capacities and counts for various S/36 models).

Only the 5360 lets you use lots of memory — up to 8 MB with third-party enhancements — all other models are limited to 2 MB. The 5360 is also the most confusing model in which to install memory — it accommodates five different card sizes in four slots, and a given machine may have any combination of cards already installed. Because only four slots are available, you must use four 2 MB cards to achieve the maximum 8 MB capacity. Any existing cards of less than 2 MB capacity must be removed (you can sometimes obtain credit for these from your memory supplier). Note that although IBM officially supports only 7 MB (three 2 MB cards and one 1 MB card), the 5360 D accommodates 8 MB with no problem. One third-party supplier (AI/GBT, 12450 Beatrice Street, Los Angeles, CA 90066, (800) 243-4433 or (310) 305-8616) offers an 8 MB upgrade package that includes a software patch necessary to satisfy IBM's configuration program.

The 5362's four slots work with 128 K, 256 K or 1 MB cards in any combination, up to 2 MB. Only the 256 K and 1 MB cards have any residual value, though, so you might consider keeping your old cards for an emergency; if a memory card fails, you can get your machine up again, albeit running more slowly, while you wait for replacement parts.

The 5364 has zero upgrade options — a short and sad story — and another reason to avoid this model. The 5363/9402 duo accommodate a single additional 1 MB memory card, bringing total memory to 2 MB.

Technical Note

Only the 5360 D model supports more than 2 MB of memory. You may encounter some rare C models that support exactly 2 MB. All other 5360 models limit main memory to 1.75 MB.

Disk

Depending on the model, the S/36 supports from one to four disk spindles (Figure 4.4 lists the possible configurations). The number of configuration options for each S/36 model is limited, and performance and other characteristics of disk drives vary widely. Because disk is the most common performance bottleneck on the S/36, you should evaluate disk drive characteristics carefully when selecting a S/36 upgrade path.

The characteristics critical to disk drive performance are average seek time and rotational delay, number of bytes per cylinder, and effective data transfer rate. Average seek time and rotational delay determine the time required to access data randomly; the number of bytes per cylinder and effective data transfer rate determine overall drive throughput. Figure 4.5 compares these characteristics for the seven available disk drive models.

Of the available drives, the 5363 and 9402 models, ranging in capacity from 67.5 MB to 314 MB, have the best random access performance. (The model numbers for the disk drives supported by the S/36 5363 and 9402 are the same as the machine model numbers.) Their combined seek times and rotational delay yield an access time of less than 20 ms — about twice as fast as the higher-capacity 10SR drive. For applications requiring frequent random-record retrieval (e.g., interactive applications), choose the 5363 or 9402 CPUs for their quick access. A 5363 with four 314 MB drives offers nearly as much disk capacity (1256 MB) as the 5360 D (1436 MB), running at about half the access time. This makes the 5363 a very good interactive application platform. The 9402 supports less total disk (760 MB), with a slow data transfer rate (0.83 megabits per second), making it a less attractive candidate for interactive work.

For batch work, effective data transfer rate is a controlling characteristic — it determines how quickly data can be read sequentially from the disk. Here again the 5363 shines, with a data transfer rate second only to 9332 drives. The 9332 drives are something of an enigma: Available only on the 5362, they use the ANSI Intelligent Peripheral Interface (IPI-3), which supports data transfer rates of nearly 6 megabytes per second! While the 5362 processor can't always sustain this high throughput, the 9332 shows markedly improved performance when batch processing files. The 9332, a general-purpose, externally housed drive, is also available on the AS/400 in 200 MB, 400 MB, and 600 MB capacities. The 5362 supports only the 200 MB and 400 MB models, however, and

Figure 4.4
Disk Configurations

Drive model and capacity	Number of Spindles				
	5360	5362	5363	9402	5364
Maximum system capacity (MB)	1438	660	1256	760	130
Maximum internal spindles	4	2	4	4	2
Maximum external spindles	0	2	0	0	0
21ED 30 MB	2	2			
21ED 60 MB	2	2			
10SR 200 MB	4				
10SR 359 MB	4				
9332 200 MB (external only)		2			
9332 400 MB (external only)		1*			
0665 40 MB					1
0665 65 MB					1
5363 65 MB			4	4	
5363 105 MB			4	4	
5363 314 MB			4	4	
9402 160 MB			4	4	
9402 200 MB			4	3	

(*N/A if internal disk > 60MB)

Figure 4.5
Disk Drive Characteristics

Characteristic	Drive Model						
	21ED	10SR	9332	0.665	5363	5363	9402
Capacity (MB)	30.8/61.6	200.9/359.7	200	41.9	67.5/106.2	314	160/200
Average seek time (milliseconds)	35	25	19.5	40	30	12.5	12.5
Average rotational delay (ms)	9.52	10.1	9.6	8.33	8.1	6.95	6.95
Bytes per cylinder	69,632	351,232	151,552	57,344	116,480	257,280	172,032
Effective data transfer rate (MB/sec)	0.941	1.18	5.7	0.625	1.25	1.25	0.83

only permits attachment of the 400 MB unit when the internal 5362 disk doesn't exceed 60 MB. (For a time, IBM offered the 600 MB model for the 5362 as an RPQ, but that RPQ since has been discontinued. You may find such machines on the used market, though.)

The third critical characteristic — the number of bytes per cylinder — can affect both batch and interactive processing. This figure represents the number of bytes that can be read or written without moving the disk actuator arm. For random access of indexed files, a high value means fewer disk seeks to search an index. For batch processing, a high value means fewer disk seeks during sequential processing. Of all the drives, the 5360's 10SR has the largest number of bytes per cylinder: 351,232. But the 5363's 314 MB drive takes a reasonably close second place, at 257,280 bytes per cylinder.

In general, the S/36 platform with the best balance of disk performance and capacity is the 5363. The 5362 places second with its fast 9332 drives. All other disk options represent old and slow technology best avoided by upgrading.

Workstations

You probably use your S/36 more for interactive than for batch work. A key configuration point for various S/36 models is workstation support — the number of users supported by a given CPU model. For best performance, you're interested in local workstations: devices attached directly to your machine via high-speed (1 megabit per second) twinax lines.

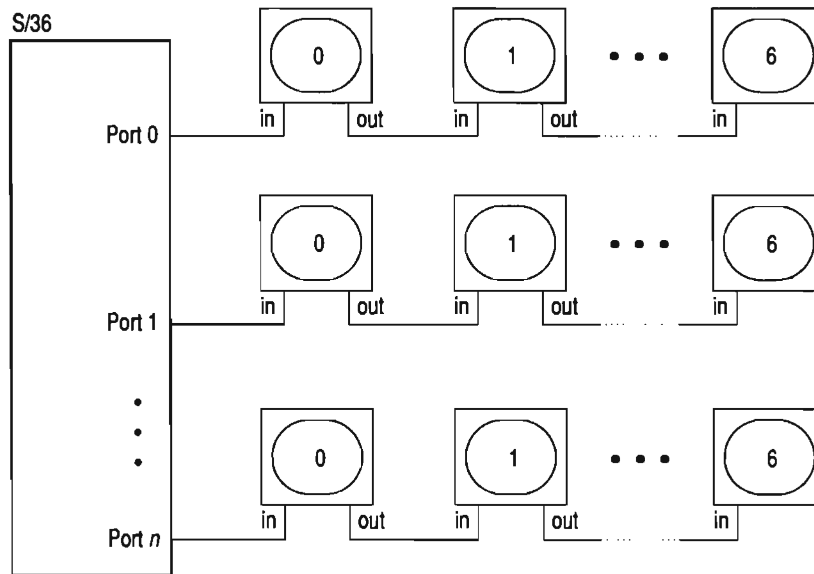
You connect twinax devices — either display stations or printers — to the S/36 in daisy-chain fashion ("pass-through" in IBM-speak): Each device has two connectors for a cable-in and cable-out (Figure 4.6). Each string of up to seven devices attaches to a single S/36 twinax port. The total number of devices each system supports depends upon both the number of ports and the amount of *workstation controller storage.* All S/36 models come with a base amount of workstation controller storage supporting a certain number of devices; you increase that number by adding one or more *workstation expansion* features. Figure 4.7 shows the number of local twinax ports available on various S/36 models and the number of devices supported in base and expansion configurations. One special feature — extended input fields — increases the number of available input fields from 128 to 256, which lets 5250 terminals emulate 3270 displays (which can have up to 256 input fields). However, the extra input fields use up workstation controller storage, which reduces the number of devices that can be supported (as reflected in Figure 4.7). You can run twinax cable up to 5,000 feet, although the number of devices and splices in a given line can reduce this distance considerably.

An alternative to twinax is unshielded twisted-pair cabling (UTP), which uses lightweight, inexpensive, standard telephone wire. The primary advantage of twinax cable is its high noise immunity, which it achieves by sending the data signal down two wires simultaneously. One signal is the exact inverse of the other signal, which produces a uniform, or *balanced,* magnetic field in the cable. The balanced signal is less susceptible to corruption by stray external electronic noise and internal signal reflections. The term "twinax" refers to the

Figure 4.6
Workstation Configurations

Workstation feature	Number of Ports/Devices				
	5360	5362	5363	9402	5364
Basic (ports/devices) with extended input fields feature	6/6	4/6 4/3	4/6 4/3	4/6 4/3	1/6 1/3
First WS expansion (ports/devices) with extended input fields feature	6/36 6/18	4/28 4/18	4/28 4/18	4/28 4/18	2/16 2/9
Second WS expansion (ports/devices) with extended input fields feature	12/72 12/36				

Figure 4.7
Twinax Daisy-Chain Local Workstation Network



two center conductors, although the cable actually has a third ground conductor in the form of a braided metal shield.

Twisted-pair wiring has only two conductors — a signal and a ground — with minimal to no shielding. The single signal wire means the magnetic field is *unbalanced*; it is more susceptible to noise, which effectively cuts the

maximum run length to 1,000 feet or less. For many applications, though, this is enough. Cheaper wiring and maintenance costs make twisted-pair an attractive alternative to twinax.

You attach twinax devices to a twisted-pair wiring system using *balun* (*bal*anced-to-*un*balanced) connectors, which convert the balanced twinax signal into an unbalanced twisted-pair signal. Because twisted-pair cabling is cheap and run lengths often short, you can forego twinax-style daisy chaining and use a more convenient star network topology (Figure 4.8). In a star configuration, each terminal connects to a central patch panel, which in turn connects to the S/36 twinax ports through baluns. The star topology lets you easily add new connections and reroute existing ones. It also lets you detect and isolate bad connections without blocking other devices; in a twinax daisy chain, all devices downstream from a failing device are forced offline.

Technical Note

Detailing the great variety of 5250 devices available is outside the realm of this book. Both terminals and printers, with a wide range of features, are available from IBM and numerous third-party manufacturers. When upgrading your 5250 network, you might consider using PC or Macintosh 5250 emulation boards, which offer both text and windowed emulation of multiple 5250 terminals, and add sophisticated features such as cut-and-paste, keyboard macros, and spreadsheet data translation.

If you run out of local device capacity, or need to provide access to remote users, you can attach additional devices via Remote Workstation Support (RWS). RWS uses communications lines to connect additional devices that appear as 5250 terminals to SSP and your programs. One or more communications lines serves as an interface to a *remote workstation controller*, a device that provides the same functions as the local workstation controller built into the S/36 system unit (Figure 4.9). The remote controller can be located in the same room as the S/36, or on the other side of the world, depending on the kind of communications connection you use. In general, though, you're limited to data rates much slower than twinax's 1 Mbps. Remote line speeds range from 57.6 Kbps for local modem-eliminator connections to 38.4 Kbps for leased lines and 19.2 Kbps for switched lines. The maximum speeds available, and number of attached devices, depend on CPU model and the communications adapter used. (For a description of various communications connections, see the Communications section in this chapter).

Figure 4.8
Twisted-Pair Star Network Topology

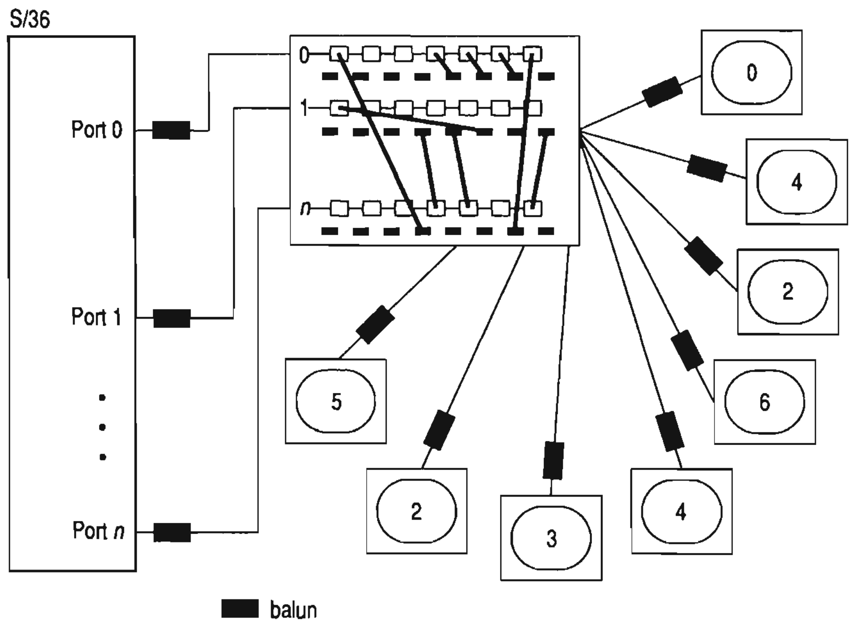
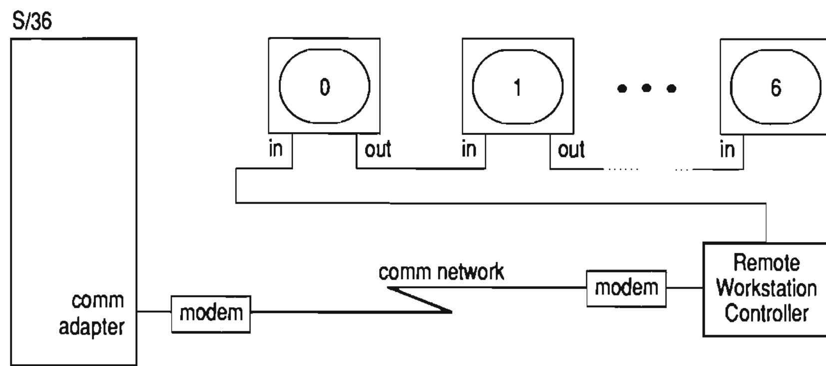


Figure 4.9
RWS Support Using Communications Lines and Remote Workstation Controllers



Technical Note

Although a detailed discussion of remote controllers is outside the scope of this book, you'll find four basic types on the market. The first, and oldest, is the 5250 Model 12, a 5250 workstation containing a built-in remote controller, to which you can attach up to seven additional 5250 terminals. The Model 12 controller has its microcode fixed in read-only storage, and was developed before IBM added DisplayWrite word-processing functions such as word-wrap and spell-check to 5250 support. Thus the Model 12 doesn't support DW/36. The second-oldest remote controller is the 5294, a standalone equivalent to the Model 12 that supports downloadable microcode and, thus, DW/36. The successor to the 5294, the 5394, supports up to 16 remote devices. Finally, the IBM PC supports individual remote terminal emulation cards that let a PC emulate a remote workstation controller and one or more 5250 terminals.

If you need to connect ASCII or 3270 (IBM mainframe) terminals to your S/36, you'll need a *protocol converter*, which translates foreign-terminal protocols to 5250 data stream commands. IBM offers two: the 5208 ASCII-5250 Prototcol Converter and the 5209 327X-Link Protocol Converter. Each protocol converter lets you connect up to seven foreign terminals to a single twinax port. The 5208 supports "dumb" ASCII terminals using VT-100 protocol; the 5209 attaches 327X terminals. With the 5208, you can even use dial-in switched line connections to support remote access using asynchronous modems and ASCII terminals or PCs emulating ASCII terminals. Similar devices are available from several third-party manufacturers.

Communications

The S/36 supports a myriad of communications options through one or more communications lines. The S/36 offers three different *communications adapters*:

- SLCA (Single-Line Communications Adapter)
- MLCA (Multi-Line Communications Adapter)
- ELCA (Eight-Line Communications Adapter)

These support up to two, four, or eight lines, respectively. Each adapter in turn supports one of seven *physical interfaces*:

- EIA/CCITT (RS/232)
- DDSA (Digital Data Service Adapter)
- X.21 (synchronous leased and switched)
- X.25 (packet switched)

Figure 4.10
Communications Configurations

Feature	5360	5362	5363	9402	5364
Single line/aggregate max speeds (bps)					
SLCA	9,600/9,600	9,600/19,200	19,200/64,000	19,200/64,000	9,600/9,600
MLCA	57,600/67,200	57,600/115,200			
ELCA	19,200/170,000				
Speed limits per line (blank=not avail)					
EIA/CCITT adapter (SLCA/MLCA)	9,600	9,600	19,200	19,200	9,600
EIA/CCITT adapter (ELCA)	19,200				
DDSA adapter SLCA	9,600	9,600			
DDSA adapter MLCA	56,000	56,000			
DDSA adapter ELCA	56,000				
X.21 adapter SLCA	9,600	9,600			
X.21 adapter MLCA	9,600	19,200			
X.21 adapter ELCA	19,200				
X.25 SLCA/MLCA	9,600	19,200	9,600	9,600	
X.25 ELCA	19,200				
V.35 SLCA/MLCA	57,600	57,600	64,000	64,000	
V.35 ELCA	57,600				
ASYNC	9,600	9,600	9,600	9,600	9,600
BCS	9,600	9,600	9,600	9,600	9,600

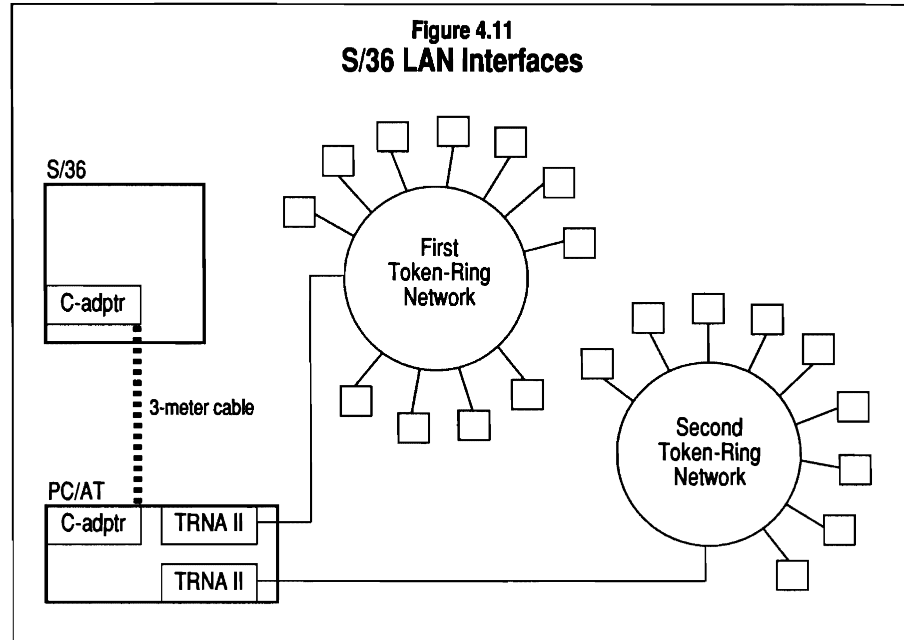
- V.35 (European switched)
- Asynchronous
- BSC (Bisync)

The line speeds available for each line depend on the kind of adapter and physical interface used. SLCA has the slowest speeds because it shares the native S/36 CSP as a communications controller. The MLCA and ELCA adapters each have dedicated communications processors, and support much higher transmission speeds. Each adapter supports a maximum *aggregate* line speed, which is the sum of all the individual line speeds. Individual lines also have maximum speeds. One line can usually run faster if no other lines are in use; that speed is called the *single* line speed. Figure 4.10 summarizes communications line speeds for various combinations of adapters and physical interfaces.

Local Area Networks

IBM added Token-Ring LAN connectivity to the S/36 late in its life; and in fact LAN support is one configuration option often overlooked even by veteran

Figure 4.11
S/36 LAN Interfaces



S/36 users. Don't you overlook it: LAN access offers a sophisticated, high-speed connection to other S/36s, AS/400s, S/370s, and PCs that could extend the useful life of your S/36. The LAN moves data at either 4 or 16 megabits per second between up to 260 attached systems, making it the widest bandwidth connectivity option available for the S/36.

Figure 4.11 shows how you connect your S/36 to a Token-Ring LAN. In every case, the actual LAN connection is provided by a Token-Ring Network Adapter (TRNA) II card installed in a dedicated PC/AT (model 5170). You can connect to a second LAN via a second TRNA card. S/36 5360 and 5362 CPUs use an external PC/AT with 512 K of memory and no hard drive. A keyboard and display, required for diagnostics but not for normal operations, are optional. The PC connects to the S/36 through a high-speed channel adapter board connected to a similar board in the S/36 via a special three-meter cable. The channel adapter provides direct memory-to-memory transfer between the PC and the S/36, which is necessary to sustain the high LAN data rates. The 5363 and 9402 Y10 CPUs don't need an external PC/AT; instead, you use the internal PC/AT which serves as the processor expansion interface. The 5364 uses the PC attached as a console controller to hold the TRNA and channel adapter cards.

Technical Note

5360 LAN support requires a Stage 2.1 or later processor. All other S/36 models contain the correct processor for LAN attachment. LAN software requires SSP 5.1 or later.

The S/36 also needs special software to make the LAN connection operational: the S/36 LAN Communications Program Product (5727-EP1 for the 5360/5362; 5727-EP6 for the 5363/5364/9402). This software lets you configure either one or two LAN adapters as communications lines 9 and 10. Each LAN supports up to 50 devices for the 5360/5362 CPUs, and up to 15 for 5363/5364/9402 CPUs. Connecting both TRNAs to one Token-Ring LAN doubles the number of available devices for that network.

After installing the hardware and LAN Communications Program product on the S/36, you configure LAN support via CNFIGSSP. When you activate LAN support (via ENABLE for lines 9 and 10), SSP automatically downloads LAN support code into the PC/AT, establishes a presence on the network, and becomes ready to establish active sessions.

With S/36 LAN access enabled, you can establish several kinds of Token-Ring connectivity (Figure 4.12 illustrates various LAN connection options):

- PC Support Server functions: up to eight virtual disks, eight shared folders, three virtual printers, and one file transfer session for each LAN session.
- PC Support Workstation functions: up to five 5250 or 3278 emulation sessions, one of which may be a printer session. (All workstation functions for all users run under a single LAN session for a given S/36.)
- Advanced Program-to-Program Communications (APPC) functions: Distributed Data Management (DDM), Display Station Pass Through (DSPT), Advanced Program-to-Program Network (APPN), File Transfer, and the APPC Application Program Interface (API)
- S/370-specific services: SNA Upline Facility (SNUF), Multi-Station Remote Job Entry (MSRJE), 3270 device emulation, Communications and Service Management (CS&M).

Optical Storage

The S/36 has long been used to store and process accounting and other electronic business records, using magnetic hard disks for short-term storage of current data and diskettes or tapes for long-term storage of historical data. The advent of fast optical scanners has also made it possible to store paper

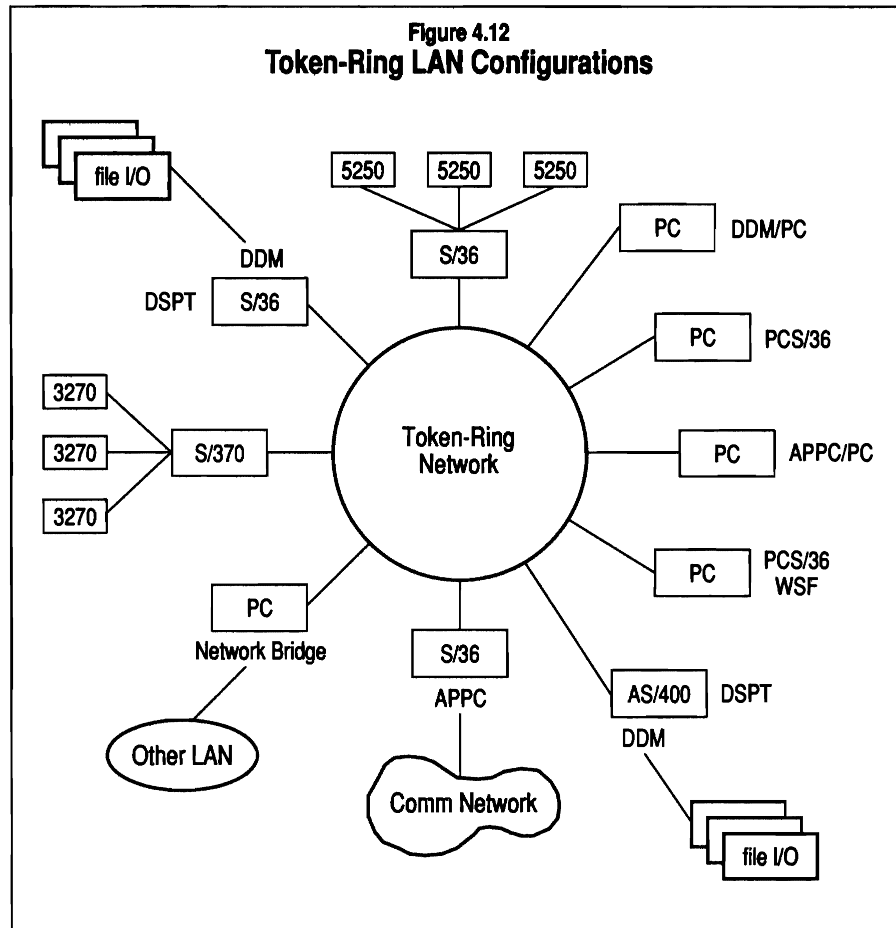


Figure 4.12
Token-Ring LAN Configurations

documents, such as letters, claim forms, invoices, and checks as digitally encoded images. Such images, while quickly accessible by computer, require considerable digital storage space — on the order of 50 K for a single 8- by 10-inch form. A textual customer file formerly requiring less than 10 K of disk space could balloon to 1 MB or more once you start storing a digitized paper trail. At $10 per megabyte, traditional magnetic disk storage is just too expensive as a replacement for paper.

However, the cost per megabyte of optical storage — particularly write-once-read-many (WORM) optical discs — runs less than $1 per megabyte, making digitized record storage practical. Optical discs of the WORM variety use a laser to permanently etch a digital bit pattern upon the disc's rotating surface. Each disc holds a gigabyte or more per side.

The S/36 5363 lets you attach IBM's 9247 Optical Storage drive ($20,000) via a special RPQ hardware adapter. The 9247 uses 12-inch optical cartridges costing about $300 each, and holding 2 gigabytes (1 gigabyte per side). The drive reads only one side at a time, requiring a "flip" operation to change sides. With an average seek time of 150 milliseconds, you won't mistake the 9247 for a hard disk drive. However, 150 milliseconds is certainly much quicker than retrieving paper records manually from a wall full of file cabinets. Once data has been located, the 9247 transfers it at a rate of 1.2 megabytes per second, equivalent to hard-disk transfer rates.

The 9247 is a single drive; for those really big data storage jobs, you can use the 9246 Optical Library (about $200,000), a five-foot cube holding 128 gigabytes in 64 disc cartridges. The library also contains from two to four 9247 drives that operate simultaneously to provide reasonable access times for multiple users.

The 5363 operates as a controller for the drives, while providing file server facilities for attached PC workstations. You can use APPC via communications line or LAN to provide optical storage access to other host S/36s or AS/400s. Two software packages provide basic optical storage access: Optical Drive Support (5799-DAA) and Optical Library Support (5799-DAB). These packages use S/36 Folder Management Services to organize objects on optical discs, using S/36 hard disk storage as a "staging area." An "object" can consist of text, digitized images, documents, or sound files. Virtually any kind of data file you can generate on the S/36, AS/400, or PC can become an optical storage object. Your application programs interact with the optical storage subsystem using ICF INTRA (for local programs) or APPC (for programs running on a remote system). Figure 4.13 shows how application-requested documents move from a permanent folder stored on optical disc to a working folder on the 5363. The staging process lets you manipulate optical-based objects efficiently. Because WORM storage can never be erased, you want to minimize the number of WORM write operations; with staging, your application only writes the document on the WORM when an updating session is completed.

Although the WORM contains all previous versions of a given folder object, IBM's software only lets you access the most current version. The software does, however, let you view an audit trail of versions for a given object.

Instead of writing your own application programs to provide image processing services, you might consider IBM's S/36 Workfolder Application Facility, or the SAA Workfolder Application Facility for the AS/400. Both provide image scanning, cataloging, retrieving, and process-tracking capabilities required for most office automation tasks.

Conclusion

The foregoing picture of S/36 hardware configurations should give you a good

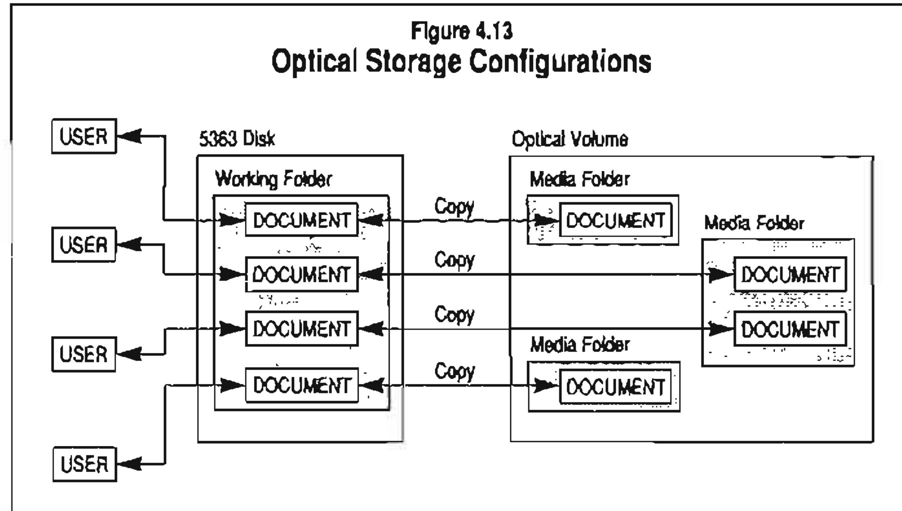


Figure 4.13
Optical Storage Configurations

idea of the expansion options for your existing system. If you have a 5360 C with many users, the first order of business should be upgrading to a D model and 8 MB of memory. If you have one or more 5362s, make sure you have 2 MB of memory in each. Any time you increase memory, don't forget to make sure you're taking advantage of the memory (see Chapter 5 for memory and disk management tips).

If you are unfortunate enough to have a 5364 or 5360 A model, bite the bullet now and replace it with a more capable 5362 or 5363. The 5360 A models are so limited that even a 5362 will give you much better performance, as well as greatly improved upgrade opportunities.

If you're currently maxed-out on a 5363, and interactive performance is inadequate, you might consider bringing in a 5360 D model, with the primary advantage of a four-fold memory capacity increase to 8 MB. But be careful about disk selection: Upgrade to a machine with the same or more disk spindles as your 5363, or you could see dramatically increased disk I/O, thereby dramatically decreasing performance.

Finally, avoid the 9402 Y10. It's siren song of AS/400 upgradability is deceptive, given the high cost of upgrading. And in the bargain, you'll be locked into a slower, less expandable box.

Chapter 5

The Importance of Memory and Disk Space

Many S/36 users face a true dilemma: The S/36 is doing a good job meeting current needs, but it won't live forever. Studies show that most S/36 users do have room to grow — to upgrade and enhance their machines. But continual improvements to the AS/400 and RS/6000 — and the emerging technologies these hardware platforms support — make the choice of upgrading or migrating a difficult one at best.

What's a S/36 user to do? Does the life expectancy of the S/36 warrant spending the money to upgrade? Is it reasonable to postpone migration for a year or two, or is the pay-off better now?

For many shops, upgrading the S/36 is the wise interim strategy. S/36 upgrades are inexpensive and easily attainable. They will help give you the time you need to intelligently sort out your migration options. Think of S/36 upgrades as "cheap gas," buying you time to let the migration alternatives mature. For less than the cost of a new personal computer, you can upgrade your S/36, adding memory and disk storage, and give your machine another two or more years of life. And even if you choose not to migrate in the foreseeable future, these upgrades will help extend the life and performance of your S/36. (See "Is the S/36 Worth Upgrading?" page 88.)

Just Add Memory

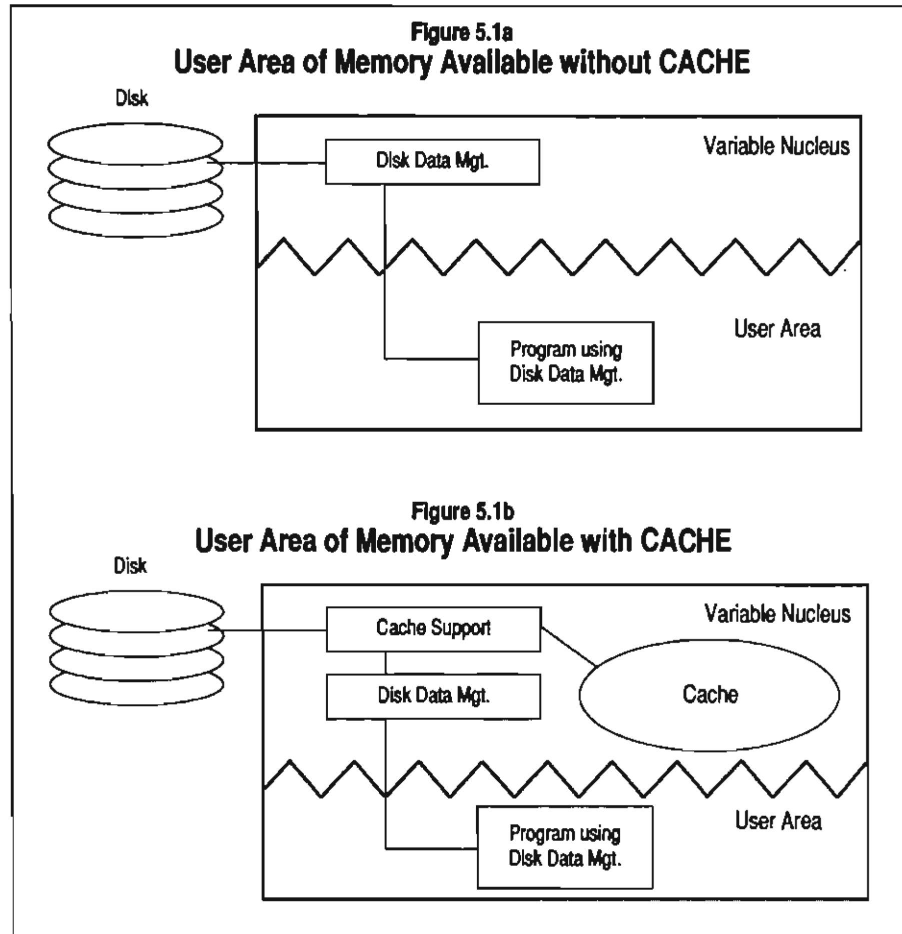
The S/36 is a virtual memory machine — that is, it allows SSP to overcommit real memory. When real memory is overcommitted, chunks of programs are paged from real memory to virtual memory workspace (the Task Work Area) on disk. This paging can dramatically impede performance. With more available memory, more programs and data stay memory-resident; performance improves because paging occurs less frequently and there are fewer disk accesses.

Your S/36 should have as much memory as it can physically handle. Don't rely on antiquated rules of thumb to determine the amount of memory required per user. Put as much memory in your S/36 as possible — as soon as possible. A flourishing third-party market offers very affordable memory upgrade prices, and there simply is no better or less expensive way to quickly and easily improve S/36 performance than to add memory.

> **Performance Tip**
>
> **Don't waste time trying to performance tune a memory-constrained S/36. Given the low street prices of S/36 memory upgrades, adding more memory to a S/36 is the cheapest, easiest way to improve S/36 performance.**

Here are more reasons your S/36 should have as much memory as possible:

The CACHE Facility. Using the S/36 CACHE facility efficiently requires



plenty of real memory. When CACHE is enabled, SSP allocates memory for it from the variable nucleus area of memory — reducing the size of the user area and leaving less real memory for your applications (Figures 5.1a and 5.1b). Therefore, using CACHE on a memory-constrained S/36 is not likely to improve performance. In fact, when used without enough real memory, CACHE-induced memory swapping will probably degrade overall system performance. Chapter 15 discusses the impact of CACHE on your system and provides you with tools to determine whether or not CACHE is beneficial for your system.

Even if you don't explicitly use the S/36 CACHE facility, the S/36 automatically uses all memory available to cache SSP routines. Many system programs are called to execute each task the S/36 performs. For example, a system program named #CIRN is called just to process the OCL // RUN statement. As

you can imagine, #CIRN is used by the system often. To avoid constantly reloading SSP routines such as #CIRN, the SSP lets many users share the same copy of the program in memory. However, depending on other real memory demands, SSP routines can be bumped out of real memory. Each time the // RUN statement is encountered in OCL and #CIRN has been displaced, SSP must reload the program from disk. Figure 5.2 shows that with only 1 MB of real memory, the number of SSP routines resident in memory is constrained, but with 3 MB of memory, more SSP routines can be resident in real memory. SSP regularly uses more than 200 routines. With enough memory, these routines can stay resident, reducing disk I/O while improving overall system performance.

Accessing Screen Formats. Having plenty of memory also allows efficient access to screen formats — by keeping frequently used formats resident. Without memory-resident screen formats, SSP must read screen formats from disk every time they're displayed. By configuring your system using CNFIGSSP's screen 17.0 (Figure 5.3) to allow memory-resident screen formats, the SSP will cache screen formats used by a program in a shared area of real memory. Not only will performance improve for the one application, but because multiple applications share this "pool" of memory-resident screen formats, performance improves for all programs using workstation I/O.

Performance Tip

Use CNFIGSSP's screen 17.0 (Figure 5.3) to enable memory-resident screen formats. This will reduce the disk accesses required to load and display screen formats for your application programs.

Using EPCs. The S/36 has external program call (EPC) capability built into its control storage processor. EPCs allow one program to call another, just as they do on the AS/400. By taking advantage of the S/36's virtual memory facility, EPCs allow many programs to be associated with one task, letting you write modular and more maintainable applications while circumventing the S/36's program size and file limit constraints. With EPCs, when real memory becomes overcommitted, SSP pages infrequently called programs to disk. Additional memory lets more called programs remain resident, improving performance and reducing disk I/O. EPC capabilities are available from IBM and from two third-party companies. If you have ongoing program development on your S/36, you should consider using EPCs to reduce wasted disk I/O and improve program portability. And, if you do use EPCs (as discussed in detail in Section III), the more real memory you give your S/36, the faster it will run.

Performance Tip

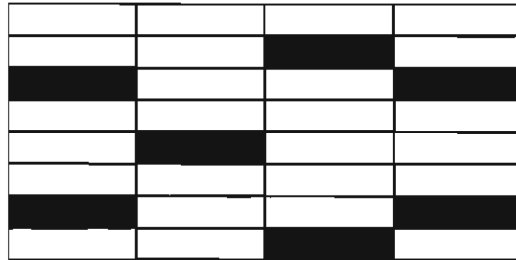
Not all paging is bad. When paging replaces more time-consuming operations — as it does when program calls replace OCL // LOADs — response time actually improves.

Memory Configuration

Figure 5.4 shows the maximum memory each S/36 model supports. A stock 5360 Model D can support up to 7 MB of memory. However, because the S/36 has an effective memory address of 23 bits (see Chapter 2, "S/36 Memory Management"), theoretically it can address up to 8,388,608 bytes of real memory. One third-party memory provider, AI/GBT, provides an SSP patch that lets the S/36 use that eighth megabyte of memory. The patch isn't needed to make the S/36 address the eighth megabyte of memory — even without the SSP patch the S/36 would "see" the additional memory and perform

Figure 5.2

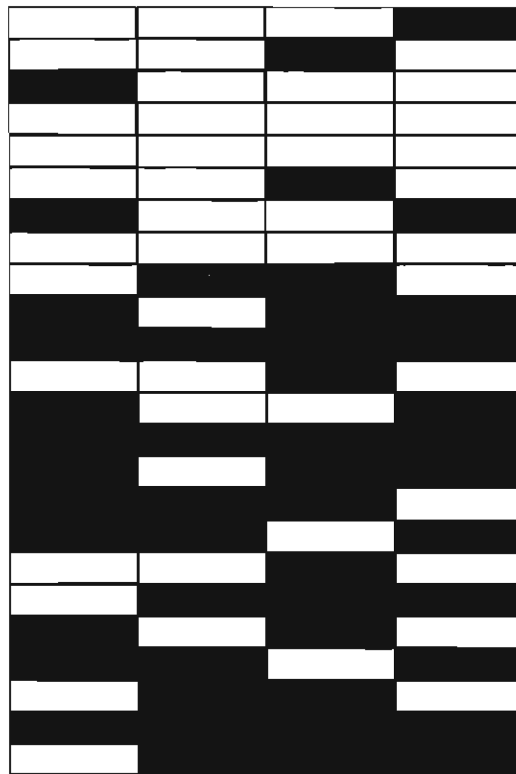
More Memory Means More System Programs Resident



1 MB of memory with a few system programs resident

System program

User program



3 MB of memory with many system programs resident

Figure 5.3
CNFIGSSP Screen 17.0

```
17.0        CNFIGSSP - BASE SSP III                         SYSCNFIG  M1

1. Default forms ID . . . . . . . . . . . . . . . . . . .    0001

2. Specify how many lines you want
   printed per page . . . . . . . . . . . . . . . 001-112    066

3. Default library name . . . . . . . . . . . . . . . . .

4. Do you want your display formats
   to reside in main storage? . . . . . . . . . . . . Y,N    Y



Cmd3-Previous menu          Cmd19-Cancel
```

Figure 5.4
Maximum Memory Per S/36 Model

Model	5360 Mod B	5360 Mod D	5362	5363	9402	5364
Memory capacity	1.75 MB	8 MB	2 MB	2 MB	2 MB	1 MB

the necessary memory diagnostics at IPL time. However, the patch is necessary to modify several CNFIGSSP screens that need to know the eighth megabyte is available. This eighth megabyte of memory could very well be just what's needed to extend the life of a currently maxed-out S/36.

5360 Model B users are limited to 1.75 MB of memory. For many S/36 5360 shops, the justification to upgrade to a Model D processor is not really the more powerful processor, but rather the additional 6.25 MB of memory capacity. If you have a memory-constrained 5360 Model B, consider upgrading to a Model D simply for the additional memory.

Performance Tip

If you have a maxed-out Model D 5360 with 7 MB of memory, be aware that you can add an eighth megabyte of memory. To find out how, contact AI/GBT, 12450 Beatrice Street, Los Angeles, CA, 90066, (800) 243-4433 or (310) 305-8616.

Pricing Memory

When searching for memory upgrades for your S/36, you will find that third-party vendors offer the best buy. IBM publishes S/36 memory upgrades in its price lists, but the prices are not competitive with third-party memory prices. IBM lists 1 MB of memory at $4,160; 2 MB lists for $8,330. Third-party vendors

offer memory upgrades starting at about $500 for 1 MB and about $1,000 for 2 MB. In many cases, the memory from the third-party vendors won't be IBM-brand memory — but it will do the job as effectively and reliably.

If you are concerned that you might lose IBM maintenance, don't worry. Buying third-party memory won't invalidate your IBM maintenance contract. With proper notification, IBM will continue your maintenance contract even if you have upgraded your machine with third-party equipment. Any reputable vendor can supply you with the form letter you'll need to inform IBM that you're upgrading your machine through a third party. Very likely — though it doesn't always happen — an IBM engineer will visit your shop shortly after this notification to "recertify" your machine for IBM maintenance. Some of the money you saved buying third-party memory must be spent to pay IBM for this visit. It's not a lot, but put $200 or so in your upgrade budget to have your S/36 recertified. Or you can avoid the situation altogether by using third-party maintenance services, which are discussed in more detail in Chapter 6.

Technical Note

For a 5360 Model B processor to support up to 1.75 MB of real memory, it must have a Stage 2.1 processor. Model Bs with a Stage 2.0 processor will support only 1 MB of memory. A Stage 2.1 processor upgrade sells for $500 or less from third-party vendors. You can check your S/36's processor level by opening the control panel cover and reading the embossed label in the lower right corner. It will say Stage 1, 2.0, 2.1, or 3.

If your third-party upgrades were performed using genuine IBM parts, after recertification IBM will adjust your maintenance contract and IBM maintenance will cover the new parts. However, if your machine was upgraded with non-IBM parts, the recertification only confirms that the third-party parts are causing no problems and that IBM maintenance will continue unchanged — covering what you had but not the new parts. Generally, when you upgrade your S/36 with non-IBM parts (currently, memory cards are the only non-IBM add-on), you'll need to provide for maintenance on these parts through the vendor. Be aware, also, that if you add non-IBM parts to your S/36, the unlikely, but possible, circumstance exists for a few finger-pointing problems ("It's *their* memory card that's causing the problem," "No, it's *their* memory card"). Some third-party memory vendors — for example, EMC^2 — provide a switch on the front of their memory card that causes the card's memory to be logically ignored by the S/36 at IPL. By helping to diagnose the location of memory-related problems, this determines whether the EMC^2 board is part of the problem.

Technical Note

Even if your S/36 is under IBM maintenance, you *can* upgrade your machine using third-party vendors.

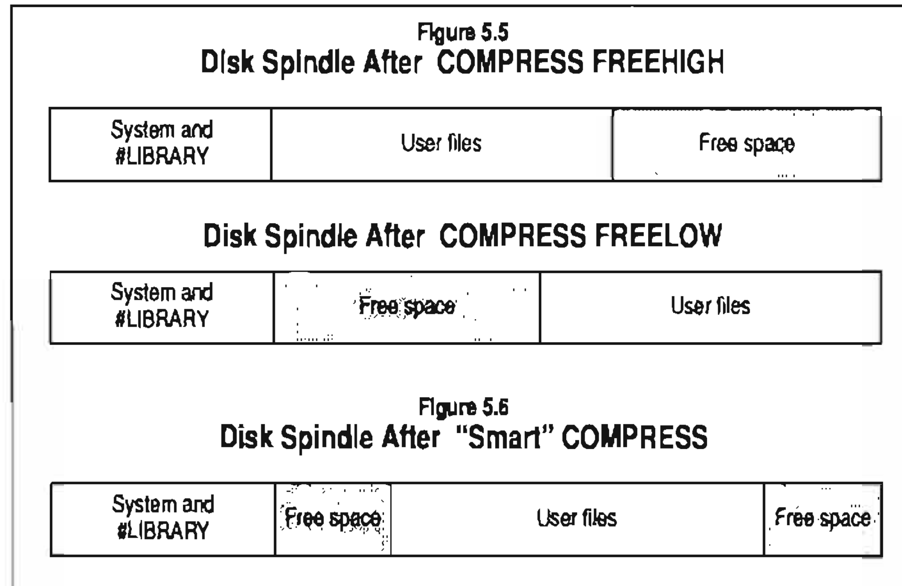
The Importance of Free Disk Space

In addition to addressing your system's memory capacity, you also need to consider additional disk storage. The only mechanical part of the S/36 — the disk drive — is the primary performance bottleneck, and the slowest component on the S/36. Each disk access requires 35 milliseconds on average. In that time, the CPU can perform up to 35,000 machine instructions. With abundant memory, you can eliminate much performance-robbing disk I/O. But no matter how much memory you have, you cannot eliminate *all* disk I/O. What can minimize the impact of disk I/O is plenty of disk space, which allows you to tune and manage your system effectively. Here's how system performance can be improved with abundant disk storage:

System Work Areas. If you have enough disk, you can provide large system work areas. The Task Work Area, for example, should be set to its maximum amount of 6,553 blocks to avoid Task Work Area extents. In addition, if your system uses external program calls heavily, having TWA maxed out to 6,553 blocks might not be enough — there should be enough disk space available so that your system can create contiguous Task Work Area extents (see Chapter 7). By allowing for a large spool file, free disk space also helps avoid costly spool file extents.

A Smarter Compress. Typically, use of the COMPRESS procedure gathers all free space within the user area of each spindle into one contiguous area at the beginning or end of the spindle. Figure 5.5 shows the effects of COMPRESS FREEHIGH and COMPRESS FREELOW to move all free disk space to ends of the disk spindle. However, if you have enough disk space available, a much better disk organization is shown in Figure 5.6. Here, free space has been collected into two groups and allocated on either end of the spindle. The space on the left, or "low," end of the spindle is targeted as the high-activity area of the spindle. Frequently used files will be extended or allocated here first, filling up this space as the day goes on. Disk utilization of this area of disk requires less disk arm movement and provides quicker access than the area at the high end of the spindle. Less frequently used files — those whose disk accesses happen less frequently — generally will go into the second free space.

The key to creating these two empty areas on disk is to use a large "placeholder" file in conjunction with two calls to the COMPRESS procedure (Figure 5.7). The A1 spindle is compressed first, moving all available user



space to the low end of the spindle (COMPRESS FREELOW). Before compressing the spindle a second time, a 3,000-block file is created as a placeholder. Then the second compress moves all available user space (less the size of the placeholder file) toward the high end of the spindle (COMPRESS FREEHIGH). After completing the second compress, you delete the placeholder file, leaving a 3,000-block area (7.6 MB) of "high-activity" disk space available near the low end of the spindle. The remainder of the disk space is available in a second area at the high end of the spindle. Figure 5.8 shows a variation of Figure 5.7, the procedure that produces the appropriate "holes" on a two-spindle system. Keep these free areas on disk available by frequently, nightly perhaps, performing this "smart compress."

Technical Note

The "smart compress" technique applies only to systems with a lot of free disk space. Given minimal free disk space, you could use a smaller "placeholder" file. But on a disk-constrained system, creating the two free areas of disk space per spindle increases the chance that there won't be enough contiguous free disk space to perform file extends.

Successful File Extends. When an extend-capable file fills, SSP needs a contiguous area of disk space large enough to hold the newly extended file

Figure 5.7
"Smart" COMPRESS for Spindle A1

```
IF DATAF1-PLACEHLD DELETE PLACEHLD,F1
COMPRESS A1,FREELOW
BLDFILE PLACEHLD,S,BLOCKS,3000,256,A1
COMPRESS A1,FREEHIGH
DELETE PLACEHLD,F1
```

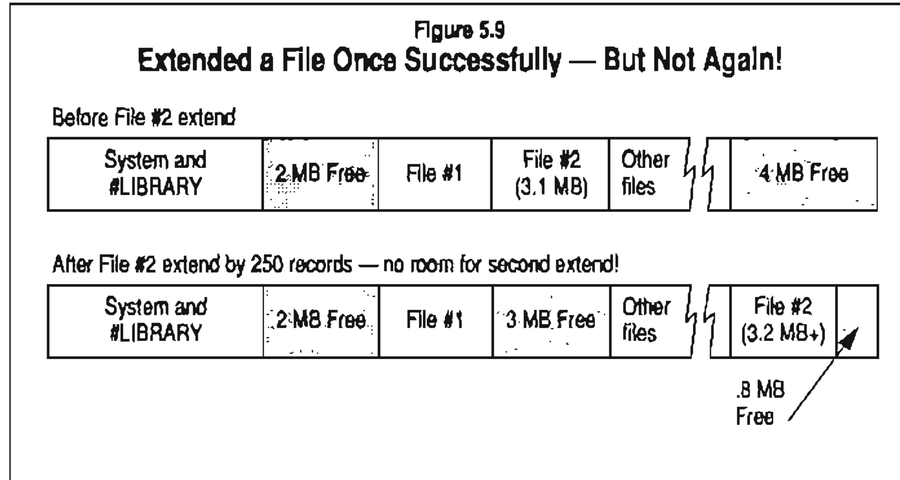
Figure 5.8
"Smart" COMPRESS for Spindles A1 and A2

```
IF DATAF1-PLACEHLD DELETE PLACEHLD,F1
COMPRESS A1,FREELOW
BLDFILE PLACEHLD,S,BLOCKS,3000,256,A1
COMPRESS A1,FREEHIGH
DELETE PLACEHLD,F1
*
COMPRESS A2,FREEHIGH
BLDFILE PLACEHLD,S,BLOCKS,3000,256,A2
COMPRESS A2,FREELOW
DELETE PLACEHLD,F1
```

(the size of the original file plus the extend value). If this contiguous disk space is not available, the file will not be extended; SSP issues a "file full" message and terminates your application — requiring an unpleasant application recovery process.

Files are extended by either the value specified with the EXTEND value in the // FILE statement or by the default EXTEND value established when the file was created. The EXTEND value in the OCL overrides the default EXTEND value. Specify EXTEND values carefully. If the EXTEND value is too small, file extends may occur more often than you want. Each time a file extends, an EDF-Wait occurs and all applications currently using the file are suspended until the extend operation finishes.

The extend operation will fail if there isn't enough contiguous free disk space to perform the extend, so don't make extend values arbitrarily large. Consider a sequential file with 12,000 256-byte records with an EXTEND value of 250 records. With 12,000 records, this file requires approximately 1,200 blocks of disk space. On a system with 1,500 blocks of contiguous disk space, only the first extend would succeed. A second extend would fail because not enough contiguous disk space is available (Figure 5.9). Note, though, that with the original 1,500-block area, there would have been room to extend the file by approximately 3,000 records.



Proper File Placement. Abundant disk space also lets you increase performance by properly placing files and libraries. To minimize disk seeks across one spindle, file location on one spindle is important; having available disk space will allow you to place the most-used files near the spindle's low end — the end most quickly accessed. Even more important than the ability to place files near the low end of a spindle is the ability to balance the placement of all files across all spindles to minimize disk arm movement. When creating a new file without a specific spindle specified, the system uses internal spindle-activity counters to place the file on the spindle with the lowest activity. In the case of a two-spindle system, the A1 spindle activity count will almost certainly be greater than the A2 spindle activity count because system files are located on A1. The result may be a disproportionate number of new files located on A2. An abundance of disk space lets you manage disk placement of new files, thereby leveling the load on each spindle. Your goal should be to have disk utilization for each spindle, as reported by SMF, within 10 percent to 15 percent of each other.

Additional Spindles. Although adding disk space is important for minimizing disk I/O and improving system performance, having additional disk spindles can really increase performance-tuning potential. The more disk arms you have to perform disk I/O, the more leverage you have to minimize disk arm motion. With multiple spindles, the disk arm is more likely to stay in position, making subsequent disk accesses faster. For example, if you have a two-spindle system, placing a parent file on one spindle and its alternate index on the other will yield faster disk throughput than if both files were on the same spindle, because the arm motion on two drives will be less than that on one.

Figure 5.10
Maximum Disk Capacity for S/36 Models

Model	5360	5362	5363	9402	5364
Disk capacity	1438 MB	523 MB	1256 MB	1256 MB	130 MB

Pricing Disk Drives

Generally, you'll want as much disk space in your S/36 as you can afford. Figure 5.10 shows an overview of maximum disk capacity values for each S/36 model (configuration details are discussed in Chapter 4). As they are with memory, IBM disk upgrade prices are high. Consider the cost of adding a second 200 MB drive to a Model B 5360 — a B23 to B24 upgrade. This upgrade is available from IBM at a current list price of $17,950. You can purchase refurbished 200 MB IBM drives from reputable third-party vendors for about $2,000 — a major savings for any S/36 shop. In most cases, IBM prices probably render an IBM disk upgrade a non-option. If yours is a true-blue, "No third party stuff here!" shop, take a few minutes to re-evaluate that position. Reputable third-party vendors offer an excellent way to increase the life of your S/36 (as a guide to choosing a reputable vendor, see "Avoid Third-Party Pitfalls," page 89).

As it is with memory upgrades, given proper notification and certification, IBM is obliged to honor your maintenance contract when you install third-party disk drives. Because only IBM disk drives are available for the S/36 — even if a third-party vendor installs your disk upgrade — your IBM maintenance contract, once your machine is recertified, will cover the third-party disk drives.

Eventually, most of the S/36s humming away happily right now will run out of resources and need to be replaced with newer, bigger, and better computers. Until then, upgrading your S/36 is a superb computing value and a great way to extend your machine's life and performance — thereby postponing your need to move to a new computer.

Although IBM seems to have priced itself out of the S/36 upgrade market, don't overlook third-party vendors' potential role in upgrading your S/36. Many third-party vendors can deliver the value and performance your S/36 needs.

Performance Tip

When upgrading a S/36, consider lower-capacity drives to achieve a higher spindle count. Four 200 MB drives give you more file placement options than two 200 MB drives and one 358 MB drive. Likewise, two 200 MB drives offer more options than one 358 MB drive.

Is the S/36 Worth Upgrading?

When the AS/400 appeared in 1988, S/36 users everywhere were at once overwhelmed with envy and "deja vu." Not long ago, it seemed, they had made passionate pitches to their CEOs about the need to replace aging S/34s with S/36s. With haunting memories, S/36 users pulled out the same arguments ("It's bigger, better, faster, easier, cheaper...") to convince their CEOs of the need for an AS/400. For many S/36 users, those arguments didn't sell as well the second time around: The S/36 wasn't that old, it still had lots of life and — probably most important to the CEO — it wasn't fully depreciated.

Today, many S/36 users are reconsidering the AS/400. Prices have come down and performance has gone up. IBM seems to be delivering on its claims that the AS/400 S/36 environment is viable in its own right — not just as a migration crutch. But at least three reasons still make the decision to migrate to an AS/400 a difficult one:

- AS/400 price/performance ratios, while better than ever, could still be better. Dollar for dollar, the lowly S/36 still delivers considerable bang for the buck — especially if your S/36 is nearly depreciated, is keeping your users happy, and is meeting your business needs.
- The AS/400 may be nearly halfway through its lifetime. Remember the anguish of buying a S/36 midway though its life cycle and seeing the AS/400 announcements a year or two later? While IBM insists the AS/400 will buck the typical midrange life cycle, many S/36 users are skeptical. There isn't a S/36 user in the world who doesn't shudder at the thought of buying an AS/400 today only to see the AS/500 (or whatever it will be called) announced in a year or two.
- The RS/6000 adds a new dimension of fear, uncertainty, and doubt to many S/36 migration plans. What is this new box with the funny sounding operating system? Wasn't Ming the Merciless from Unix? Not all that long ago most midrangers thought RS/6000 was a fuel additive, but now most of us can at least speak conversationally about the POSIX standard, interoperability, and the concept of an "open system."

This isn't to suggest that you should keep your head in the sand. The AS/400 is more than a supercharged S/36. Not only does the AS/400 do all the things the S/36 does, it also will do many things the S/36 cannot — and IBM adds additional capabilities almost monthly.

With an appreciation for the technological potential of the AS/400, however, the reality of the situtation for all but the maxed-out S/36 shops is that the longer migration can be deferred to another platform, the better. While you squeeze extra mileage out of the S/36, AS/400 price/performance ratios improve, the RS/6000 solidifies its position in the midrange market, and you have more time to identify the technological pivot points for your business through the end of the century. For the near future, many S/36 users will find upgrading machine resources to be a sound, logical investment.

Avoid Third-Party Pitfalls

However beneficial, using third-party parts to upgrade your S/36 isn't a task to be taken lightly. The addition of third-party products adds previously non-existent variables to the S/36 reliability equation. And you want to be certain that your S/36 won't take an unexpected afternoon off.

You can reduce the risk, and feel comfortable with the money saved, by carefully researching third-party vendors before signing on the dotted line. Before you buy from a third-party S/36 vendor:

- Ask members of your local user group which third-party vendors in your area are particularly good at providing customer service.
- Ask the Better Business Bureau if any complaints have been registered against a vendor.
- Ask for, and research, vendor-provided customer references.
- Ask about warranties, guarantees, and maintenance contracts.
- Ask about replacement parts, ongoing maintenance, and expected response time when problems occur.
- Ask for, and secure, your third-party vendor's help (before money exchanges hands) in getting your upgraded S/36 "recertified" for IBM maintenance.
- Ask your local IBM branch approximately how much you could expect to spend on machine recertification after installing third-party upgrades.

Chapter 6

Other Configuration Considerations

Beyond memory and disk space options, you may want to consider one of the following six ideas in your quest to keep your S/36 alive and useful:

- Upgrading your 5360 model B or C to a model D
- Upgrading your smaller S/36 to a 5360
- Setting up a dedicated program development machine
- Upgrading data communications to reduce CPU workloads
- Using Distributed Data Management (DDM) to expand disk capacity
- Turning to third-party service and support to reduce ongoing maintenance costs

Model D Upgrade

As you learned in the previous chapter, the more memory your S/36 has, the better. The 5362, 5363, and 9402 support up to 2 MB of memory and the 5364 supports up to 1 MB of memory — regardless of model designation. For the 5360, though, the maximum amount of memory depends upon the processor class.

IBM developed four different processor classes for the 5360 over its lifetime, each offering better performance or memory capacity than its predecessor. In IBM terminology, the four classes are called *stages*. You usually can determine the stage level of a given 5360 CPU by looking for a label next to the serial number tag on the lower right-hand corner of the raised CE panel cover. The label contains the word "stage" followed by a number: 1.0, 2.0, 2.1, or 3.0. Beware though: Some third-party installers fail to change this label after upgrading (or downgrading) a CPU. Any competent IBM or third-party customer engineer can verify the processor level by examining the installed processor boards.

The Stage 1.0 or 2.0 processor addresses up to 1 MB of memory, while the 2.1 processor addresses up to 1.75 MB. You'll find most 5360s in the field already are upgraded to the Stage 2.1 processor to accommodate the extra 0.75 MB. However, if your 5360 isn't upgraded, third-party vendors can install a Stage 2.1 upgrade very inexpensively. When coupled with the additional 0.75 MB of memory, the 2.1 upgrade is a cost-effective way to squeeze more performance out of a 5360 Model B. You can let SSP automatically take advantage of the additional memory, or allocate the memory to a specific purpose, such as disk cache or memory-resident screen formats. Although three-quarters of a megabyte doesn't seem like much — especially when

compared to memory capacities found in today's PCs — you're almost doubling the amount of memory SSP can devote to otherwise disk-intensive program management.

For even more memory in the 5360, though, consider upgrading to a Model D, which supports up to 8 MB — a 400 percent increase over the B and C models. This additional memory often is reason enough for many 5360 users to upgrade to a model D machine. In addition to addressing more memory, the model D's Main Storage Processor (MSP) and Control Storage Processor (CSP) both are faster than those in other models. The MSP is twice as fast; the CSP is 88 percent faster than Stage 1.0 CPUs, and 50 percent faster than Stage 2.0.

Performance Tip

If your S/36 consistently reports CSP or MSP utilizations greater than 60 percent, your machine is a good candidate for a Model D upgrade.

On 5360s without a Stage 3.0 processor, and for which System Measurement Facility (SMF) consistently reports CSP or MSP utilizations higher than 60 percent, the Model D upgrade greatly improves performance by reducing the CSP/MSP processing bottleneck.

While the fast Model D can eliminate the processor bottleneck, that won't do you much good if you immediately run into a second bottleneck in disk I/O. This is where the Model D's higher memory capacity helps. Remember that SSP is designed to operate even with small amounts of main memory — as little as 128 K. It accomplishes this feat by using disk space in place of needed memory, swapping application and system programs for one task onto disk when memory is needed to run some other task. The disk activity associated with swapping is a great performance robber, not only by delaying the execution of programs waiting for memory, but by also using up disk I/O capacity that would be better spent reading and writing your applications' data files. With 8 MB of memory at its disposal, SSP usually can keep everything it needs *memory resident*, reducing swapping to an insignificant amount. In fact, you may find swapping eliminated with only 3 or 4 MB of additional capacity; you can use any remaining memory to reduce application data file disk accesses by setting up a disk cache. Section V, "Performance Measurement and Tuning," provides you with the tools you need to benchmark your system and measure its resource utilization.

Performance Tip

Even if your CSP and MSP utilizations are low, you should consider upgrading to a Model D if SMF reports high swap rates. The vast increase in memory capacity of the Model D lets SSP keep frequently accessed application and system programs memory resident, reducing disk I/O and improving response time.

Model D upgrades are available both from IBM and third-party vendors. However, once again, IBM prices itself out of the ballpark. The IBM list price for a 5360 B23 to D24 upgrade, the result of which is a Model D 5360 with 2 MB of memory, is $9,910. On the street, from reliable third-party vendors, the same Model D upgrade costs about $3,500 installed. Third-party upgrades might use third-party memory, but the processors will be true-blue parts. Require a lifetime warranty on the memory, and buy from a reputable dealer, and your service will be equal to that of IBM's.

Upgrading to a 5360

A Model D upgrade should be high on 5360 users' wish lists, but what if you

have a smaller S/36? Other S/36 models cannot be upgraded to a full-fledged Model D like the 5360 can, which leaves you but two options: Either trade up to, or buy outright, a 5360 D model.

Even though some smaller S/36 models have processors approaching Model D performance, they have memory ceilings of 1 or 2 megabytes, substantially less than the 5360's 8 MB ceiling. Smaller machines also face smaller disk capacities, with the largest of the little CPUs supporting only 85 percent of the 5360's 1,438 MB DASD maximum. Communications adapters on low-end machines generally don't contain their own processors, adding to CSP and MSP workloads. The 5362 is the only small S/36 that supports the Multi-Line Communications Adapter (MLCA) — the others only support the Single-Line Communications Adapter (SLCA). None of the small S/36s support the Eight-Line Communications Adapter (ELCA) offered on the 5360.

In addition to memory, disk space, and communications options, the smaller S/36s also lack some of the additional processors of the 5360. The 5360, for example, has a Data Storage Controller (DSC) dedicated to ferrying data to and from tape and diskettes. Without a DSC, tape and diskette data transfer become the responsibility of the MSP and the CSP, resulting in tremendous response-time degradation during save and restore operations.

A used Model D 5360, with 7 MB of memory and 400 MB of DASD, costs about $8,000 — excluding the cost of licensing SSP. If you currently have a 5362, you can legally move your current licensed copy of SSP to the 5360 (as long as you stop using it on the 5362). You need 5360-specific microcode, and it should be provided by the dealer as part of the machine's maintenance package. However, the 5363 and 5364 versions of SSP are, by design, incompatible with the 5360; you must purchase a second SSP license for your new 5360, adding approximately $5,400 to the system cost.

The computing capabilities of the 5360 Model D over the smaller models are substantial — see Chapter 4 for maximum model configurations — and represent a good solution for 5362 shops hitting the resource ceiling but still wanting to stay with the S/36. For 5363/64 shops, the 5360 upgrade option is not quite as attractive because of extra SSP licensing fees. Still, for $15,000, the 5360 Model D offers lots of computing horsepower and few migration headaches; it could be a good solution for maxed-out 5363/64 shops needing more computer power, but not yet ready for the AS/400 or RS/6000.

One final note if you consider upgrading to a 5360 from one of the smaller models: Don't underestimate the hidden costs of providing a place to put the 5360. The 5360 is a big, hot, noisy beast requiring 220V power (in the U.S.). It uses as much as eight times the power of any smaller model, and requires lots of elbow room. It performs most reliably in an air-conditioned room. Most shops won't be able to simply roll in a 5360 and plug it in where the 5362 or 5363 used to sit. Carefully consider how much you will spend to

prepare a site for your 5360 and to pay ongoing higher electrical power costs.

A Dedicated Development Machine

If you actively develop and maintain the software in your shop, consider adding a second S/36 as a dedicated development machine. When a programmer (or worse, programmers) perform CPU-intensive source code editing, compiling, and testing all day long ("Just one more compile and surely this &^$%##*& program will work correctly!"), the S/36 runs as though molasses had been poured into the diskette slot. With a dedicated development machine, your programmers enjoy snappy response time throughout the edit-compile-test cycle and your users get back those stolen CPU disk, memory, and processor cycles.

A used 5362, the most convenient upgrade option because it uses an 8-inch diskette for data exchange with the 5360, costs approximately $1,450 with 2 MB of memory and 120 MB of disk space. However, you must also license a second copy of SSP from IBM at a cost of $5,400 (which includes the SSP and the necessary programming utilities). This solution is costly; but if your programmers constantly bring your production S/36 to its knees, it certainly is an option to consider.

SSP licensing costs are much lower on the 5363, and the overall package is much more compact than a 5362. However, the lack of an 8-inch diskette drive rules out diskette data transfers between machines. A 5363, with 2 MB of memory and 120 MB of disk space, with SSP and utilities, costs about $4,000 used. Except for the data-transfer wrinkle, the 5363 makes a good dedicated developer's box. To solve the data-transfer problem, consider using a cartridge tape drive, PC support to transfer files from machine to machine via PC diskette, or communications features such as Distributed Data Management (DDM) (see Section IV for more on DDM), or the file transfer subroutines (FTS) included as a free component of the S/36's base communications feature.

An optional, but extremely handy, software add-on to consider in a two-machine environment is IBM's Display Station Pass Through (DSPT) feature. DSPT lets you sign on to a remote system using the 5250 terminals attached to the local system. With DSPT, your programmers can sign on to the production machine without leaving their development machine terminals. And because DSPT is bidirectional, you also could sign on to the development machine from any production system terminal (provided you have the proper security clearances). Programmers can use DSPT to conveniently test production installations, or to access software development tools from the production system.

If you plan to use communications, keep in mind that you must have compatible communications interfaces (X.21, X.25, EIA, DDSA, or V.35) on both CPUs (see Chapter 4, "S/36 Models and Configurations," for details on communications hardware), and that the kind of connection you set up may

limit line speed. For example, using the EIA interface and a synchronous EIA modem eliminator (about $200 new) to directly connect two systems without phone lines limits line speed to 9,600 bps on the 5362 and 19,200 bps on the 5363 and AS/Entry (9402 Y10). If both S/36s have V.35 interface, however, you can use a synchronous V.35 modem eliminator (about $400 new) to run at speeds as high as 64,000 bps. Although the V.35 modem eliminator costs twice as much as the EIA version, you get four times the performance.

Here is another thought for providing your programmers a dedicated programming box (this might smack as heresy to some S/36 users): Buy or lease your S/36 programmers a small, used AS/400. A model E02, with 8 MB of memory, 1 GB of disk space, a tape drive, a workstation controller, OS/400 and utilities, can be had for less than $12,000. Add a couple of used terminals and you have a terrific two-programmer S/36 development box. If you suspect that you will migrate to the AS/400 in the next year or two, providing your programmers now with their own development AS/400 might be a wise investment. Your S/36 will run faster minus program development and, perhaps more importantly, your programmers can start getting their feet wet with the AS/400. They can use the AS/400's S/36 environment for S/36 development and, in their spare time (what little programmers have — but they always dig some up with a new computer around), they can start dabbling with the AS/400 and its myriad array of bells and whistles. When migration time finally arrives, you'll have a staff of programmers to whom the AS/400 is no longer a stranger.

The 5364, the smallest S/36, could also be considered a programmer's box; but again, without an 8-inch diskette, it offers cumbersome data transfer options. The 5364, which requires a PC as a console and to provide a soft control panel (an old PC or XT clone works just fine), is available at give-away prices from third-party vendors. You'll probably be able to find a 5364 in the $200 to $500 range (like the 5363, the SSP and utilities are bundled with the 5364). Add the cost of an old PC or XT (which you can probably get just for the asking in the right places), and you'll have a S/36 for less than $1,000. It won't be much of one, but it will be enough to use for minor program maintenance. Keep in mind, however, the extremely slow performance of this machine — it is the slowest S/36 model, bar none. You can't, for example, effectively perform program development chores while simultaneously transferring files over a communications line.

Communications Upgrade

S/36 communications can be a notorious resource hog. If you use communications, don't overlook the burden it imposes on S/36 performance. The Single-Line Communications Adapter (SLCA) uses the S/36's CSP to perform such data-link chores as polling and protocol management; the communications

Figure 6.1

Recommended Communications Adapters

Deciding factors:

I. How you use communications
   A. Occasional dial-up, 1 or 2 lines
   B. 1 or 2 lines connected all day
   C. More than 2 lines connected all day

II. The level of transaction activity
   1. Occasional inquiry and/or light update
   2. Regular inquiry and/or update
   3. Intensive inquiry and/or update

Recommendations:

For A or B1, a SLCA is adequate

For all other possible uses and levels of activity, choose either a MLCA or ELCA

processing required by the CSP can add up to 50 percent to its processing load (reflected in CSP utilization on an SMF report). A S/36 already burdened with application programs might be pushed into uselessness by the extra communications processing overhead.

The Multi-Line Communications Adapter (MLCA) and the Eight-Line Communications Adapter (ELCA) each contain a dedicated communications processor that relieves the CSP of low-level data-link overhead. Installing an MLCA or ELCA is actually like installing a dedicated CSP for communications activity. Although the names of these two adapters imply that you must install multiple lines, you actually can run either with just one line. In fact, most S/36 models can support higher data rates when using just one line at a time. During communications sessions with an MLCA or an ELCA, the S/36's primary CSP is needed only when received data is moved into program buffers. Figure 6.1 shows recommended adapters for specific communications activity. Third-party communications adapters are readily available. Expect to pay approximately $850 to $1,000 for an MLCA and $1,500 to $2,000 for the ELCA from third-party vendors.

Another consideration when upgrading communications facilities is the interface — the electrical connection to your communications network. The S/36 supports five different interfaces — EIA/CCITT, DDSA, X.21, X.25 and V.35 — and three different kinds of networks — nonswitched, switched, and packet. For any two interconnected systems, the interfaces must match. Each interface has its own type of cabling, and is designed for particular kinds of networks.

Following are the services provided by each kind of network:

Nonswitched network: A direct, continuous connection between two or more machines. A nonswitched network can be as simple as two machines in the same room connected via modem eliminator, or as complex as several machines across the country connected by a single leased telephone line. The first configuration is called *point-to-point*, and the second is called *multipoint* or *multidrop*. The line speed supported by a given nonswitched network depends on the noise characteristics. A pair of 25-foot shielded cables with a modem eliminator has very good noise immunity, and thus accommodates the highest line speeds. A short-distance, leased-line connection (e.g., between two offices in the same telephone exchange), using short-haul modems, has some noise, but can still usually support speeds as high as 38,400 bps. A long-distance leased line (e.g., between two cities), using long-haul, leased-line modems, has considerable noise and won't run faster than 19,200 bps without special routing agreements between all the telephone companies involved. Leased connections are also available from Digital Data Service (DDS) providers; these connect systems directly to a separate digital network rather than through modems over telephone wires. Finally, a custom analog service available from some telephone companies, called 60-108 kHz group band circuits, supports speeds as high as 48,000 bps.

Switched network: Also called dial-up, switched networks use the Public Switched Telephone Network (PSTN) — telephone companies' switching system — to connect two computers. Switched networks offer only point-to-point links: One system dials the other to establish a connection. The telephone switching system determines the routing for the connection, so the quality of the connection can vary considerably. Switched connections usually only support speeds of 9,600 bps. Note that some modems now claim higher speeds, but they usually accomplish this by compressing data in the sending modem, transmitting it at 9,600 bps, and decompressing it again in the receiving modem. Depending on the nature of your data, you may not see much actual increase in throughput beyond 9,600 bps. Compression algorithms in such modems typically reduce data volume by 50 percent for plain ASCII text; the reduction rate is less for binary data.

Packet-switched network: A packet-switched network is actually a computer system (the packet switch) that receives data in variable-sized blocks called packets, with each block containing the address of a destination system. The packet switch sorts packets and routes them to their proper destination system, which also must be connected to a packet switch in the same network. Commercial packet-switching networks such as Telenet and Tymnet are called Public Switched Data Networks (PSDNs). The PSDN lets you use one physical communication line to communicate with many remote locations simultaneously, by establishing separate *virtual circuits* to each partner with

which you wish to communicate. You can connect to the packet-switching system itself via dial-up connection or leased line, but you must have a packet assembler/disassembler (PAD), either onboard the S/36 as X.25 software, or outboard in an external X.25 controller.

Each interface provides different kinds of network options:

EIA/CITT: Also called RS-232, the electrical connection is a shielded 25-conductor cable using DB-25 connectors at each end. This is the most common kind of interface, connecting directly to most modems and modem eliminators. For the S/36, it also has the lowest line speed ceilings for most systems. With EIA/CCITT, you can connect to leased and switched networks, but not packet-switched networks without special X.25 external controllers. The EIA interface also supports asynchronous and Binary Synchronous (BSC) connections. Asynch/bisynch software support is built into SSP's free base communications feature.

DDSA: The Digital Data Service Adapter connects to commercial digital data services (DDS). The electrical connection is by special cable to a DDS-provided Digital Service Unit (DSU). The commercial DDS provider guarantees a certain line speed, and most support the highest available line speeds for any S/36 (64,000 bps).

X.21: Electrically identical to EIA, X.21 provides for higher line speeds when using S/36 MLCA or SLCA (19,200 vs. 9,600).

X.25: Electrically identical to EIA, X.25 also performs packed assembly/disassembly (PAD) services for packet-switched networks. The S/36 supports an integrated PAD in software with its X.25 communications feature, or you can use an external X.25 controller.

V.35: The electrical interface is a special 34-pin shielded cable and connectors. Used on special wideband leased telephone lines (called 60-108 kHz group band circuits), V.35 supports line speeds as high as 48,000 bps. With data compression synchronous modems (or a V.35 modem eliminator for local connections) you can achieve the highest S/36 data rates of 64,000 bps on a single line.

When choosing the communications adapter, interface, and network for your system, keep in mind the need to maintain compatibility at *each node* in your network. To this end, you're better off using equipment from a single manufacturer. For example, ensuring that the modems at each node are of the same brand and model eliminates one source of incompatibility that could stifle your efforts to get a network up and running.

Distributed Data Management

If your disk storage requirements go far beyond what a single S/36 can support, you might consider storing the data on a remote S/36, or even on a remote AS/400, and accessing the data at the record level using IBM's Distributed Data

Management (DDM) facility. With DDM, you could double the disk capacity of your S/36 installation by simply adding another S/36 as a DDM server; or you can obtain practically unlimited capacity by using an AS/400 or PC as the DDM server. Everything costs something, however, and DDM is no exception. The price you pay for this expansion path is in performance: Accessing data stored remotely is slower than accessing locally stored data. How much slower depends on the speed of the communication connection between your S/36 and the server machine. If you primarily need to keep massive quantities of historical data online, DDM-based files offer much faster access than you can get restoring tape or diskette archives. DDM also has some limitations on the operations you can perform on remote files. You can almost do anything with a DDM-based file that you can with a local file. You have to decide if the undoable makes DDM unworkable for your installation. Because DDM runs under APPC (advanced program-to-program communications) and APPN (advanced program-to-program networking), it offers record-level access to multiple systems simultaneously, if your network supports it. Just to expand local disk capacity, however, you need only a simple point-to-point connection.

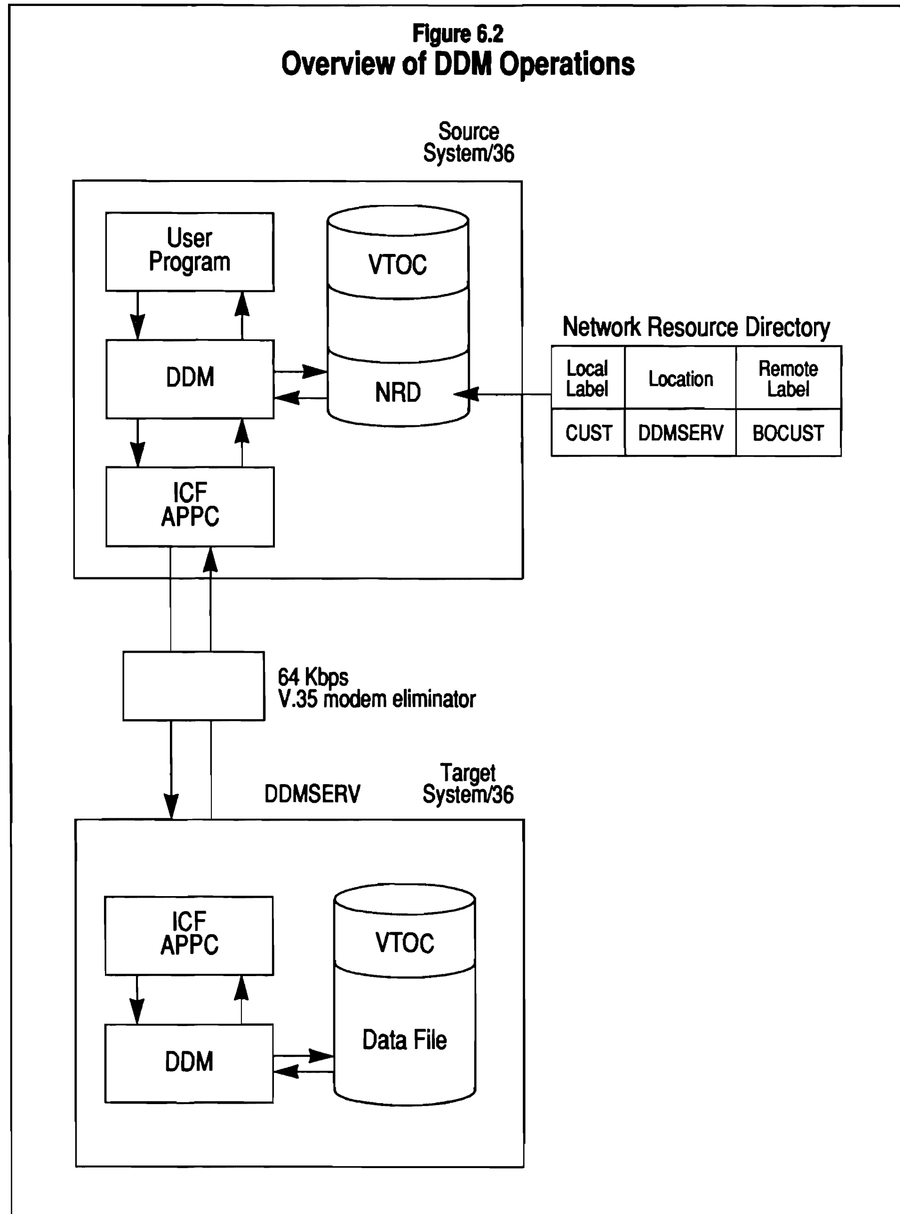
DDM supports the following functions:

- Record-level access by RPG, COBOL, BASIC, FORTRAN and Assembly Language programs
- DFU, WSU, #GSORT access to data in remote files
- MSRJE for S/370 connectivity
- BLDFILE and BLDINDEX on the remote system
- DELETE and RENAME on the remote system
- CATALOG to list specific remote files
- LISTDATA and LISTFILE to display contents of remote files
- SAVE and RESTORE of remote files on local save/restore devices
- COBOL runtime sorting (SORT statement) of remote files

All of the forgoing capabilities apply only to data files. DDM doesn't support the following kinds of access:

- Access to libraries or folders on the remote system
- Work files for #GSORT and other utilities (e.g., RPGC)
- Date-differentiated files
- Query/36 access to remote files
- DisplayWrite/36 access to remote files or documents
- Streaming tape as a save/restore device
- SAVE ALL or RESTORE ALL for the remote system

Figure 6.2 presents an overview of how DDM works in a local disk-expansion scenario. In DDM terminology, the system making DDM requests is



called the *source* machine, and the machine containing remotely stored data is the *target* machine. The example shows the two machines connected via a V.35 64,000 bps nonswitched local network (modem eliminator). A special file

on the source system, called the Network Resource Directory (NRD), contains a list of files stored on the target system. You use the EDITNRD procedure to maintain this file, which lists the name of each file as it is used on the source system, the name of the target system that contains the actual data, and the name of the file as it is stored on the target system. Whenever a program or SSP utility tries to access a file that can't be found on the source system, DDM looks for an entry for the file in the NRD. If one is found, DDM establishes and maintains a connection to the target system, passing all data requests to the target system to be satisfied. Except for speed, access to data on the target system is completely transparent to programs on the source system. No changes are necessary to OCL procedures or application source code.

As mentioned earlier, file access via DDM is slower than via local disk data management. As a point of reference, consider that native, unblocked, random indexed file reads on the S/36 occur at a rate of about 12 per second. With a 64,000 bps connection, the fastest DDM rate is two per second. You can improve this somewhat by using blocking, but DDM only performs blocking for unshared or read-only files. For shared or update-capable files, DDM transfers one record at a time across the network. You might also improve performance somewhat by using a Token-Ring connection. However, keep in mind that only a small fraction of the apparent Token-Ring bandwidth is effectively available for transaction throughput. Rather than the 250-fold speed improvement you might expect on a 16 Mbps (megabit per second) Token-Ring over a 64,000 bps V.35 connection, you'll probably see more like a 10-fold improvement. Significantly faster, to be sure, but perhaps not enough faster to justify the extra hardware costs in your situation.

Here are the general steps to follow for setting up DDM:

- First, determine the kind of communication connection you'll use. DDM works with everything from a local non-switched network using a synchronous modem eliminator running at 9,600 bps to a cross-country V.35 network using wideband analog modems. You can also run DDM across a Token-Ring network. As with everything else in life, faster costs more. You might want to start out with an inexpensive EIA connection running at 9,600 or 19,200 bps and see if that provides adequate performance. You can always buy more equipment later.

- Second, you'll need DDM software installed on your local S/36 and whatever remote machine you choose as a DDM server. For the 5360 and 5362, order IBM program number 5727-SS1; for the 5363 and 5364, program number 5727-SS6. IBM's *S/36 Distributed Data Management Guide* (publication number SC21-8011) provides detailed information about setting up and using DDM. You'll also need a companion manual for whatever system is acting as DDM server (for the

AS/400, *Communications: Distributed Services Network Administrators Guide*, SC21-9588; for the PC, *Using Distributed Data Management for the IBM Personal Computer*, SC21-9643). The IBM S/38 and S/370 also support DDM connections with the S/36, but neither is a viable disk expansion alternative.

- Third, configure an APPC PEER subsystem under ICF. IBM's *ICF: Base Subsystems Reference* (publication #SC21-9530) explains this straightforward process.
- Fourth, after installing the equipment necessary for your chosen network connection, enable the PEER subsystem to establish a live connection between the two systems.
- Select the files you want to reside on the target system and create entries for them in the NRD using IBM's EDITNRD procedure.

DDM is now ready to run. You can move files to the target system using IBM's COPYDATA procedure. As long as the copy-to filename is listed as a local name in the NRD, DDM will automatically create the file on the target system and move the data records there. For alternate indices, you must build the indexes rather than copy them. Just run the BLDINDEX command on the source system, specifying the parent file resident on the target system and an alternate index file name that exists in the NRD. DDM automatically builds the index on the remote system without transferring any additional data.

Performance Tip

To get the best DDM performance: Do not run batch applications against DDM files concurrently with interactive jobs. Configure your APPC subsystem under an APPN network, even for point-to-point, because APPN uses a faster protocol-exchange algorithm. Use DBLOCK for random-access, read-only files where some requests are also sequential, but do not block larger than 4 K, and do not block purely random retrievals. Use JOB-YES to minimize remote VTOC requests. Ensure both systems are at SSP Release 5.0 or higher, as earlier releases use a slower DDM protocol. Avoid heavy local usage on the target system (the DDM server). Because DDM has priority permanently set to LOW, local jobs will overwhelm DDM support. Use REORG-YES to improve remote build time for COPYDATA.

Maintenance

In addition to using S/36 third-party vendors for hardware upgrades, don't overlook using them to also provide S/36 hardware maintenance. There is an ample supply of S/36 parts available, and many third-party S/36 maintenance providers offer good value for your maintenance dollar. The money you save on S/36 maintenance can be used to help finance memory and DASD upgrades.

Generally, two types of S/36 maintenance agreements are available. The most popular maintenance agreement type is a monthly maintenance contract under which you pay a fixed monthly fee, regardless of service and parts needed. The other is a time-and-materials maintenance contract under which you pay a varying fee for parts and labor only when problems occur. With the declining street value of used S/36s, it might be worthwhile to keep a spare S/36 in the closet and maintain your primary S/36 with a parts-and-labor maintenance contract. Maintenance of this sort is generally not for faint-of-heart, mission-critical shops.

Because of the potential high cost of S/36 replacement parts, most S/36 users shy away from a time-and-materials-type contract and opt for the monthly maintenance agreement. For most shops, this type of agreement

offers the best protection. Monthly maintenance contracts are available for protection during normal business hours; or, by paying a little extra, around-the-clock coverage is also available.

IBM and third-party providers offer maintenance contracts of both types. However, both often price time-and-materials contracts very high to make the monthly contracts more appealing. Realistically, your maintenance contract choices are generally reduced to using IBM or third-party providers for monthly maintenance contracts. If you've been a true-blue diehard in the past, turning your nose up at non-IBM parts and service, think again. Many third-party maintenance providers are actually more attuned to their customers than IBM is, and it's surprising how many ex-IBMers are employed by third-party maintenance providers. Because many of these vendors make their livings solely, or at least primarily, by providing hardware maintenance, they have to be very good at it. Third-party maintenance is almost always less than IBM maintenance, with third-party contracts offering anywhere from a 20 percent to 50 percent savings over IBM maintenance prices. The abundance of available S/36 parts, third-party maintenance vendors' willingness to "bend over backwards" for your business, and IBM's increasing withdrawal from the S/36 all add up to make third-party maintenance a workable and practical option.

Section III

External Program Calls

"Everything new meets with resistance."

—*Russian Proverb*

If there is a silver bullet for S/36 application programming, it is external program calls (EPC). With EPC you can achieve subsecond response time in your existing applications, circumvent the 64 K program size barrier, and bring the powers of modular design and coding to bear on application programming.

Unfortunately, although the S/36 operating system always has had built-in support for EPC, the capability wasn't available for RPG or COBOL application programmers until well into the S/36's lifetime. An IBM COBOL PRPQ (custom programming feature) and third-party RPG compiler products were the first tools to make EPCs available, and thousands of sites took advantage of these tools. Now IBM offers a limited form of RPG EPC for free to all SSP licensees as part of the S/36 Value Added Software Package (VASP). The result is that you're in the enviable position of having several EPC implementations from which to choose. Whatever your final decision, how you proceed with EPC will have an impact on your short-term ability to take advantage of the feature and your long-term ability to migrate to follow-on platforms.

Chapter 7 in this section tells you how EPC works and how to use it, Chapter 8 gives you the rundown on the various EPC products available, and Chapter 9 presents a tutorial on modular application design. With this material you can load your programming gun and shoot down old S/36 performance limitations.

Chapter 7

How External Program Calls Work

Suppose you had an opportunity to give your users subsecond response time when they switch between applications. Suppose you could add new capabilities to your applications with ease, chopping through your programming backlog in half the time you're taking now. And suppose you could throw off such S/36 restrictions as the 64 K region size and the 15 disk-file limit. Now suppose you could do all these things with no additional hardware resources.

If you would jump at such a chance, you'll want to jump right into an oft-ignored S/36 feature: the external program call (EPC). EPC lets you invoke other programs from within your application without using OCL and provides program-to-program communications without using the LDA. Called programs can, in turn, call other programs without limit. And a called program can return to its caller without going to end-of-job, so subsequent calls don't require the overhead of program initiation.

This amazing array of capabilities lets you write efficient, well-structured, modular programs that are easier to maintain and enhance than huge monolithic programs. Modular programming also clarifies application design — "hiding" the way your program implements a solution (i.e., what the textbooks call "process abstraction," or deferring coding details) to let you focus your attention on the program's logical structure. And modular programs perform better, too, because EPC circumvents the 64 K region limitation and RPG's maximum of 15 disk files — even while adding more function. Learning how to get EPC capability, how to code EPC statements in RPG, and how to incorporate EPC features in your existing applications will get you on the path to improved design, faster development, and better performance.

IBM's Little Secret

EPC capabilities have been built into the SSP from the beginning. In fact, IBM uses the feature in some of its own program products, such as SDA, DisplayWrite, Query, and ODF. About five years ago, third-party programmers discovered IBM's little secret and started selling products that made EPC accessible to RPG programs. Two third-party vendors — Amalgamated Software of North America and BPS Information Systems — provide RPG interfaces to EPC by harnessing this S/36 under-the-covers capability, not by employing any unreliable computing voodoo. In the fall of 1990, IBM finally released its own EPC add-on to RPG, which now is built into SSP release 6.0.

Although each vendor has implemented a somewhat different set of

features and quirks (see Chapter 8, "A Comparison of EPC Vendor Offerings"), all three products follow the basic RPG EPC syntax and operation rules IBM established for the S/38 and continued on the AS/400. This means your EPC programs are not only compatible with other S/36 EPC products, but also upwardly compatible with the AS/400, which protects your programming investment.

Third-party EPC products do have one drawback: They cost money. Adding non-IBM EPC capability to your S/36 will cost you anywhere from $950 to $2,250; however, third-party EPCs provide a number of improvements over IBM's newer, but weaker, implementation. Read on and you'll find that using even extra-cost third-party EPCs quickly provides a return on your investment.

128 MB of Memory, Virtually!

Using EPCs requires memory. Fortunately, like the S/38 and the AS/400, the S/36 is a virtual memory (VM) machine. That is, when a task requires more real memory than is currently available, SSP pages less-recently-used memory "pages" to disk to free up the required real memory (see Chapter 2, "S/36 Memory Management," for details about how virtual memory operates). The Task Work Area (TWA), also called the #SYSTASK file, is where paged-out memory is held until needed again. The SSP uses the TWA as a "backing store," quickly shuffling data between real and virtual memory as needed. The size of the TWA thus determines the maximum amount of VM.

S/36 *real* memory capacities range from a maximum of 2 MB on 5363s to a maximum of 8 MB on the 5360 model D (see Chapter 4, "S/36 Models and Configurations," for memory limits of various models). Logically, however, the virtual memory ceiling on the S/36 is limited only by the maximum size of the TWA — which can be as much as 128 MB. This means that you can have up to 128 MB of programs running *at one time* on a S/36 with one megabyte or less real memory!

Technical Note

The maximum size you can configure for the TWA is 6,553 blocks, which yields about 16 MB of virtual memory. But this is only the "initial" size of the TWA. The first time the TWA fills up (due to program requests for more virtual memory), SSP automatically extends it by 400 blocks; the second time it fills, SSP extends the TWA again, but this time by 800 blocks; the third time, by 1600 blocks. The process repeats, with each extension doubling in size up to 6,553 blocks. As many as 16 extensions can occur; depending on available disk space, SSP can expand the TWA to a maximum of about 53,000 blocks (128 MB).

Although the S/36 will let you use a great deal of VM with only a small amount of working real memory, performance may suffer. Increasing the quantity of real memory lets the S/36 keep more programs "paged in," reducing swapping (disk activity) and thus improving performance. While a program CALL requiring VM paging is fast, it isn't as fast as a real-memory CALL. For example, a call to a program already resident in real memory can take as little as 4 milliseconds (depending on whose EPC product you're using), while the same call to a program paged out to the TWA would take at least 35 milliseconds. Even with VM paging, EPCs are fast — but with enough real memory, they are very fast! And hey, a millisecond here, a millisecond there, pretty soon you're talking real time! So to get the most out of EPC, consider adding as much real memory as your machine supports (see Chapter 5, "The Importance of Memory and Disk Space").

Performance Tip

Make sure the SSP has as much VM backing store (TWA) as possible. Use CNFIGSSP to set your TWA to 6,553 blocks. Even if you aren't using all of the TWA space, the extra room helps reduce TWA fragmentation, which can slow performance. This small trade-off in disk space will pay off handsomely as you learn to make EPC work for you.

Primitive Modules

Before the days of EPC availability on the S/36, few techniques were available to write efficient, modular programs. We attacked most large application requirements with large, monolithic programs — maxed-out, 64 K, 15-file, do-everything, interactive monsters. The kind that took half a box of green bar to print and always had something wrong somewhere. A monster program like this generally performed fast (at least it started out fast) and was easy to write (at least it started out easy). But its logic was hard to comprehend, and it was difficult to maintain, slow to initiate, and nearly impossible to enhance without introducing bugs in unrelated parts of the program.

About the time we were all realizing the hopelessness of these monolithic slabs of code, a few enterprising programmers introduced us to the technique of emulating EPCs by chaining programs together via the LDA. The problem with this technique is that it is slow. Each emulated call requires the complete initiation of the program — a slowwww process. It's amazing how long two seconds can seem in an interactive environment. To combat the slow initiation time between chained programs, a better technique was introduced that chained single-user MRT-NEPs (Multiple Requester Terminal-Never Ending Programs). This technique eliminates job-initiation time when chaining from program to program. The technique, however, has serious limitations: It doesn't provide communications between programs with formal parameters; it requires lots of tricky, less-than-intuitive OCL and a little hoop-jumping to end the programs; and (because MRT-NEPs are executed in a single-threaded fashion) it can cause performance bottlenecks.

Exploiting the /COPY feature of RPG's autoreport facility is one other way monolithic RPG programs can be broken into separate modules. This method is faster than the LDA program-chaining technique, but it just hides the fact that you are writing a monolithic program. None of the included mod-

ules has local variables or indicators, and communications between these modules is the same as it is with any normal RPG subroutine — through global variables known throughout the program. The /COPY method works great for small, simple tasks; but for any large task or function, it offers none of the advantages of real EPCs.

With the EPC design alternative, multiple programs exist in a single task. This approach offers quick program-transfer time; almost unlimited formal parameter passing; private fields, indicators, and I/O areas for each program; a way to circumvent the 64 K program size and file limits per program; and a way to write code upwardly compatible with the S/38 and the AS/400.

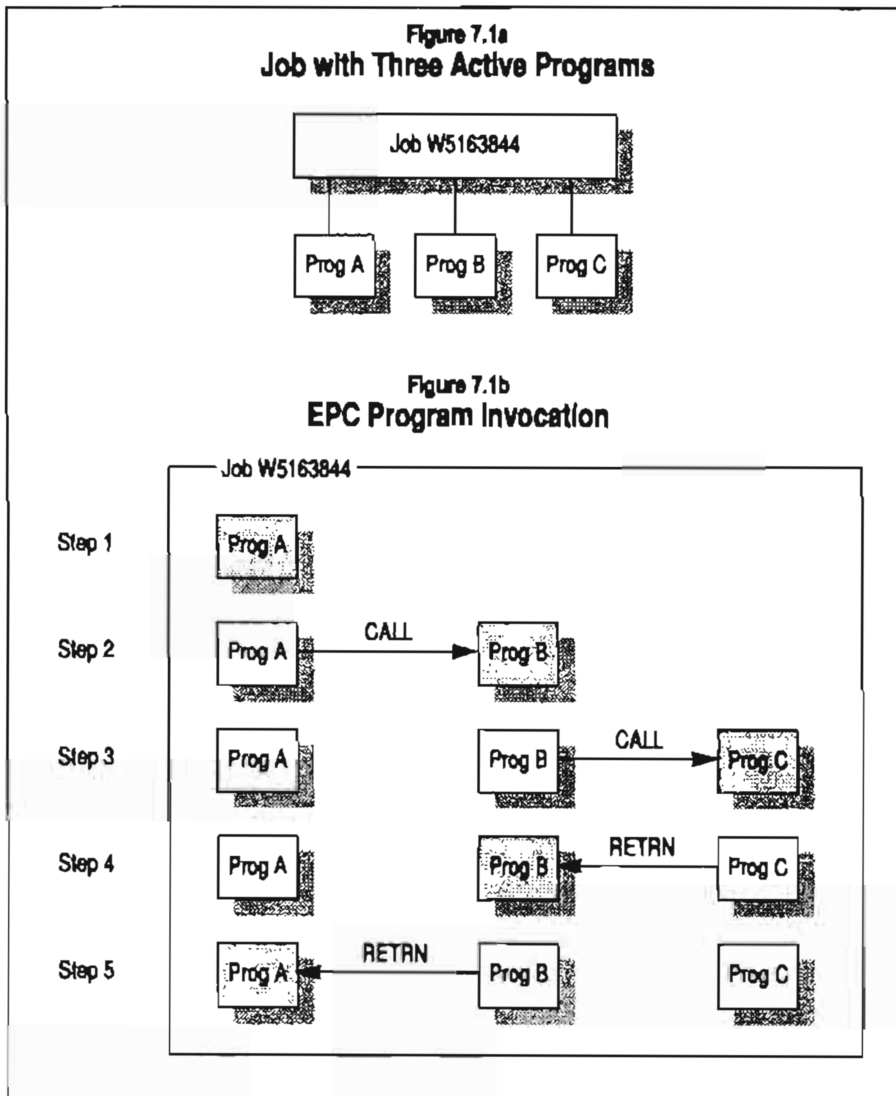
EPCs in Action

EPC uses some special terminology to describe the relationships between executing modules (programs); understanding this terminology can help you see the power of EPCs as modular programming tools.

Normally, a S/36 RPG task consists of just one RPG program. With EPCs, however, one task can consist of two, three, or 100 RPG programs. The number of programs is limited only by the available VM. Each program in a task has its own 64 K region. One program, called the *main program*, is the controlling program for the entire task; it's the program started by the // LOAD OCL statement. The main program can call, or *invoke*, other programs, called *subprograms*. Subprograms can also call other subprograms (again, with no specific limit on the depth of calls), but the main program is special. When a subprogram goes to end-of-job (EOJ), it does not affect other subprograms; when the main program goes to EOJ, all subprograms are automatically terminated. Figure 7.1a shows a task with three active programs attached (each with its own 64 K region and a private copy of variables, indicators, file buffers, record positions, and so forth).

Both the main program and subprograms have their own private data: variables, indicators, open files, record positions, and I/O buffers. No program can directly change another program's private data. Thus setting on indicator 11 in one program has no effect on indicator 11 in any other program in the task. Similarly, reading a particular record in a file doesn't change the current record position or buffer contents for any other program using the same file. This concept of private data — also called *information hiding* — is central to modular design, because it lets you ensure that modules only interact through well-defined interfaces. With RPG EPC, modules interface by exchanging *parameters*. But to understand how the parameter interface works, you must first understand the *program invocation* process.

Figure 7.1b shows a typical EPC scenario with a main program, PROGA, and two subprograms, PROGB and PROGC. Initially (Step 1), main program PROGA runs by itself, and is considered the *currently executing*



program. Then PROGA CALLs (invokes) subprogram PROGB (Step 2), causing SSP to *activate* PROGB by allocating memory, loading the program into that memory, opening files, and initializing variables. PROGB then becomes the currently executing program, while PROGA is *active-but-suspended*. The same sequence repeats when PROGB invokes PROGC via a CALL (Step 3): PROGB becomes active-but-suspended, and PROGC is activated and becomes the currently executing program. At this point, two programs are

invoked (PROGB and PROGC) and three programs are active (PROGA, PROGB, and PROGC); but only one program is currently executing. In any task, only one program at a time is ever the currently executing program — all others are active-but-suspended.

Next, PROGC finishes its work and RETRNs to PROGB (Step 4); PROGB resumes execution with the statement following the CALL. At this point, PROGC is no longer invoked, and PROGB has regained its status as currently executing program. However, PROGC doesn't go to end of job and disappear; instead, it becomes uninvoked and active but suspended — parked in VM awaiting another call. All of its private data remains intact, usable on the next call.

Finally, PROGB RETRNs to PROGA (Step 5), leaving PROGB uninvoked and active-but-suspended. In this last state of affairs, PROGA is the currently executing program, while subprograms PROGB and PROGC are waiting for another call.

Technical Note

One aspect of S/36 EPCs that isn't obvious from the example is how massive use of VM impacts interactive application performance. It turns out that having a large number of active subprograms doesn't necessarily slow response time. In fact, designed correctly, modular applications have significantly better response time than do applications put together with other techniques. That's because only one program in any given task is the currently executing program. Programs that are active-but-suspended eventually get paged out of real memory, where the only resource they consume is TWA space. Paging such programs back into real memory when required takes only a few milliseconds — an imperceptible amount of time for interactive modules.

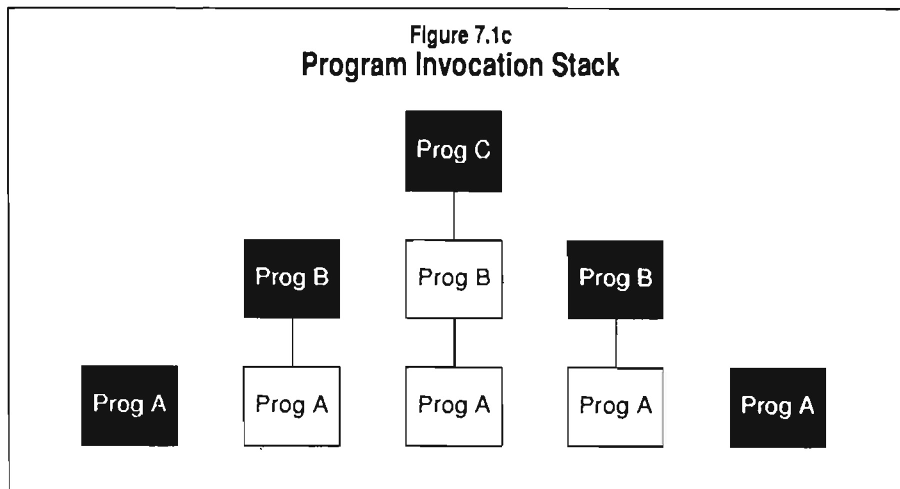
You can make some interesting observations about the three programs at this point. First, subsequent calls to either PROGB or PROGC will be faster than the initial call because the programs are already activated. Activation of a subprogram takes from several seconds to a minute or more, while calling an activated program takes only a few milliseconds. Second, since neither PROGB or PROGC are currently invoked, they could be called later in a different order (e.g. PROGA calls PROGC which calls PROGB). As long as an active subprogram is uninvoked, any other program in the task can call it. Third, because each active subprogram preserves its private data across invocations, you can use subprograms as storage areas, expanding the amount of memory available to a task. You could, for example, use a subprogram to provide access to a large array, as long as the entire array fit inside the subprogram's

64 K region. Keep these three observations in mind — they'll come in handy later when you begin to decompose a problem for implementation as a set of modular programs (see Chapter 9, "Implementing Modular RPG Applications," for details on modular decomposition).

The Program Invocation Stack

Before leaving the world of program invocations, one more concept needs investigation. When programs begin calling each other willy-nilly, the SSP needs some mechanism to keep track of which programs called which other programs and in what order. That mechanism is the ***program invocation stack*** (Figure 7.1c). As an analogy, think of how a cafeteria's spring-loaded stack of plates works, and you have a good picture of the program invocation stack's behavior. The main program is like a single plate on the spring-loaded stacker. Calling a subprogram is like putting a second plate on the stack — it "pushes down" the main-program's plate and the subprogram plate becomes the new top of the stack. When a subprogram returns to its caller, its plate comes off the stack. All the plates in the stack, then, represent invoked programs, and the top plate of the stack is the currently executing program. Understanding the program invocation stack is useful during debugging, because the set of currently invoked programs gives you useful information about how your application arrived at its current state. It also helps you keep in mind a limitation of EPCs: No program can call any currently invoked program. In the plate analogy, no program can call a program whose plate is already on the stack. Such calls are called recursive because they can result in the same program being called over and over. While recursive calls are useful in some esoteric situations, they aren't allowed with S/36 or AS/400 EPCs.

Figure 7.1c
Program Invocation Stack



Note also that each task has its own invocation stack — the set of invoked and active programs is private to each task, unlike MRT-NEPS.

Technical Note

On the S/36, program transfers from real memory can take anywhere from 4 to 76 milliseconds, depending on whose EPC implementation you use. (On the S/38, program transfers from real memory take less than 2 milliseconds and are even faster on the AS/400.) To put those transfer times in perspective, one disk access on the S/36 takes 35 milliseconds. Even though S/36 EPCs happen relatively quickly, evaluate carefully those tasks you defer to subprograms. Avoid repeated calls to programs that perform trivial tasks.

Performance Tip

SSP intelligently manages VM to minimize paging for frequently called programs. When many programs compete for real memory, SSP keeps a "popularity count" for each program and pages out programs with the lowest frequency of use. This keeps the most-frequently called programs in real memory, reducing VM paging (disk accesses) and keeping response times short. Programs that don't interact with the user, but which are called many times (perhaps in a loop) will stay in memory, keeping program invocation time to a minimum. This makes using modules to encapsulate such tasks as table-lookup and file I/O practical.

Coding EPCs: A Primer

With a good grounding in program invocation theory, you're ready to learn about the parameter interface and EPC coding. RPG has five new opcodes supporting EPC:

- CALL: calls a specified program
- PARM: specifies a parameter for CALL or PLIST
- PLIST: specifies the entry point of a called program
- RETRN: returns to the calling program
- FREE: deactivates an active but suspended program

The two program fragments in Figure 7.2 demonstrate the use of these opcodes and also help illustrate the basic steps of EPC coding. First, you specify the name of the subprogram to be called (PROGB in the example) with either a quoted literal or a field name. The named field, if specified, must contain the subprogram's name, left-justified, optionally followed by a period and the library name. For example,

```
MOVEL*BLANKS   PRGLIB 16
MOVEL'PROGB.MY'PRGLIB
MOVE 'LIB'     PRGLIB
CALL PRGLIB
```

calls PROGB in library MYLIB (field PRGLIB contains 'PROGB.MYLIB'). The library name defaults to the job step's current library (the library at the time of // LOAD) if you omit it. To further enhance programming flexibility, EPCs are *late binding*; the called program doesn't have to exist until runtime. Late binding lets you determine which program to call on the fly — a powerful feature.

Figure 7.2
Program Fragments with EPC Opcodes

```
*... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6

     C* Program A (PROGA)
     C                     SETOF                     202122
     C                     Z-ADD5         IVALA   50
     C                     Z-ADD7         IVALB   50
     C                     Z-ADD0         OVALC   50
     C                     CALL 'PROGB'
     C                     PARM           IVALA
     C                     PARM           IVALB
     C                     PARM           OVALC
     C*  ovalc now contains 12, and 20, 21, & 22 are still off
     C* Note: FREE use is shown but its use is not recommended
     C                     FREE 'PROGB'


     C* Program B (PROGB)
     C           *ENTRY    PLIST
     C                     PARM           IA
     C                     PARM           IB
     C                     PARM           OC
     C           IA        ADD  IB        OC
     C                     SETON                     202122
     C                     RETRN
```

Second, you list the names of variables to be passed to the subprogram. These variables, and these variables alone, communicate data to the subprogram. The subprogram may change the contents of these variables upon its return. In the example, PROGA has three PARM statements — specifying the variables IVALA, IVALB, and OVALC — to be passed as arguments to PROGB (in a calling program, PARM variables are called arguments; in a called program, they are called parameters, to distinguish between the two sides of the program interface).

PROGB's C-specs begin with the PLIST opcode (which always uses the keyword *ENTRY in factor 1) to specify the entry point of the program. PROGB lists three parameter variables — IA, IB, and OC — on individual PARM statements. An important rule of parameter interfacing is that each parameter's length must match the length of the corresponding argument variables in the caller, with the same number of argument and parameter variables. The parameter names in the subprogram need not match the argument names passed by the calling program; the names are irrelevant as long as the parameter count and lengths correspond. In fact, a CALL doesn't require any parameters as long as the subprogram doesn't expect any.

Technical Note

The *ENTRY PLIST line *does not* have to be the first C-spec. When called the first time, a subprogram always starts execution at the first C-spec. Subsequent calls, however, then start execution at the location of the *ENTRY PLIST. Locating the *ENTRY PLIST later in the C-specs lets you bypass one-time initialization of subsequent calls.

Performance Tip

After a main program ends, subprograms end in the *reverse order* of activation. So if the main program and any of the subprograms modified the LDA or the UPSI switches, subprogram changes to either override changes made by the main program. If you don't want the LDA or UPSI switches changed by a subprogram, consider using the FREE opcode to explicitly end subprograms before the main program ends.

The S/36's EPC mechanism passes arguments to parameters by reference — that is, you think of the variables as referencing the same physical memory locations. Changes made to any parameter values in a subprogram are also made to the corresponding variables in the calling program. PROGB in the example doesn't change the value of variables IA or IB, but if it did, those changes would be reflected in the corresponding variables IVALA and IVALB in program A.

In PROGB, the RETRN opcode causes an immediate return to the calling program. The return statement is logical, not structural — it can occur any number of times in a subprogram, not just at the physical end of the calc specs. This simplifies the coding of exception handling. For example, after detecting an error inside a nested series of IF statements, you could simply set a return code and execute the RETRN operation, eliminating the need for a GOTO out of the body of the IF structure.

Following the execution of the example program illustrates EPC coding in action. PROGA is calling PROGB to add two numbers together and return the result. First, PROGA sets off indicators 20-22, and sets IVALA to 5 and IVALB to 7. Next, PROGA calls PROGB, which is activated and receives the contents of PROGA's IVALA, IVALB and OVALC arguments in its corresponding IA, IB and OC parameters. PROGB adds IA to IB, storing the result in OC, sets indicators 20-22 on, and returns to PROGA. When PROGA resumes execution, its variable OVALC contains the value 12 and IVALA and IVALB are unchanged (because PROGB didn't change the values of IA or IB). Even though PROGB set on its copy of indicators 20, 21, and 22, those indicators are still off in PROGA because each program has its own set of indicators and other private data. If PROGB were to be called again, its indicators and private data would have the same values they held at the RETRN point.

In Figure 7.2, after calling PROGB and then resuming execution, PROGA executes a FREE statement. This unilaterally ends PROGB, forcing it to EOJ. The next time PROGB is called, a fresh copy will be activated and all private data reset to initial values. Although FREE *explicitly deactivates* a subprogram, all subprograms automatically get deactivated when the main program ends, so individual FREEs aren't necessary. You should only use FREE when

you need a fresh copy of a subprogram, keeping in mind the additional overhead for re-activating the subprogram. Another way to end a subprogram is by setting an indicator LR in the subprogram before RETRNing. This has the same effect as FREE, and for the same reasons, should only be done when circumstances require a fresh copy of the subprogram. Remember, the only significant resource-activated subprograms consume is VM (space in the TWA). They place no execution load on the system. Going out of your way to send subprograms to EOJ usually results in reduced performance.

Technical Note

Although the example treated IA and IB as input parameters, and OC as an output parameter, EPC makes no distinction — all parameters are both input and output. When designing modular applications, you should use some naming convention (e.g. 'I' for input, 'O' for output) to indicate which variables are intended for input and output. Keep in mind, though, that EPCs don't enforce your convention (see Chapter 9, "Implementing Modular RPG Applications," for naming-convention ideas).

EPCs and Disk Files

The code fragments in Figures 7.3a and 7.3b illustrate how to code EPCs with disk files. The OCL that calls the initial program must include a // FILE statement for each file used in a subprogram. In a large interactive application with lots of EPCs, it's not unusual to have many (30 or 40 or even more) // FILE statements between the // LOAD and // RUN for the main program. (Remember, in OCL you don't have the file limit imposed by RPG. You are, however, still limited to 15 disk files in any one RPG program.)

Performance Tip

See Chapter 17, "Harnessing the Power of Assembler Routines," for a tool that can reduce or eliminate the need for many // FILE statements in an EPC jobstep.

Figures 7.3a and 7.3b also illustrate what happens when two subprograms share a file (program C and program D use the same physical file). Just before calling program D, program C positions its file pointer to the fifth record in MYFILE. Upon return from program D, which positioned its file pointer to MYFILE's record number 10, program C is still positioned at its previous file position. This example illustrates the fact that each program has its own private copy of file buffers and file record positions.

This illustration brings up a subtle, yet possibly troublesome, effect of two subprograms sharing a file. The OCL in Figure 7.3a specifies that both logical copies of MYFILE be opened for modification by owner or other user (via the SHRMM keyword). PROGC's read of record 5 locks the record; PROGD subsequently deciding to read record 5 would result in a deadly embrace — PROGD would wait forever for PROGC to release the record. This problem

Figure 7.3a
OCL for Program File Sharing

```
// LOAD PROGC
// FILE NAME-MYFILEC,LABEL-MYFILE,DISP-SHRMM
// FILE NAME-MYFILED,LABEL-MYFILE,DISP-SHRMM
// RUN
```

Figure 7.3b
RPG File Sharing Between Programs

```
*... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6

     F* Program C (PROGC)
     FMYFILEC IC        128R           DISK
     IMYFILEC NS
     I                                       1 128 DIREC
     C           5          CHAINMYFILEC
     C                      CALL 'PROGD'

     F* Program D (PRDGD)
     FMYFILED IC        128R           DISK
     IMYFILED NS
     I                                       1 128 DIREC
     C           *ENTRY     PLIST
     C           10         CHAINMYFILED
     C                      RETRN
```

isn't directly related to the use of EPCs; it could happen just as easily in programs that don't use them. Be aware of potential record-deadlock situations in your applications, and code defensively to avoid them. (For more information about record locks, see Chapter 18, "Profiling and Advanced Debugging").

Technical Note

If two or more programs in a task reference the same file, the FILE statements must have unique NAMEs (but identical LABELs), with DISP set to compatible share levels. With third-party EPCs, the requirement for unique NAME parameters on the // FILE statement is handled by special statements in the RPG source program that equate the unique names with RPG internal names (which are probably the same for all programs). IBM's EPC isn't so slick: You must code unique file names on the RPG F-, I-, O- and C-specs as well, greatly complicating program maintenance (see Chapter 8, "A Comparison of EPC Vendor Offerings," for details on IBM's coding complications).

A Practical Example

A simple, real-world example drives home the value of EPCs in application design. Consider a typical order-entry program — one that provides for customer search and selection, inventory-item search and selection, order-line entry and edit, picking-ticket and invoice printing, and all necessary file updating. Using traditional techniques, this application would have been coded as one monolithic program, hard to maintain, difficult to comprehend, and impossible to enhance without unpleasant side effects.

Figure 7.4 shows what a small portion of a EPC modular might look like. PROGA is the top-level program, performing some basic services. When another service is required, customer selection for example, a CALL is made to PROGB, which actually performs the customer search. PROGB in turn calls PROGC to perform a service (perhaps to check the customer's credit limit). Based on a value returned from the CALL to PROGC, PROGB might immediately return to PROGA (if the customer were over the credit limit, for example). If PROG gets the go-ahead from PROGC, PROGB calls PROGD to perform services and immediately returns to PROGA to start the process over again. Note that program B has two exit points. Subprograms must have only one *ENTRY statement but can have many RETRN statements. However, the RETRN statements in your subprograms should be well documented; program exits buried in obscure places make maintenance more difficult.

The modular approach has both programming and performance advantages. Coding, compiling, testing and documenting small program modules proceeds much faster during development than the same cycle for other approaches. And after the application is in production, the inevitable changes and enhancements tend to affect only one or two modules at a time, isolating other modules from "enhancement rash" — inadvertent bugs introduced by maintenance coding. The performance advantage comes from reduced disk I/O: Once all the subprograms are activated, transfers between them are instantaneous, with no need for program initiation and termination. Not only does this eliminate unnecessary disk I/O, it leaves more I/O time available for performing application-related work.

Performance Tip

In a batch application, calling a subprogram to perform a trivial task could bring performance to its knees. Consider a batch main program that reads 100,000 records for update, calling a small subprogram once for each of 10 fields in every record. Those 10 CALLs, multiplied by 100,000 records, would add more than an hour to the processing time in the *best* circumstances (4 milliseconds per call). It's probably better to code subroutines entirely within the application if the overhead of the CALL greatly exceeds the subprogram's actual working time. (This might even be the time to drag /COPY out of the closet and use it to externalize the coding. Just let RPG internalize everything as one load member.)

Reaping the Benefits of EPCs

Isolating important parts of an application in subprograms gives you, and your users, some terrific benefits. In the old days, whenever you needed a customer-search program, you coded it (usually from scratch) in every program. Now, by writing a generic, flexible customer-search program, you have a process that goes about its private way of letting the user select a customer and then returning the primary key of the customer selected to the calling program. The calling program doesn't know how the search was implemented, nor does it care. It just needs the customer's primary key to continue its work. As

Chapter 8

A Comparison of EPC Vendor Offerings

External Program Calls (EPC) is one S/36 feature on which IBM played catch-up with third-party vendors. Although IBM built EPC capabilities into the S/36 operating system from the start, only IBM's own programmers had access to them. Eventually, as mentioned in Chapter 7, several independent software vendors cracked the secret and offered EPC — via the CALL and PARM opcodes — to RPG programmers. IBM enhanced its RPG compiler years later to also support CALL, PARM, and related RPG extensions — all as a chargeable PRPQ. Finally, as this book went to press, IBM had announced that RPG users would get CALL/PARM for free as part of the basic compiler license.

Now, as an RPG application programmer, you have a selection of programming environments offering EPC features. Choosing which product is best for you requires weighing the relative merits of each, with an eye to ease of use, performance, and future compatibility. Given IBM's late entry into the field, you might expect its "free" product to have one up on the early competitors; however, that's not the case. With IBM's EPC, you literally get what you pay for. Whether that will be adequate for your needs is a question only you can answer. This chapter gives you the facts upon which to base that answer.

The Contest

In addition to IBM's standard RPG product, two other RPG development tools — by independents Amalgamated Software of North America (ASNA) and BPS, Inc. — offer EPC features: 400RPG and RPG II½, respectively. Both of these products also add other RPG/400 language features in addition to CALL/PARM: externally described files, multiple-occurrence data structures, dynamic file open/close, the *IN indicator array, indicators as fields, AND<u>xx</u>/OR<u>xx</u> operation codes, and several miscellaneous advanced RPG operations. While these additional features might influence your purchase decision, here we examine only the EPC implementations. Because EPCs are so powerful and so important to reliable, maintainable, and portable application construction, you'll want to consider several important EPC-related factors before you look at other issues.

The primary factor to examine is each product's fidelity to IBM's established RPG language standard, RPG/400. You may need to migrate your applications to the AS/400 at some future date, at which point you'll be glad you considered AS/400 compatibility now. Even if you don't move to the

AS/400, RPG products for other computer platforms, such as the RS/6000 and PC, use RPG/400 as a touchstone for language compatibility.

Secondary factors to examine are usability and performance. Each product has some subtle (and some not-so-subtle) restrictions that may turn into coding barriers for your application programmers. And even if your programmers find a given EPC product usable, the resulting applications might not perform well enough. The time required to invoke a program thus becomes an issue, and each EPC product performs differently in this regard.

Because these evaluation factors — compatibility, usability, and performance — come up at every turn when developing modular applications, we examine the factors in the same order an application programmer is likely to consider them: design, coding, and testing.

Design and Coding Considerations

Putting together a modular application means decomposing the problem into well-defined tasks that can be isolated into separate modules and defining interfaces for each module. For the most part, all three products let you use all RPG commands and capabilities in called programs. There are limitations, however. None of the EPC implementations permits any program in an EPC task to have overlays — not an onerous constraint, as one reason for modularizing programs is to reduce program size. IBM adds a further restriction: It limits each module's size to 60 K — 4 K less than the maximum region size of 64 K. IBM uses the last 4 K to map onto the caller's memory for copying to and from the caller's arguments. You'll want to keep this limitation in mind — especially when incorporating existing RPG programs into a modular application design.

File I/O presents another design issue. Most modular applications share a common set of files, using common file layouts to improve maintainability. To this end, IBM's RPG/400 language standard supports common file description through *externally described files* — file descriptions stored outside program source code and copied in automatically at compile time. Both 400RPG and RPG II½ provide for external file descriptions compatible with RPG/400. IBM's S/36 EPC supports only the antique /COPY statement, which isn't compatible with RPG/400, presenting a serious application development problem.

The problem arises because S/36 SSP requires each program in a task using a common file to reference that file by a *unique* name at execution time — the name is specified via the NAME keyword on a // FILE statement. But at compile time you want these programs to reference the common file by the same name, so that the compiler can locate and copy in the description for the common file (Figure 8.1). RPG II½ and 400RPG let you have different compile-time and execution-time names for each file. You code the compile-time name as always: on the RPG F-spec. You code the unique execution-time name for that file on an extension spec for each file. Figure 8.2 illustrates this coding

Figure 8.1
OCL for a Single File Used by Two Subprograms

```
// LOAD MAINPG                            Main Program
// FILE NAME-CUSMASTA,LABEL-CUSMASTP,DISP-SHR    For PROGA
// FILE NAME-CUSMASTB,LABEL-CUSMASTP,DISP-SHR    For PROGB
// RUN
```

method for two programs, PROGA and PROGB, that both reference the same file, CUSMAST. Both programs have 'CUSMAST' coded on the F-specs, while PROGA has 'CUSMASTA' coded as the file's execution-time name, and PROGB has 'CUSMASTB' coded. The compiler uses the identical compile-time name to locate the CUSMAST file description and copy it into the program. SSP uses the unique execution-time names to associate each program with its corresponding // FILE statement. Both // FILE statements then reference the same physical file on disk, CUSMASTP. To later migrate programs using this notation to the AS/400, you simply delete the F-spec extension lines.

In contrast, IBM's EPC requires a unique F-spec file name for each file used by any program or subprogram. Figure 8.3 shows the previous coding example using IBM's native RPG. PROGA must use a different internal name for file CUSMAST than PROGB on the RPG F-spec, I-spec, C-spec, and O-spec. This effectively precludes external file descriptions using /COPY, or using any other method, for that matter. This approach also creates a maintenance nightmare, as copying code from separate programs that use the same file requires changing the file name used in the code. If you later decide to move to the AS/400 or another platform, you'll have to change all these file names again to take advantage of external file descriptions in the new environment.

Performance Tip

In addition to /COPY, both non-IBM EPC products also support AS/400-style externally described files. These are both easier to use and more powerful than /COPY. Use them!

If you plan to use EPCs only to connect existing programs that don't share a large number of files (and that you won't be maintaining much in the future), this limitation to IBM's EPC may not be an obstacle. But for implementing new modular designs, or for converting existing applications to modular ones, you'll likely find yourself frequently cursing IBM for this oversight.

A design issue related to file I/O is workstation I/O. Programs access the workstation device through a workstation file, and the workstation file identifies the name of a screen format member containing screen formats for use by the program. SSP has a limit of 255 screen formats for a given job, due to the way screen formats are opened and cached in memory. RPG II½ and 400RPG let you keep your screen formats in separate screen-format load members, as long as the total number of formats is less than 255. Any program can use screen formats used by any other program in a task, so the screen format names must also be unique between programs. With six-character program

Figure 8.2
RPG II½ and 400RPG Coding for Subprograms PROGA and PROGB Referencing a Common File

```
1...:...10....:...20....:...30....:...40....:...50....:...60....:...70....:..80
     H                                                                    PROGA
     FCUSMAST UD        128R            DISK
     F                                  K    OCLNAM CUSMASTA
     I/COPY CUSMAST
     C           *ENTRY    PLIST
     C                       .
     C                       .
     C                       .
     C                     READ CUSMAST
     C                       .
     C                       .
     C                       .
     C                     RETRN
     OCUSMAST E                CHGREC
     O                       .
     O                       .
     O                       .

1...:...10....:...20....:...30....:...40....:...50....:...60....:...70....:..80
     H                                                                    PROGB
     FCUSMAST UD        128R            DISK
     F                                  K    OCLNAM CUSMASTB
     I/COPY CUSMAST
     C           *ENTRY    PLIST
     C                       .
     C                       .
     C                       .
     C           1         CHAINCUSMAST
     C                       .
     C                       .
     C                       .
     C                     RETRN
     OCUSMAST E                NEWREC
     O                       .
     O                       .
     O                       .
```

names and eight-character screen-format names, you can meet this requirement by simply appending the program name with a two-digit sequence number to generate unique screen-format names.

IBM's EPC, however, permits only one screen-format member, shared by all programs. This becomes a design burden when integrating modules from application systems that might reside in separate libraries — the single screen-format module must reside in only one of those libraries. You can still keep separate screen-format *source* members, but you must compile them all

Figure 8.3

IBM RPG Coding for Subprograms PROGA and PROGB Referencing a Common File

```
1...:...10....:...20....:...30....:...40....:...50....:...60....:...70....:..80
     H                                                                     PROGA
     FCUSMASTAUD          128R           DISK
     ICUSMASTA
     I                        .
     I                        .
     I                        .
     C          *ENTRY    PLIST
     C                        .
     C                        .
     C                        .
     C                    READ CUSMASTA
     C                        .
     C                        .
     C                        .
     C                    RETRN
     OCUSMASTAE               CHGREC
     O                        .
     O                        .
     O                        .
1...:...10....:...20....:...30....:...40....:...50....:...60....:...70....:..80
     H                                                                     PROGB
     FCUSMASTBUD          128R           DISK
     ICUSMASTB
     I                        .
     I                        .
     I                        .
     C          *ENTRY    PLIST
     C                        .
     C                        .
     C                        .
     C          1         CHAINCUSMASTB
     C                        .
     C                        .
     C                        .
     C                    RETRN
     OCUSMASTBE               NEWREC
     O                        .
     O                        .
     O                        .
```

into a single *load* member using the SSP FORMAT procedure.

One final design consideration affects module interfaces. All three products let you pass a number of variables as parameters, but IBM's product restricts you to a maximum of 15 parameters. This may seem like plenty; but when you're in the thick of modular design, running into this limitation forces

you to complicate interfaces by passing data structures that contain multiple parameters. Not only does this make interfaces less clean, it eliminates the advantage of length-checking provided by EPC at execution time. Changing the data structure in one program but not another can result in parameter data overlapping in a called program — a very difficult problem to track down.

Again, depending on how far you plan to "buy into" EPCs, this limitation may or may not be significant to you. Both RPG II½ and 400RPG make the point moot: They allow passing any number of parameters.

Run the Good Race

Once your programs are up and running, you must deal with performance issues. One fact that can affect performance is the time required to make a subprogram call. Although all three EPC products use the same underlying EPC mechanisms, each has a different way of handling program activation and parameter passing, which causes considerable variance in the time required for a CALL.

The importance of this factor in your situation depends on how you use EPCs. If you plan to call a given subprogram iteratively, the work it performs should justify the overhead of making the call. Every call requires a minimum amount of time — perhaps no more than the equivalent of two or three RPG divide operations — to locate the activated subprogram and establish addressability to it. More time may be required to page in the subprogram from disk if it happens to be paged out. When the subprogram returns, reestablishing execution in the caller requires additional time. Consequently, repeated calls to a subprogram that performs some trivial computation (adding two numbers, for example) makes no sense. Subprograms used in this manner become a processor bottleneck unless the repetitions are few.

However, a module doesn't have to be complicated to be a good subprogram. It is eminently practical to call a subprogram that accumulates statistics in a 50 K array — thereby keeping data in high-speed real storage — rather than storing the statistics in a disk file for use by a monolithic program that must fit into a 64 K region. Another example is an alphabetic search function, which several modules in an application might require. By implementing the function once as a called module, you gain both the time and storage economies of activating the module only once per user, and simplified application maintenance. Generally, the decision about which functions to place in a subprogram should be dictated first by design and then by performance. Merging modules that are the result of too-detailed decomposition is usually easier than trying to continue decomposing the design of a system already in production. But if your application design tends to favor encapsulating low-workload functions (such as table lookup) that are frequently invoked, CALL overhead may become a factor. One additional rule of thumb: For subprograms performing

Figure 8.4

Performance Test of 10,000 CALLS Passing 128 Bytes of Parameters

Vendor	Elapsed Time (Seconds)	Time per Call (Milliseconds)
IBM	950.0	95.0
ASNA	190.0	19.0
BPS	44.0	4.4

workstation I/O, CALL overhead is always inconsequential. Thus, dividing large programs along workstation I/O boundaries is a good way to modularize an existing program without introducing performance problems.

All three products use *late binding* when activating a program for the first time. With late binding, the name of the program to be called is determined at execution time, rather than at compile time. Because late binding lets you generate the called program name on the spot, you don't have to recompile calling programs after making changes to a called program. Thus, the process of activating a program for the first time includes locating the program in a library, loading it, initializing variables, and opening files. Each product accomplishes this using IBM-supplied SSP library services; so it isn't surprising that the time for program activation for all products is about the same: approximately 100 milliseconds (one tenth of a second).

Once a program is activated, though, subsequent invocations are much faster. How much faster varies with each product. Figure 8.4 shows the elapsed time for each product making 10,000 calls to a do-nothing subprogram, passing 128 bytes of parameter data (the tests were performed on a S/36 model D). The first call, which activated the subprogram, is not included in the 10,000 count. IBM is the slowest contender, taking 950 seconds. This works out to 95 milliseconds per CALL/RETRN sequence — a little more time than that required for two disk operations. The similarity to disk I/O times isn't accidental: For every CALL/RETRN, IBM's EPC performs disk I/O related to the work of passing parameters and changing addressability to the subprogram.

ASNA's 400RPG comes in second place, at 190 seconds, or 19 milliseconds per CALL. ASNA also performs disk I/O for every CALL. However, because only a single disk read is being called, the disk arm tends to remain stationary in this test; only disk I/O rotational delay and transfer time are consumed. In real life, a frequently called subprogram might move the disk arm, however, which would add seek time (12 to 25 milliseconds) to the overhead for each CALL.

Fastest is RPG II½, which ran the race in only 44 seconds, yielding a per-CALL time of just 4.4 milliseconds. This is less than the time it takes RPG to

perform a couple of DIV operations! RPG II½ owes its speed to the use of a reentrant parameter-passing transient, which once loaded into memory never needs to be reloaded. Thus, RPG II½ performs no disk I/O for most CALLs, so disk arm placement (and the effect of subprogram I/O) has no effect on the overhead for calling to and returning from the subprogram. However, the amount of data transferred now has some effect on CALL time. Where IBM and ASNA have uniform times for parameter lengths ranging from 128 to 1024 bytes, BPS shows increased times for larger parameters. At 256 bytes, RPG II½ jumps to 7 milliseconds per call, but then only increases to 9 milliseconds for parameter lengths up to 1024 bytes.

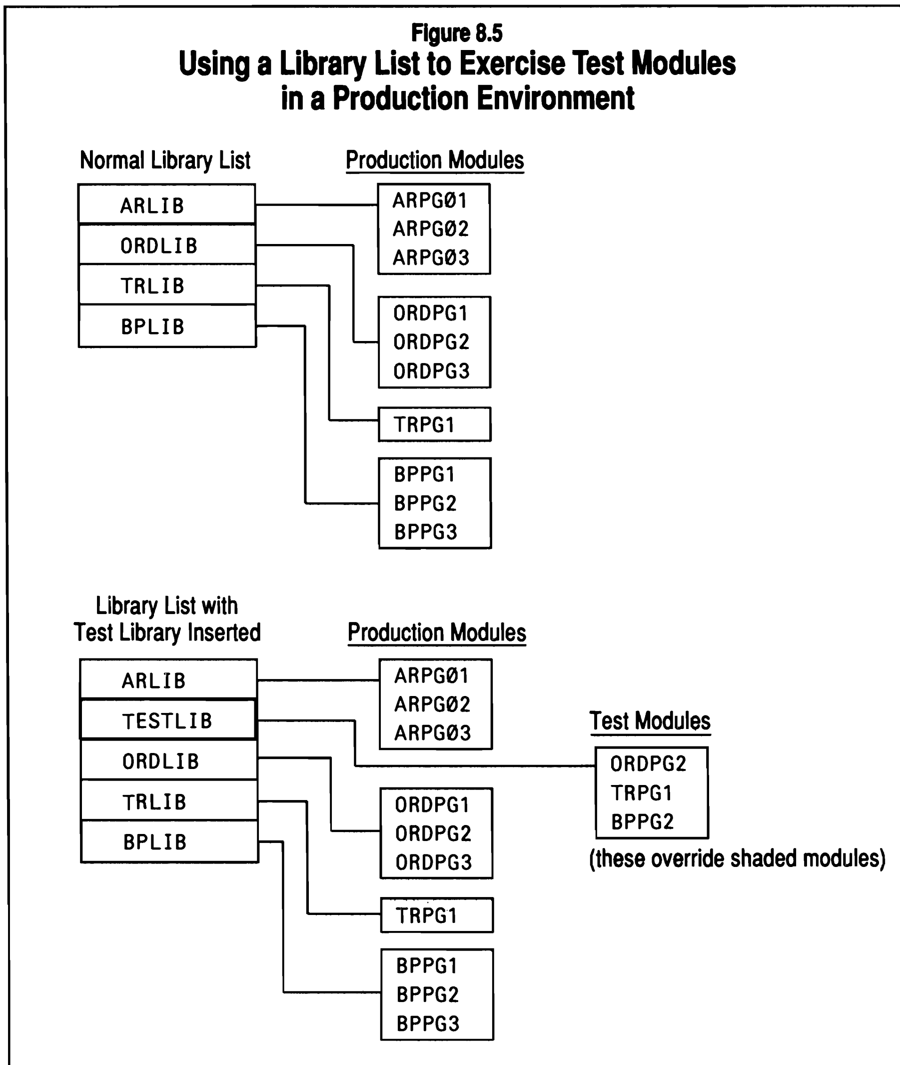
Testing and Production Considerations

Once you've designed and coded your modular programs, you need to test them, and after ensuring they work properly, put them into production. Testing usually proceeds in two phases: unit and integration. In unit testing you exercise individual modules in isolation, using "scaffolding" code to provide the minimum amount of ancillary structure necessary to provide input to, and collect output from, a single module. Integration testing verifies correct interaction between modules in as close to "live" conditions as possible.

The AS/400 offers several features that help you with unit and integration tests. One is the ability to run subprograms as standalone programs, which simplifies unit testing. When testing a module as a standalone program, you supply parameter values via hardcoded values or temporary input files and collect output parameters on printed listings or in temporary output files. The ability to test modules as standalone programs eliminates the need to build one-time "main" programs that simply call the module to be tested, saving time during unit testing. Both RPG II½ and 400RPG let you run called modules as standalone programs using // LOAD. IBM's EPC doesn't. In fact, subprograms must be coded with an 'S' in position 55 of the RPG H-spec to let the compiler know that the *ENTRY PLIST and RETRN operations are legal. The compiler then flags the program as non-executable (i.e., you can't use // LOAD to invoke the program). This limitation of IBM's EPC implementation rules out standalone unit testing — you must instead write one-time programs whose sole purpose is calling the subprogram under test.

Another AS/400 feature geared to testing modular programs is the *library list* facility. A library list is a list of libraries to be searched to locate called programs. When a CALL doesn't specify a library name (or specifies *LIBL), the system looks in each library in the library list for the target program until it finds the program. Each running job has its own private library list. The library list facility simplifies testing modules under development alongside production modules: You simply insert the name of the library containing your test modules into the library list ahead of the production library names (Figure 8.5).

Figure 8.5
Using a Library List to Exercise Test Modules in a Production Environment



The test library can contain procedures and programs that refer to test files, while the production library contains procedures and programs operating on production files. (On the AS/400, files, as well as programs, are contained in libraries, extending the value of library lists even further). ASNA's 400RPG is the only EPC product that supports library lists. While it only supports this feature for programs, and not for files, it still makes a valuable testing tool. Library lists are useful in a production environment, too, as they let you use modules from many different libraries within the same task.

The Gravy

As mentioned earlier, IBM's RPG compiler offers CALL/PARM as its only enhancement, completely missing such modular programming aids as externally described files and library lists. While you must pay for third-party EPC products, you get a host of "gravy" features in the bargain. Beyond extras mentioned at the beginning of this chapter, ASNA supports data areas, AS/400-style DDS statements for external file descriptions, RPG/400 V2R1 enhancements, and a facility for invoking S/36 procedures from inside an RPG program. BPS also supports DDS, but lacks the other ASNA extras. It has a few extras of its own, however: a macro facility (portable to the AS/400) that lets you create your own RPG macro operations and an integrated symbolic debugger. ASNA's product is significantly more expensive than BPS's, though, so you'll have to weigh the value of its unique features carefully.

Wrap Up

One final issue needs examining if you plan to make your plug-and-play modules available for others to use. While all the EPC products are compatible at the source-code level (due to their common RPG/400 heritage), none of them — including IBM's — is compatible with any other at the object-code level. Because of differences in parameter-passing techniques, you can't make, say, an IBM EPC program call an ASNA 400RPG program. For in-house programmers with access to their source code, this isn't likely to matter. But for independent software vendors wanting to incorporate EPC interfaces into their products for the benefit of their customers, IBM's third option simply complicates an already complicated situation. Such vendors must either provide source code for their products so that customers can recompile the programs using the EPC environment of their choice, or they must provide separate compiled versions of their products for each different EPC environment. Both independent vendors let you freely distribute their runtime modules with your product, which ensures that your customers can use your EPC-oriented product even if they are not ASNA or BPS customers. With IBM, you must ensure that your customers have installed release 6.0 of SSP (or PTF 3600 for SSP release 5.1), which involves you in customer system software maintenance.

Whatever your needs, you now have enough information to make an educated choice. IBM's belated entry into the S/36 EPC arena gives you basic EPC functions, but nothing else. You can use IBM's EPC as an experimental platform for investigating EPC use in your own applications. However, when you start running into IBM's EPC limitations, you may want to consider either ASNA's or BPS's offering; each provides much more function than IBM's while remaining compatible with the AS/400.

Product Information

Amalgamated Software of North America (ASNA)
P.O. Box 1668
42011 Big Bear Boulevard
Big Bear Lake, CA 92315
(800) 321-2762 or (714) 866-9000
400 RPG — $2,250 (or optional $200/month rental)

BPS Information Services, Inc.
P.O. Box 9, Department N3
Marion, IA 52302
(319) 377-7599
RPG II½ — $950

IBM
(contact your local IBM marketing representative)
SSP Release 6.1

Chapter 9

Implementing Modular RPG Applications

Chapter 7 described the advantages and mechanics of External Program Calls (EPCs), and Chapter 8 presented the relative merits of various S/36 EPC products; but you've not yet seen how to build a modular application from the ground up. The term "modular" in this context means a single application program broken down, or "decomposed," into small routines. In contrast with the most common way S/36 users first take advantage of EPCs — to connect existing programs for improved response time, the goal of modular programming is to produce programs that are easy to understand, debug and modify. One immediate benefit of modular design, manageability, means you can shorten development time by making each module a separate work assignment so that parallel coding and testing can occur. Another immediate benefit, comprehensibility, implies that because you can study the application one module at a time, the whole application will be better understood and better designed. The long-term benefit, flexibility, means you can change a module drastically without having to change other modules.

This chapter describes the modular design process, first explaining a few terms and concepts, then illustrating an example application design.

The Mysterious Module

The ideal software module should behave like a black box. That is, it should accept input, process it, and produce output without requiring other modules to know how the work was accomplished. Communication between modules should take place through well-defined interfaces, while variables and algorithms used by those modules remain hidden. The idea of concealing the implementation of a module — that is, how the module works internally — from the programs that call it is called *information hiding*.

Why hide such details? Because a program that has access to the internals of another module can become dependent on those internals, complicating the process of changing the implementation at some future date. Information hiding forces programs to interact at only one point: the module interface. Traditional RPG subroutines, on the other hand, often interact at many points — through variables, indicators, even file I/O. *Coupling* is a term used to describe the degree to which modules interact with one another. Tight coupling means the modules interact at many points; loose coupling means they interact at only a few (preferably one) points. Tight coupling makes program

debugging and maintenance harder because it increases the likelihood that a change in one module will affect another module adversly. Loose coupling has just the opposite effect: It tends to isolate the effect of changes to just the module changed.

In RPG, the only way to implement loosely coupled modules is to use separately compiled programs that interface with each other using EPCs. A main program calls external subprograms to carry out the work of the application, using only the CALL/PARM interface to communicate with the subprograms.

From the RPG programmer's viewpoint, external subprograms serve much the same function as internal subroutines: Both help you subdivide application code into logically distinct, reusable units. But external subprograms differ from internal subroutines in important respects, including the way they are invoked, the scope they give to named variables, the way they communicate between modules, and the time at which the subprogram name is bound to an address in memory. Here is a quick review of the differences between external subprograms and internal subroutines.

Activation and Invocation. The first time an external subprogram is called, it is *activated*: Virtual storage is assigned, the program is loaded, variables are initialized, and files are opened. When the subprogram returns to its caller, it remains activated in virtual storage; all variables remain intact until the next call. Each call, including the first, is referred to as an *invocation.* An invocation ends when control returns to the caller, but the activation of the called program persists. Internal subroutines are similar: They're activated (when the main program starts up) and the contents of variables remain intact across invocations. But the resemblance ends there.

External Subprogram Deactivation. At external subprogram deactivation, the subprogram's files are closed and its virtual storage is released. Consequently, if the deactivated subprogram is called again, it goes through the complete activation process; variables are reinitialized and files are reopened on the subsequent call. Internal subroutines don't have the concept of deactivation— except in the sense that they get "deactivated" when the main program ends. This means that there is no way to get an internal subroutine back to a "fresh" state without manually reinitializing its variables.

Local Names. In RPG, all tags, fields, indicators and file-access paths have names. A name can be either *local* or *global* in scope. A name is considered local if it is known only to the program or subroutine in which it is defined; it is considered global if it is known elsewhere. All names in an internal subroutine are global with respect to the program; thus, a variable defined in one subroutine is thereby defined for all subroutines and the main program. In contrast, the names in an external subprogram are local with respect to the program, which means they are inaccessible to other subprograms in the same task or job step. When an external subprogram first is called, the fields, indicators,

and file access paths all have initial values; on subsequent calls these items retain their values from the prior call, regardless of whether or not the internal variable names are used in other subprograms.

Parameter Passing. Internal subroutines communicate with the main program and other subroutines through global variables. However, because external subprogram variables are local rather than global, subprograms use *parameter passing* for intermodule communication. When an RPG program calls a subprogram, it optionally passes data to the called program through the PARM variables specified on the call. You can see that internal subroutines end up, by definition, having tight coupling with the main program. External subprograms have loose coupling because variable names have local scope, and the only interaction between modules is through the CALL/PARM interface.

Late Binding. When coding an EXSR statement to invoke an internal subroutine, you must know the name of the subroutine, and the subroutine must exist when you compile the main program. With external subprograms, the calling program need not know the name of the subprogram until just before making the call. This *late binding* of the program name to the invocation is much more flexible than *early binding*, where the subroutine name must be known at compile time. Because late binding allows you to generate the called program name on the spot, it is easy to implement table-driven designs, or to call user-selected programs. For example, a menu program might use a table relating menu options to program names. This table could be conveniently stored outside the program, so that changes in the table don't require recompilation of the program. Late binding also lets you compile and test an application in smaller increments, and later, to change subprograms without changing the main program.

Activation, deactivation, local names, parameter passing, and late binding — these are the features that set external subprograms apart from internal subroutines, making information hiding and loose coupling possible in RPG. With these two capabilities in hand, you're ready to begin the modular design process called *functional decomposition.*

Breaking it Down

Traditional RPG program design involves breaking down a problem into logical programming steps, using a main program and a set of internal subroutines to carry out these steps. This approach uses procedural design criteria (i.e., a flow chart) to make each major step into a subroutine. However, following a flow chart to break down, or decompose, a problem results in a *procedural* decomposition, with each module dependent on the previous module. Tight coupling results because internal subroutines have only global variables. Modular programming uses *functional* decomposition, in which you design modules to perform specific functions rather than steps in a procedure.

As a simple example, consider a program that reads records from a file and prints a report. A procedural decomposition would use one subroutine to read a data record, another to perform computations, a third to print header lines, a fourth to print detail lines, and a fifth to print total lines. The main program would call each subroutine in turn, in a loop, to produce the report. A functional decomposition would use a module to read the file, another to perform computations, and a third to print header, total, and detail lines. The main program would call the print module, which would call the read and compute modules as required to produce the report.

With the procedural design, changing the input data record requires changing the main program and all the subroutines, because each has access to all the data. The functional design, however, requires changing only the read module — the parameter interface to this module isolates other modules from the input file changes.

This gives you a taste of the functional design process. Following the steps for completely decomposing a real-world problem will give you the information you need to do your own functional design.

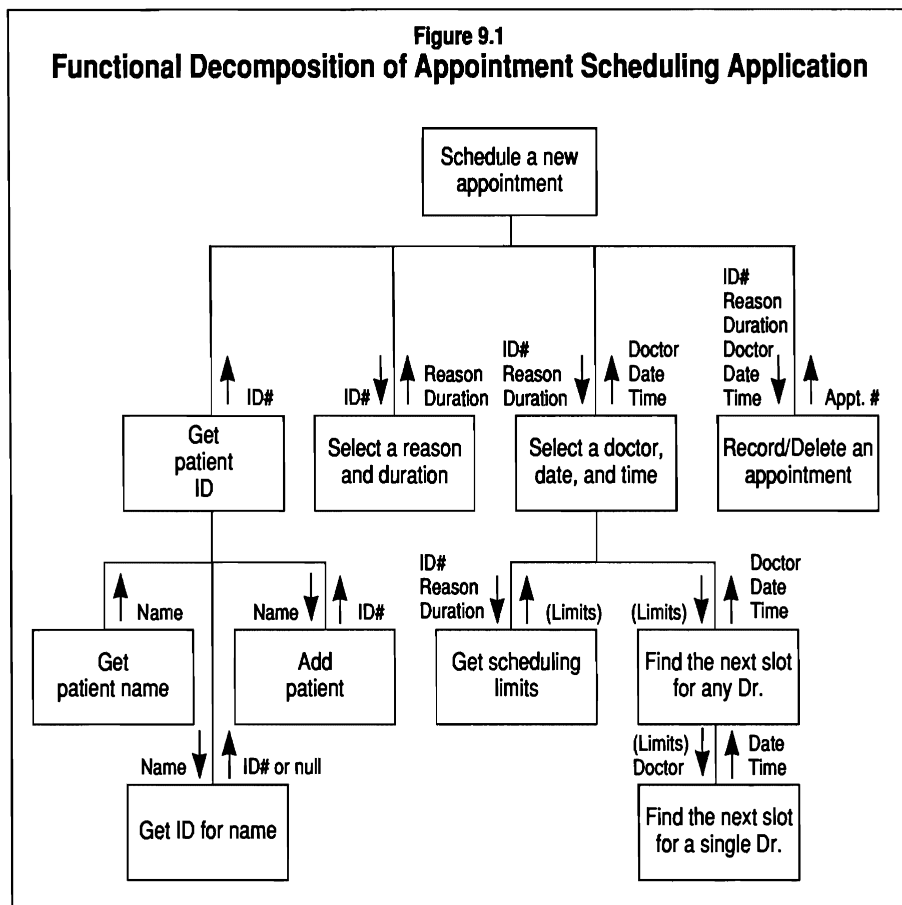
Defining the Problem

Appointment scheduling for medical clinic patients is a problem well-suited to modularization using external program calls. In this small application, a receptionist collects information from a patient, Mary Jones, to schedule a future clinic appointment. Before making an appointment, the receptionist needs seven pieces of information: the patient's name and identification number; the reason for and duration of the appointment; the doctor Mary will see; and the date and the time of the appointment.

The receptionist does not have this information at hand, but by interacting with Mary, she can obtain the necessary data. The receptionist can get the patient's name directly from the patient, but because Mary probably doesn't know her identification number, a search by name is necessary to retrieve the patient ID number. Also, Mary might want to see a particular doctor or might be willing to see any doctor who deals with the problem at hand, so a search by doctor or appointment reason is necessary. Finally, the receptionist must select the date and time of the appointment by considering both Mary's and the doctor's schedules.

To approach these data-collection problems, you first should decompose the program so that individual modules (e.g., the search by name) can be reimplemented in the future without affecting other modules. You also should create simple interfaces between modules, revealing as little as possible about the inner workings of the module. Finally, you should hide at least one key design decision in each module to make your application more flexible. You can decide which design decisions to hide in a module by determining

Figure 9.1
Functional Decomposition of Appointment Scheduling Application



which decisions are likely to change. Then you will be able to modify those modules when necessary, without affecting the behavior of the modules. For example, you might decide to hide the implementation of "finding a patient's name" in its own module and use only parameters to communicate with other modules. Then if you change the method of finding a patient's name (e.g., to improve the performance of the application), you can freely change that implementation without adversely affecting other modules in the system.

Implementing the Modules

Figure 9. 1 shows how to functionally decompose the appointment scheduling problem into 11 externally called modules. Each box represents a module. Above each box is a list of data items that specify input to and output from each module. The top box represents the controlling module of the function

being decomposed — scheduling a new appointment. The "Schedule a new appointment" module is the "main" RPG program. It has input parameters that can supply any known information (e.g., doctor's name). The module takes advantage of any information supplied by these parameters; when the information is incomplete, the module calls one or more external modules to complete the information.

When the main program calls the "Get patient ID" module, it passes no parameters *to* this module and receives only the patient ID parameter as output from the module. The details of how the module accomplishes its task are hidden from the caller. In this case, "Get patient ID" has been decomposed further into three subordinate modules: "Get patient name," "Get ID for name," and "Add patient." Each of these subordinate modules hides a design decision that might change in the future.

The "Get patient name" module prompts the receptionist to enter the patient's name and returns the name in a parameter to be passed to other modules. The method of prompting (i.e., terminal or workstation I/O) is the design decision hidden here.

The "Get ID for name" module accepts a patient name in an input parameter, and returns the patient ID (or null if the system does not recognize the patient) in an output parameter. This module might handle the problem of resolving duplicate names by presenting the receptionist with a list of similar names from which to choose. For example, the receptionist might input "Jon" instead of "Jones" and get in return the list of names beginning with the letters "Jon" with their corresponding addresses. Thus, the receptionist could check for correct spelling and compare addresses in the case of duplicate patient names. The algorithm used to select similar names could change, so this module hides the algorithm's implementation.

When the "Get ID for name" module fails to identify the patient (i.e., if the patient has never been to the clinic), it returns a value of "null" to the caller. When the "Get patient ID" module receives a "null" from "Get ID for name," it calls the "Add patient" module to perform the new patient registration process: Assign an ID, record patient history, and return the new ID to the module's caller.

Once the "Get patient ID" module returns an ID to the main module, the main module calls the "Select a reason and duration" module to perform the next function. This new module has two output parameters: a phrase describing the patient's problem and the approximate time required to treat the problem. This module hides the method used to generate these parameters from the "Schedule a new appointment" module. The receptionist might ask Mary for a reason or might inspect Mary's history to determine some likely reasons for her visit. The receptionist could display the history along with a list of default reason selections. Once the receptionist knows the reason for the

visit, she might consult a table of ailment treatment times to determine the appointment duration.

The main module next calls the routine to "Select a doctor, date, and time." This module requires a few more decisions. It accepts the reason for the appointment and the appointment duration as input parameters. The receptionist must find an empty time slot large enough to accommodate the required treatment time for a doctor qualified to administer the treatment. To complicate things, Mary might have her own scheduling limitations, such as "only after three o'clock" or "before next Tuesday." She might need to see a particular doctor or might accept any doctor able to provide treatment. The "Select a doctor, date, and time" module calls the "Get scheduling limits" module to determine which of these limitations apply for the given patient. The "Get scheduling limits" module receives three input parameters — the patient ID, appointment reason, and duration — and returns one output parameter that contains the limits.

The "Select a doctor, date, and time" module then calls the "Find the next slot for any doctor" module, which accepts scheduling limits as input parameters and calls the subordinate module "Find the next slot for a single doctor" iteratively until the subordinate module locates a slot that satisfies those limiting parameters. All these details are hidden from the main module; it is satisfied to get a doctor, date, and time by any means.

Finally, the "Schedule a new appointment" module calls the "Record/Delete an appointment" to record the appointment. This last module receives as input parameters the entire set of collected data items: identification number, appointment reason and duration, doctor, date, and time. The module records these items as a record in the appointment file, an entity known only to this module. Then the module assigns an appointment number associated with this record and passes the appointment number parameter back to the calling module. The "Record/Delete an appointment" module hides the method of storing an appointment from the "Schedule a new appointment" module so that any future changes to appointment record storage will affect only the "Record/Delete an appointment" module.

Because of this modular design, you can immediately reap the benefits of modularization by external program calls. You gain comprehensibility because you can understand the entire application by studying the individual modules. In fact, at each level in the design, you do not need the details of lower levels for overall comprehension. All of these modules are manageable because their hidden design decisions and passed parameters make them self-contained, with well-defined interfaces, resulting in loose coupling so you easily could assign each module to a separate programmer for implementation. Loose coupling between modules increases flexibility, making the application easier to modify; the modules are not dependent on each other and are not

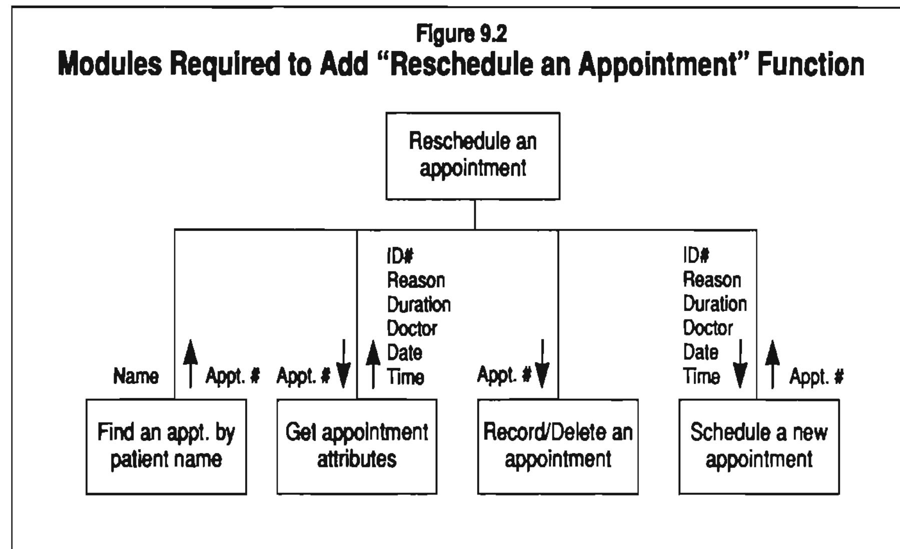


Figure 9.2
Modules Required to Add "Reschedule an Appointment" Function

tied to each other by necessity. When we propose some changes and estimate the effort needed to implement the changes, you can see the actual flexibility of the design.

Change Is No Problem

A likely change to any system is the addition of a new function to solve some new problem. In a real clinic situation, patients do not run their lives on fixed timetables, so they often want to change previously scheduled appointments. Thus, when Mary Jones calls to change her appointment, the application needs a new function to "Reschedule an appointment." Figure 9.2 shows how to functionally decompose this particular problem into modules. The top box in the diagram represents the controlling module that needs to be created for this new function.

Any of the parameters used to make the original appointment (e.g., reason, duration, doctor, date, or time) could change, so the receptionist must retrieve information about the old appointment. This requires knowing the appointment number of the old appointment — information possibly not at the patient's fingertips. So a new module, "Find an appointment by patient name," is created to retrieve the appointment number parameter. The controlling module passes the patient name to the "Find an appointment by patient name" module and receives in return the old appointment number. (Note that the "Find an appointment by patient name" module may use modules found in Figure 9.1 — e.g., the "Get ID for patient name" module — or other modules.)

The "Get appointment attributes" module also must be created. This

module receives the appointment number as input and retrieves appointment parameters needed for rescheduling: ID, reason, duration, doctor, date, and time.

Once all the attribute information has been retrieved, there are two possible ways to reschedule an appointment. You could *change* the attributes of the existing appointment to coincide with the new appointment, or you could *delete* the old appointment and retain the original parameters as defaults. Changing the existing appointment is a complex solution because changing appointment attributes may change the "limits" (e.g., limits of lab-related appointments and patient's and doctor's schedules) of the appointment. In the decomposition of the original problem, recording an appointment and deleting an appointment were different procedures within the same module (i.e., the "Record/Delete an appointment" module). Thus, this module can be used to delete the old appointment and add the new appointment, eliminating the need to create a more complicated module to change an appointment. The "Reschedule an appointment" module calls the "Record/Delete an appointment" module, which at this point uses the appointment number as an input parameter and removes the old appointment from storage. Note that the "Record/Delete an appointment" module was used in the original decomposition at a level below the controlling "Schedule an appointment" module. In this modification, it is used on the same level as the "Schedule an appointment" module, reinforcing the flexibility of modular programming.

For the final step in the appointment rescheduling process, the controlling module calls the existing "Schedule a new appointment" module. When the module is used to *reschedule* an appointment, it receives as input parameters the original appointment attributes and returns as output parameters both the changed and unchanged attributes. Usually only one or two attributes change when rescheduling an appointment, so the receptionist can avoid reentering unchanged data because the module is passed the original attributes.

Adding this rescheduling function requires writing one new main controlling module and three new subordinate modules. The "Schedule a new appointment module" remains unchanged except to accept the old appointment attributes as input. This enhancement has very little impact on the rest of the system because of the clean, decoupled modularization of the original design. If this modular design were based on procedural criteria rather than functional information-hiding criteria, many more lines of code would require changing and testing.

The rescheduling change also results in new modules at a higher level in the modularization hierarchy. What happens when you change a low-level module? When the modules are loosely coupled as they are in this application, very little happens. In the case of the fictional clinic, suppose you discover that some medical procedures can be performed only at certain times of the day. For example, suppose laboratory specimens are picked up at noon, so

lab tests must be scheduled for the morning. You should change the application in the "Get scheduling limits" module so you can apply scheduling restrictions to appointments that generate lab specimens. You also must change the module's internal logic to examine the reason for the appointment and, if the reason is lab-related, to restrict time selection to mornings only.

The method and data structures used to implement this change to the "Get scheduling limits" module have no maintenance impact on the rest of the application, because reason and duration will still be passed to the module and it still will pass back limits. Even the new modules for the rescheduling enhancement take advantage of this functional change without the extra effort of modifying other modules. Again, if you applied this change to a traditional application designed with modules decomposed from flow charts, you could expect changes in many more modules.

Clearly, you need to properly decompose a problem to accrue the benefits of modularization. You cannot build modular programs when you use tight coupling and do not hide design decisions. But if you do hide one key design decision in each module, you make it easier to separate functions, avoid the trap of decomposing according to traditional "processing steps," and consequently gain the benefits of modularization. External program calls provide a good path to implementing modular designs when you use well-defined, stable interfaces, loose coupling, and information hiding.

Living With Disk Data Management

"I can only assume that a 'Do Not File' document is filed in a 'Do Not File' file."
—Senator Frank Church
Senate Intelligence Subcommittee Hearing
1975

You use it. You fight it. You curse its lack of sophistication. S/36 Disk Data Management (DDM) is perhaps the worst bane of every application programmer. Yes, it's easy to get DDM to do what you want; the problem is getting DDM to do what you want quickly! DDM seems to have a whim of steel — sometimes fast, sometimes slow, but never either in any predictable fashion.

This section is a practical hands-on guide to making DDM perform well. Chapter 10 explains the much maligned and misunderstood disk blocking feature of DDM, giving you definitive answers about why, where, when, and how to use DBLOCK and IBLOCK. Chapter 11 explains a number of traps lurking in DDM for the unwary programmer, and workarounds for each. Chapter 12 presents a collection of tips and techniques that make DDM faster in everyday operations.

Many unique utilities make their debut in this section; each utility also is provided on the disk accompanying this book. A few of these are refinements of previously published tools, but most are available here for the first time. All will make the job of programming and administering your S/36 installation easier and more productive.

Chapter 10

Using DBLOCK and IBLOCK Effectively

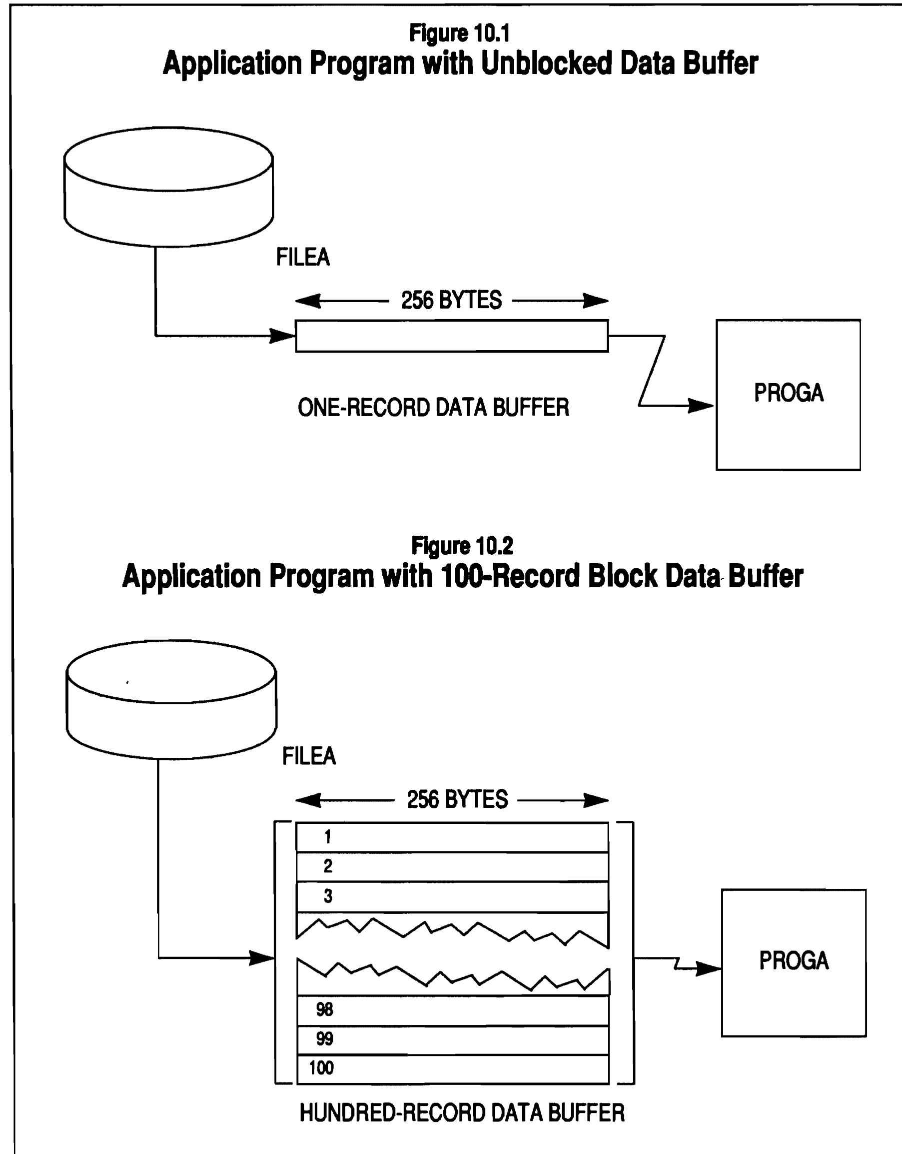
As we've mentioned often in this book, disk I/O is the largest single factor affecting S/36 performance. Generally, the best and easiest way to improve program response times is to reduce disk I/O. And one way to reduce disk I/O from within your programs is to use record and index blocking effectively. You must exercise care, however: As easy as it is to use blocking to improve application performance, it's just as easy to use blocking to ruin application performance. In this chapter, we'll look closely at record and index blocking, how they work, when you should and shouldn't use them, and how they relate to each other.

Blocking Data Records

Record blocking does for disk records what an egg crate does for eggs. Without an egg crate, you can grab only one egg at time from your refrigerator. However, if your eggs are stored in egg crates, you can grab 6, 12 or even 24 eggs at once — depending on the size of the egg crate. Think of record blocking as an "egg crate" for data records. With record blocking, the S/36 can move more than one record at a time between your program and disk, thereby reducing disk access and improving program performance.

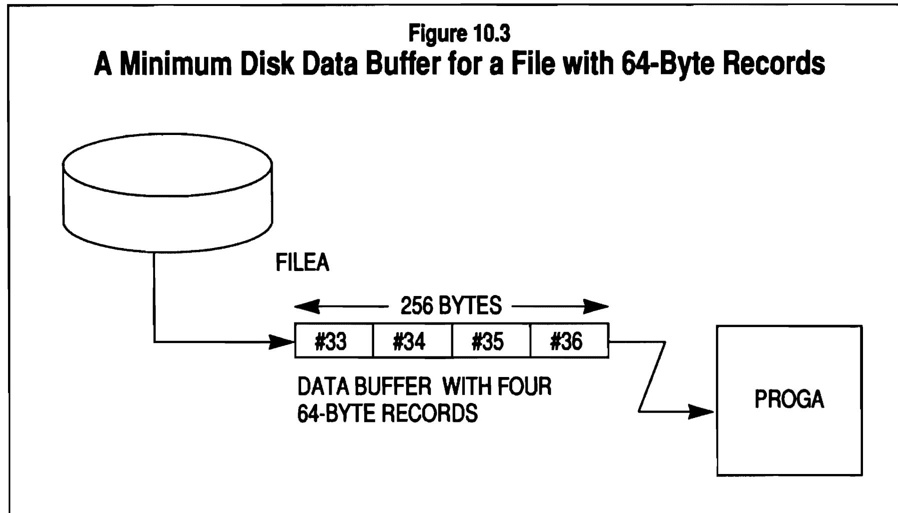
More technically, record blocking is the technique of buffering multiple records in memory for quick access by your programs. Each data file used by a S/36 application has a *data buffer* associated with it. (Indexed files also have an *index buffer*, which is discussed later in this chapter.) When your application requests a record, Disk Data Management (DDM) first puts the record in this data buffer and then moves it from the buffer to your application's data fields. Your program can read only one record at a time, but the system — transparently to your program — can put more than one record in the data buffer at a time. With multiple-record buffering, or record blocking, any record in the buffer is available to your application without performing any disk I/O.

Figure 10.1 shows a 256-byte data buffer, containing only one record, allocated by default for a file with a 256-byte record length. When PROGA reads the first record in FILEA, the buffer will contain only that first record. Each subsequent read will require an additional *physical* disk I/O operation — DDM will move records from FILEA to PROGA through the buffer one record



at a time. Figure 10.2 shows the data buffer allocated at 100 records. With this large buffer, when PROGA reads the first record, DDM also puts the next 99 records in the buffer. No disk I/O is required as PROGA reads the next 99 records. Only when PROGA requests a record not in the buffer — record 101 in this example — will the buffer be refreshed with the next 100 records.

Figure 10.3
A Minimum Disk Data Buffer for a File with 64-Byte Records



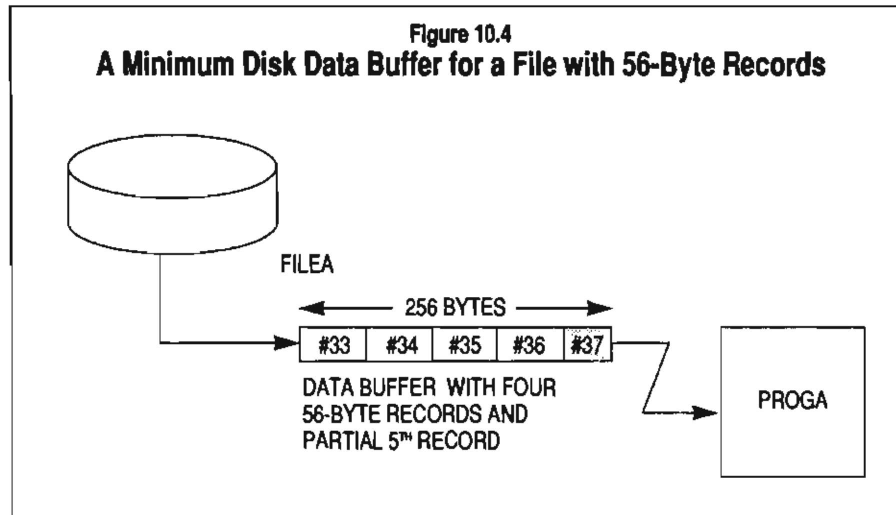
Filling this 100-record data buffer will certainly take longer than filling a one-record data buffer; but remember, at its slowest, the S/36 can transfer 1.2 MB of data per second from disk to main storage. To fill Figure 10.2's 100-record buffer using the slowest S/36 drive would take barely a second to transfer the 100 records); to read 100 records singly from disk would take at least 3.5 seconds! In this case, record blocking offers a five-to-one performance improvement! The example just presented showed blocking for a sequentially accessed file; other access methods may or may not fare well with record blocking, as you will see later in the chapter.

Figures 10.1 and 10.2 imply that the default data buffer for an unblocked file is one record. That is true for record lengths greater than 128 bytes. However, for record lengths of 128 bytes or less, DDM does a little "built-in" blocking for you. Recall that a disk sector, 256 bytes, is the smallest amount of disk storage the S/36 can read or write in a single disk operation. Therefore, a data buffer is always a minimum of 256 bytes. For record lengths of less than 256 bytes, that 256-byte buffer is filled with as many records as it can hold. For example, if your application is reading 64-byte records without using record blocking, each physical I/O actually puts four of the 64-byte records in the data buffer. Figure 10.3 illustrates the situation: When record 33 is read, DDM also puts records 34, 35, and 36 in the data buffer — even if your application hasn't explicitly enabled record blocking. No disk I/O is needed when your application reads records 34, 35, and 36. This is a tidy example because a 64-byte record length is a submultiple of 256. Let's take a look at a less-tidy example.

As explained in Chapter 3, records can span sector boundaries. For

Figure 10.4

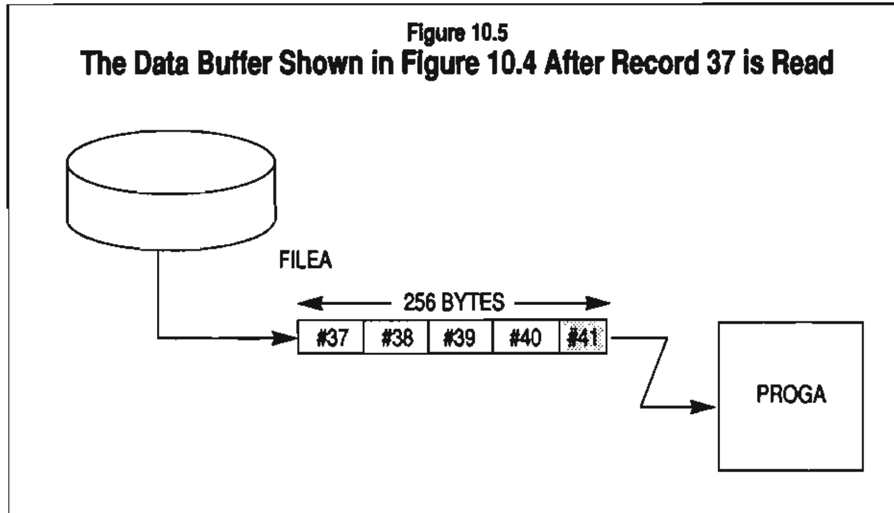
A Minimum Disk Data Buffer for a File with 56-Byte Records



example, a sector might contain four full records and only part of a fifth. Consider a non-blocked application sequentially reading a file with 56-byte records. As shown in Figure 10.4, the non-blocked disk buffer is still 256 bytes, but in this case four complete records and the first 32 bytes of a fifth record are in the buffer. The disk I/O that puts record 33 into the buffer will also put records 34, 35, 36 and the first 32 bytes of record 37 into the data buffer. DDM will make the four whole records in the buffer available to your application without any further I/O, ignoring any partial record contents in the data buffer. When the example application needs the partial record, a physical disk I/O operation will reread that record and refresh the buffer starting with that record. Figure 10.5 shows the data buffer after the program has read record 37.

Many programmers tuning the S/36 swear that record layouts must be submultiples or multiples of 256 (the size of one sector). These record sizes do provide for the most optimum usage of data buffers, but with appropriate record blocking and adequate memory, alternative record sizes don't significantly degrade performance. In extreme cases, say a record length of 129 bytes instead of 128 bytes, the time to read an *unblocked* file will double. If the record length were 128 bytes, two records would be read into the data buffer with one disk I/O operation — with a 129-byte record it will take one disk I/O operation per record (the partial 128 bytes will be wasted on every read). However, a 3,072-byte data buffer (12 sectors) could hold 23 of the 129-byte records, or 24 of the 128-byte records. With the S/36's fast data transfer rate, the time to fill that buffer is negligible. Effective record blocking changes the 129-byte record's performance from 50 percent to 95 percent of the "normal" record length's performance.

Figure 10.5
The Data Buffer Shown in Figure 10.4 After Record 37 is Read



Technical Note

Let your application needs dictate record length — not the myth that records should always be even submultiples or multiples of 256.

Enabling Record Blocking

For S/36 RPG applications, record blocking can be enabled in two ways:

- In columns 20-23 of the RPG F-spec (Figure 10.6). This method specifies the number of bytes to allocate for the data buffer and should always be a multiple of the record length. For example, if the record length is 256 bytes, you should specify a value of 512, 1024, 1536, 2048, and so on.
- With the DBLOCK keyword of the OCL FILE statement. Unlike the F-spec method, this method specifies the number of records to block, not the total size of the buffer desired. Figure 10.7 shows how you would use the DBLOCK keyword to block 10 records.

When you specify record blocking with the F-spec, the RPG program must be recompiled to change the record blocking factor. The OCL DBLOCK method is better because you can fine-tune record-blocking values without recompiling. Not only is it easier to change the blocking factors in your OCL, but it also provides more explicit control. DBLOCK lets you specify blocking factors that yield data buffers of up to 44 K; the F-spec limits you to 9,999-byte

Performance Tip

See Chapter 13 for a detailed discussion about using SMF to ensure that record and index blocking are helping, not hurting, performance.

Record Blocking Considerations

Before you enable record blocking, you should carefully consider several factors about the application. Among them are:

How is the file being read? If the file is a sequential or direct file being read sequentially or is an indexed file being read consecutively, it's likely a good candidate for record blocking. For randomly accessed files, you need to consider the "locality" of the records being accessed. Remember, the key to benefitting from record blocking is having the next record your application needs in the buffer, waiting for memory-to-memory transfer. When a file is processed randomly by key, say for an inquiry application, and there is little chance the next record read will be in the file buffer, blocking will hurt more than help. For indexed files that use mixed random and indexed-sequential processing, blocking may be of substantial value if the file is reasonably organized (i.e., the file has groups of records in key sequence). For example, an order-inquiry application might randomly access individual orders, but still find detail records for a given order in physical key sequence. By blocking the detail file large enough to accommodate the average number of detail records per order, the application can fetch all the detail records in a single disk I/O.

Some benchmarks presented later in this chapter will show how dramatically record blocking for a recently organized indexed file can affect performance. And Chapter 13, in its "cookbook" section, describes a method to directly measure the effectiveness of blocking for specific applications.

Is the application reading the file also updating the file? Do circumstances exist where another application can update the file? Figure 10.2 showed PROGA reading FILEA with a 100-record buffer. Imagine that PROGA just read the first record in FILEA — therefore filling its buffer with 100 records. For the next 99 records, PROGA is in turbo mode, zooming through the remaining data. However, what happens if another program updates the contents of record 39 just before PROGA is about to read it? PROGA now has a stale copy of this record in its data buffer!

Never fear: DDM won't let PROGA read obsolete data. Whenever a program updates a record in a shared file, DDM checks to see whether other programs using that file have the changed record in their buffers (the check is fast because DDM keeps a list of the disk address boundaries of all buffers in use for a file). In the example, upon finding that PROGA's buffer contained record 39, DDM would set a "dirty" flag on the buffer (technically called "marking the buffer invalid"). The next time PROGA tries to read from that file, DDM sees the dirty flag and refreshes the buffer's contents from disk storage, retrieving the updated record in the process. Because such collisions tend to

be rare, the dirty-buffer mechanism usually doesn't hurt performance with interactive programs that use blocking.

The situation is different for batch programs. If both the other program and PROGA are updating the file sequentially, the two programs could find themselves in "lockstep," requiring buffer refreshes with each read! The potential performance improvement that record blocking offers could become an enormous performance penalty in such situations. Avoid the batch stale-buffer problem by not blocking files when more than one program is reading sequentially for update.

Is the file opened for add or output-only? For sequential, direct, or indexed output files, record blocking will be beneficial. Just as blocking helps defer disk input operations, it can also defer disk output operations. With buffered output, records are not written to disk until the buffer is full — and the S/36's fast transfer rate ensures that writing one large block is much faster than writing many individual records. Even for indexed files with records added out of key sequence, the records themselves still get written in arrival order, so blocking the output file helps.

One especially good candidate for record blocking is reading #GSORT addrout files. A 4 K buffer can hold more than 1,300 of an addrout's 3-byte binary record addresses. When addrout files are appropriately blocked, the performance improvement is significant. Other good candidates for record blocking include:

- Random reads where the records cluster close together. For example, all detail records associated with a picking ticket, indexed on a transaction number. In this case, the blocking factor should equal the average number of details associated with a picking ticket.
- Multiple records to be read and displayed interactively. For example, invoice details from a sequential history file. In this case, the blocking factor should be the same as the number of lines displayed interactively.
- Duplicate key processing. The blocking factor should be equal to the average number of duplicate key values being processed.

Index Blocking

Index blocking lets your application keep many index entries in memory at once, reducing disk I/O to the file's index area. Chapter 3 discussed the "lookup" nature of the S/36 random record retrieval system. Given a key value, DDM first looks up that key value in a table (the index) to retrieve the associated relative record number (RRN), and then uses that RRN to read the data record. Index blocking loads multiple index entries into an *index buffer*

Performance Tip

For small indexed files it's possible to use index blocking to put the entire file's index in memory. With this method, random record access from such a file could achieve direct file-like performance.

in the same way record blocking loads multiple records into a data buffer.

The resulting reduction in disk I/O to the index area helps in two ways. First, overall system performance improves as a result of fewer total I/O operations. Second, application performance improves as a result of reduced disk arm motion between the index and data areas for a particular file. Because every indexed access is always followed by a data access, applications that don't use blocking tend to suffer from a "see-saw" effect: move the disk arm to the index area and read a key, move the arm to the data area and read a record, move the arm back to the index area to read another key, and so on. If enough keys can be kept in memory to reduce repeated returns to the index area, the disk arm is more likely to be in position over the data area for the next record operation. Because all index blocking concepts apply to alternate indexes as well as to a file's primary index, the disk arm might well have to move a considerable distance from index to data.

Recall that index entries are stored in 256-byte sectors. Index entries consist of the key value paired with a 3-byte RRN pointing to the associated data record. The number of index entries per sector depends on the key length. Unlike data records, however, index entries *do not* span sector boundaries. Let's take an example (Figure 10.9) where key values are 11 bytes long; therefore, each index entry is 14 bytes long (11 bytes plus the 3-byte binary relative record number). For 11-byte key values a sector holds 18 index entries. These 18 key entries total 252 bytes; the remaining four bytes of the sector are empty.

When an application requests one of the keys in Figure 10.9's sector, DDM reads that entire sector into memory (because one sector is the least amount of data the S/36 can read at a time). Accessing any other key in that sector is done without disk I/O. By default, one sector full of keys is the minimum your application program will read. But you can explicitly request that more sectors be blocked. Unlike DBLOCK, IBLOCK can be specified only in OCL, using the FILE statement's IBLOCK keyword. Figure 10.10 shows the IBLOCK keyword being used to block 72 keys, which would make four sectors of "memory resident" keys available to an application. Because a disk seek to the index area is eliminated, random access to any record with a key in this buffer will require half the time a random read normally takes. If the records were also blocked, *no* disk I/O would be required to read the record. (Coordinating the efforts of DBLOCK and IBLOCK are covered in more detail later in this chapter.)

Sizing Index Buffers

As with blocked records, the potential performance improvement offered by blocking index entries comes at a cost. The memory available for index buffers is limited; if you specify too much index buffering, disk I/O could increase somewhere else and performance will suffer, not improve. Figuring proper IBLOCK values would have been much easier if the IBLOCK parameter

Figure 10.9
One Sector Full of Keys

Key	RRN	Key	RRN	Key	RRN	
Key	RRN	Key	RRN	Key	RRN	W
Key	RRN	Key	RRN	Key	RRN	a
Key	RRN	Key	RRN	Key	RRN	s
Key	RRN	Key	RRN	Key	RRN	t
Key	RRN	Key	RRN	Key	RRN	e

Eighteen 11-byte keys (18 x 14 = 252) 4 bytes in sector wasted.

Figure 10.10
OCL Showing IBLOCK Keyword

```
// LOAD PROGA
// FILE NAME-AFILE,DBLOCK-10,IBLOCK-72
// RUN
```

accepted the number of bytes to block, instead of the number of keys. As it is, to determine the size of the index buffer an IBLOCK specifies, you must:

- Calculate the number of index entries per sector
- Determine the number of sectors requested
- Multiply the number of sectors requested by 256

Let's look at an example using an IBLOCK value of 200 with a 16-byte key. First, calculate the number of index entries per sector:

256 / (16 + 3) = 13.47

There are 13 16-byte keys in a sector — the remainder is discarded. Next, determine the number of index sectors requested by dividing the IBLOCK value by the index entries per sector:

200 / 13 = 15.38

That value, rounded to the next whole number, reveals that 16 sectors need to be read to fill an index buffer with the IBLOCK number of keys specified. To

Figure 10.11

IBLOCK Factors for Desired Index Buffer Size

Find key length in left column and desired IBLOCK factor in shaded area.

Desired IBLOCK buffer size in K

Key Length	Keys Per Sector	1 K	2 K	4 K	8 K	10 K
1	64	256	512	1024	2048	2560
2	51	204	408	816	1632	2040
3	42	168	336	672	1344	1680
4	36	144	288	576	1152	1440
5	32	128	256	512	1024	1280
6	28	112	224	448	896	1120
7	25	100	200	400	800	1000
8	23	92	184	368	736	920
9	21	84	168	336	672	840
10	19	76	152	304	608	760
11	18	72	144	288	576	720
12	17	68	136	272	544	680
13	16	64	128	256	512	640
14	15	60	120	240	480	600
15	14	56	112	224	448	560
16	13	52	104	208	416	520
17-18	12	48	96	192	384	480
19-20	11	44	88	176	352	440
21-22	10	40	80	180	320	400
23-25	9	36	72	144	288	360
26-29	8	32	64	128	256	320
30-33	7	28	56	112	224	280
34-39	6	24	48	96	192	240
40-48	5	20	40	80	160	200
49-61	4	16	32	64	128	160
62-82	3	12	24	48	96	120
83-120	2	8	16	32	64	80

determine the byte size of that index buffer, multiply the number of sectors to read by 256:

16 X 256 = 4,096-byte buffer

This reveals that using an IBLOCK value of 200 with a file with a 16-byte key creates a 4 K index buffer.

You may agree that it would have been easier to specify the IBLOCK value in bytes. You can avoid this math madness by using the chart shown in Figure 10.11. This chart shows IBLOCK values for 1 K, 2 K, 4 K, 8 K, and 10 K index buffers for files with any size key. To use the chart, look up the key length in the left column and follow the row across for the IBLOCK value. For example, to allocate a 4 K index buffer for a file with a 16-byte key length, the IBLOCK should be 208. For a 27-byte key with a 2 K buffer, use IBLOCK-64. And for a 33-byte key with an 8 K buffer, use IBLOCK-224. The number of keys per sector is included in the table for informational purposes only. Use this chart as a guideline — if main storage isn't available, or the total size of record and index buffers for a given file exceeds 44 K, the system may allocate less buffer space than you requested.

Technical Note

Use the table in Figure 10.11 to easily determine the IBLOCK value required to achieve a certain index buffer size.

Index Blocking Considerations

Blocking index entries can also substantially decrease application response time or batch program execution time. However, as with record blocking, there are certain considerations to keep in mind. Among them are:

Is the file being read in key sequence — and has the file been keysorted recently? Recall from Chapter 3 that indexes, unlike the actual data records, must be maintained in key sequence. If a file has recently had a true KEYSORT performed on it (see Chapter 3 for more about a true KEYSORT), its index will be free from overflow keys and performance will be increased by blocking many index entries for a file read sequentially by key. If a file has not had a true KEYSORT performed on it recently, and if many index entries are in the overflow area, index blocking is not likely to improve performance. Don't confuse physical data record order with the index being in order. As long as the index has been KEYSORTed recently (to purge index entries from the overflow area), your application will benefit from index blocking. It doesn't make any difference whether or not the data file has been reorganized recently into physical key sequence. Remember that index blocking puts a chunk of index entries in memory; it doesn't affect

IBLOCK's performance if those index entries point to non-contiguous RRN's in the actual data file. Of importance to your application is that the index entries, with their associated RRNs, are in memory. The Index Doctor utility (Chapter 11) lets you directly determine the existence and size of an overflow area for a given file or alternate index.

Is your application randomly adding many keys to an index opened for shared access? When records are added to shared indexed files or alternate indexes, DDM maintains key values in the index overflow area in key sequence. If your application is adding records in random key sequence to a shared file, IBLOCK is not likely to improve performance. However, if your application has exclusive use of the output file, DDM will not attempt to keep the index overflow area in key sequence; it will simply dump new keys at the end of the index buffer, writing the buffer to the overflow index whenever it fills. In this case blocked index entries — even in random order — may improve performance. A subsequent KEYSORT (either explicitly or automatically initiated by SSP when the next program opens the index) will clean up the out-of-sort entries (see Chapter 3 for more on this topic). In this case, set IBLOCK as high as you can or to the maximum number of records being added. Don't forget about the "hidden" way keys are added to an index — when updating a file having alternate indexes. If you change part of a data record that happens to be an alternate-index key, DDM deletes the old key and adds a new one to reflect the changed data. For shared alternate indexes, IBLOCK will degrade performance; for unshared alternates, IBLOCK can be a big help.

Can you block the entire file's index? For small files, you can often achieve spectacular random access performance if you block the entire file's index. Consider an indexed salesperson file for a point-of-sale application. For each invoice written, the salesperson file must be read, but it will be read randomly because there is no way to predict who wrote the next invoice. The entire index for a 250-record salesperson file having a 4-byte key can be buffered in less than 2 K (250 x (4 + 3) = 1,750 bytes). Beware, though, that whenever a new salesman record is added or has its key changed, the application's index buffer will need to be refreshed. Use this technique only with files maintained infrequently.

Is your application reading sequentially by key an indexed file recently organized in the same key sequence? If so, a combination of DBLOCK and IBLOCK could really improve performance. A large index buffer coupled with a large data buffer is a terrific combination when you

expect that both index and data requests will occur in physical sequence. Use DBLOCK and IBLOCK together carefully, though. If the data file has not been organized in key sequence recently, the disk thrashing required to continually refill the data buffer with the correct records could severely reduce performance.

Where Buffers Live

Although DDM keeps buffers in memory, it doesn't always keep them in the same virtual region as your program. If all the space in your program's 64 K region is used up (by the program itself, for example), DDM puts the buffers in a *Task Work Space (TWS)* that provides an additional file buffer space. But because TWS-resident buffers aren't part of the same address space as your program, DDM must jump through some extra hoops to access them. The result is slower record and key retrieval, and consequent blunting of blocking's advantages. The size of record and index buffers determines where DDM keeps them.

When a program initiates, DDM allocates data and index buffers dynamically, in the order in which the files are listed in the F-specs of the RPG program (not in the order in which they are listed in the OCL). DDM starts placing buffers in memory immediately after the end of the program. As each buffer is allocated, DDM checks to see if it will fit in the remaining memory (up to the 64 K region size). If it won't, DDM creates the TWS and allocates that buffer there. All subsequent buffers also end up in TWS. You can see that the best-case scenario is for the application program and all its buffers to fit within the S/36's notorious 64 K region size.

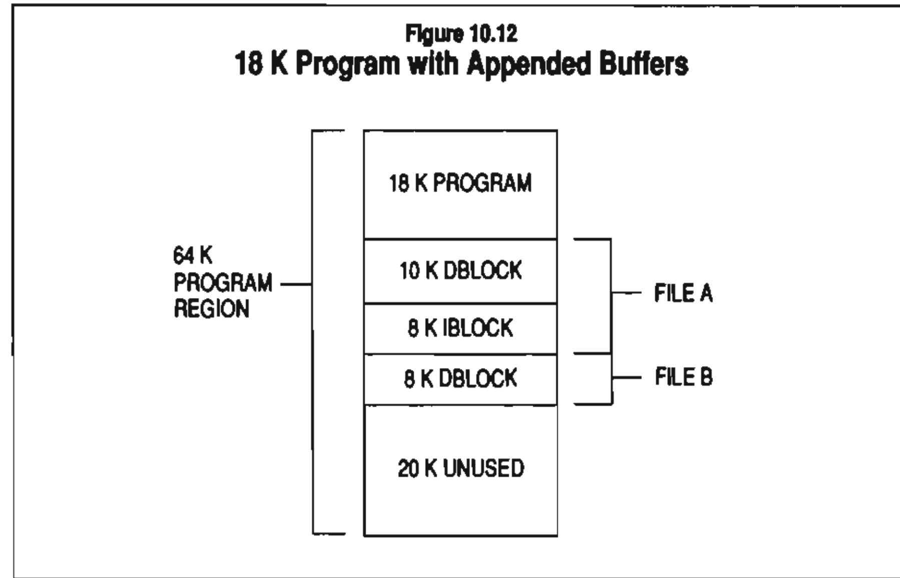
Figure 10.12 shows an 18 K program using two files. FILEA (opened first) has a 10 K record blocking buffer and an 8 K index blocking buffer; FILEB has an 8 K record blocking buffer. In this case, the total size of the program and the buffers (44 K) fits within the 64 K maximum region size. These buffers are called *appended buffers* because they are appended to the program's region. You get the best results with blocking when all buffers are appended within the program's region, as Figure 10.12 shows.

Figure 10.13 shows a 26 K program, using three files. In this case, FILEA has a 12 K appended record blocking buffer and a 6 K appended index blocking buffer; FILEB (specified second in the RPG program) has a 16 K appended record blocking buffer. After opening FILEA and FILEB, the program's partition has used 60 K of its available 64 K. However, FILEC (specified third in the RPG program, and opened last) has an 18 K record blocking buffer, which won't fit in the remaining 4 K. DDM must create a TWS to hold the oversized buffer. Using TWS will decrease performance significantly because DDM must change addressability — a process called *mapping* — to access the non-appended bufffers. If your program reads records alternately from appended and unappended buffers, the extra overhead for mapping can

Performance Tip

If the total size of an RPG program and its buffers exceeds 64 K, you might improve performance by opening sequentially processed files first to place their buffers within the program's 64 K region. Do this by listing the sequential files *first* in the RPG F-specs. If your S/36 has plenty of main storage, the impact of randomly processed files with non-appended buffers affects performance less than sequentially processed files with non-appended buffers. Generally, though, your goal should be to have no non-appended buffers.

Figure 10.12
18 K Program with Appended Buffers



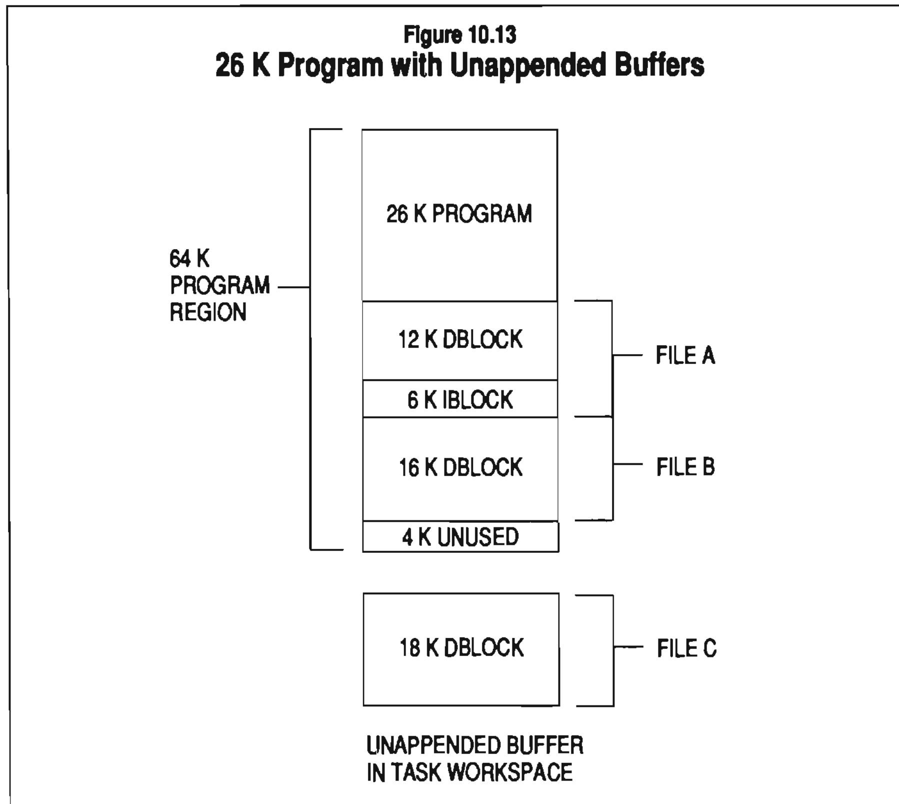
overcome completely any performance advantage gained by blocking. As a general rule, if record or index blocking causes the creation of TWS buffers, they are not worth the resulting disk activity and the related DBLOCK and/or IBLOCK values should be reduced. You can see if a running program has TWS buffers by checking the "BUFF" column on the STATUS USERS display. Any entry in this column indicates TWS buffers.

If Figure 10.13's FILEB was a randomly accessed file and FILEC was a sequentially accessed file, it might be possible to improve performance by swapping the order in which those files are defined in the RPG program. This would force FILEC's buffer to the program region and make FILEB's buffer the one that goes into TWS. But rather than juggle file order to manage record buffering, you probably will find it of more value to reduce all the DBLOCK and IBLOCK factors involved to ensure that *all* buffers are appended to the program region.

Allocating Appropriate Buffer Sizes

As you can see, it's very important to allocate reasonable buffer sizes. Before you start experimenting with DBLOCK and IBLOCK factors for a program, determine the program's existing, non-blocked memory utilization. To ensure that the program is at record and index blocking ground-zero, remove any record blocking values in columns 20-23 of the F-spec and recompile the program. Also ensure that no DBLOCK or IBLOCK values currently exist in the program's OCL. With the non-blocked program running, use the STATUS

Figure 10.13
26 K Program with Unappended Buffers



USERS (D U) control command from another workstation to determine the program's non-blocked, bare-bones memory utilization. A screen like the one shown in Figure 10.14 will be displayed. On this screen, the third column from the right, the RGN column, shows the program size. The PGM column, the second from the left, shows the total of the program region and any appended buffers. The last column on the right shows the amount of TWS allocated for unappended buffers.

Let's assume we're performance tuning the program NEWKT7, called by procedure POS1. In this case, the STATUS USERS screen shows that the program uses a non-blocked 30 K program region size. This program has 34 K for which record and index buffers can be allocated before any buffer is assigned to TWS. Using the record and index buffer sizing information from earlier in this chapter, you could now start to performance tune NEWTK7 by specifying various DBLOCK and IBLOCK factors to allocate appropriately sized buffers. Each time you make a change, run the program and check the STATUS USERS screen again.

Figure 10.14
Status Users Screen

```
Complete                                                   STATUS USERS
                                W4

JOB       PROC      PROGRAM  STATUS   ATTRIBUTES  PRTY  RGN  PGM  BUFF
W8192239  POS1      NEWTK7   Active      SRT      Norm  30K  30K  —
W5132557  POS2      NEWTK7   Active      SRT      Norm  30K  62K  —
W9313158  POS3      NEWTK7   Active      SRT      Norm  30K  42K  (34K)

                                           Unappended Buffer ————┘

 Cmd7-End     Cmd8-Help     Cmd15-Update     Cmd16-Restart     Roll-Page

                                JOBS
                    Control jobs and the job queue
   1. Display a specific job          6. Change processing priority
   2. Jobs on the job queue
   3. Stop a job
   4. Restart a stopped job           9. Prevent SSP-ICF jobs
   5. Cancel a job                   10. Allow SSP-ICF jobs to start

 Ready for option number or command

                                                 COPR IBM Corp. 1983
```

The second line of Figure 10.14 shows POS2, a copy of procedure POS1 with some blocking factors. In this case, 32 K of buffers have been allocated, and because they fit within the program's 64 K region, no appended buffers are created. Notice how the PGM column for POS2 has increased to 62 K, the total of the 30 K program size and the 32 K of file buffers. The third procedure, POS3, shows what happens when you get carried away. Here, 12 K of buffers did fit within the program region (as indicated by the PGM size of 42 K), but 34 K of buffers did not and a non-appended buffer was created in a TWS. Performance will almost certainly suffer for procedure POS3. Keep a close eye on the STATUS USERS screen and try not to let any task acquire appended buffers because DBLOCK and IBLOCK factors are too large.

Technical Note

If you use external program calls from any vendor, beware that data buffers for called programs are allocated in Task Work Space and therefore always show on the STATUS USERS screen in the BUFF column.

Benchmarks

Here are a few benchmarks showing the upside (performance gains) as well as the downside (performance degradation) possible with record blocking. These tests were performed using a program needing a 34 K region and reading a 5,000-record, 256-byte record-length file sequentially. In each case, the tests were performed on a similarly burdened S/36:

DBLOCK-1	404 seconds
DBLOCK-50	30 seconds
DBLOCK-93	26 seconds
DBLOCK-100	356 seconds

Note the difference between 93 records and 100 records. Buffering 100 records required an unappended buffer and ruined performance.

To read an indexed, 5,000-record, 256-byte record-length file sequentially by key:

DBLOCK-25	IBLOCK-100	619 seconds

After organizing the file:

DBLOCK-25	IBLOCK-100	33 seconds

Notice what a difference reorganizing the file made. Frequent file reorganization and prudent use of DBLOCK and IBLOCK can really speed sequential, by-key processing.

Technical Note

Use the DBLOCK keyword of the OCL FILE statement to enable record blocking — do not use columns 20-23 of RPG's F-spec. DBLOCK doesn't require recompiling the program to change blocking.

Mental Blocking

Proper record and index blocking can give you vastly improved performance; improper blocking can destroy performance. The only way to know for sure whether blocking is helping or hurting is to think through your blocking strategy in light of the facts presented in this chapter — and then measure performance (see Chapter 13) to confirm that you're getting expected results. Achieving good results is a mental exercise, not a random act. Keep in mind that blocking is most useful for standalone sequential batch accesses, and only somewhat useful in heavy interactive or multibatch environments. You should

explore disk caching (see Chapter 15) — a sort of system-wide blocking factor — as an alternative when blocking individual programs doesn't give you the results you want.

Chapter 11

Prescriptions for Healthy DDM

As we've mentioned elsewhere in this book, Disk Data Management (DDM) performance often leaves a lot to be desired. Previous chapters have explained some of DDM's crippling affects and warned of their impact on performance. In this chapter, we'll look at strategies and utilities you can use to shore up some of DDM's weaknesses. You're not likely to need all the tips and techniques presented in this chapter in any one application. Before you jump in and start implementing anything shown here, it will help to have read or at least skimmed some of the earlier chapters that explain S/36 DDM and its limitations. With just a brief understanding of DDM limitations, the tips and strategies in this chapter will make more sense and be easier to implement.

When squeezing the most out of your S/36, the adage "the best defense is a good offense" really applies to DDM. By knowing where DDM is weak, and how to code around those weaknesses, you'll really ramp up performance. In this chapter we present three tools to help you circumvent DDM weaknesses: Index Doctor checks out indexed files for performance-robbing conditions; KEEPOPEN holds storage indexes open to reduce program initiation time; SHOWUR reveals deadly embraces that can shut down interactive applications.

Is There a Doctor in the House?

Chapter 3 covered extensively the detrimental effect that the S/36's method of random record access has on performance. There, we discussed the primary and overflow areas of an indexed file, and we warned you that how DDM processes the index overflow area can sometimes impede performance drastically. To know what action to take to avoid DDM's problems, you need a way to diagnose an index's general health. That's where Index Doctor comes in. Index Doctor (included on the "Desktop Guide" diskette) is a utility that analyzes S/36 indexed files (parents and alternates) for deleted keys, duplicate key strings, and index gaps. After diagnosing the index, Index Doctor prints a report of its findings. Based on Index Doctor's diagnosis, you'll know if a file should be organized, keysorted, or perhaps even re-allocated to a larger size.

Running Index Doctor is a simple matter of calling procedure INDEXDR. The procedure has four parameters, the first of which is required; the remaining three are optional. The first parameter is the name of the file you want to analyze. The second parameter is an optional file date. Index Doctor's calling sequence is:

```
INDEXDR file name,[file date],[Y/N],[Y/N]
```

remove the offending deleted keys.

After printing its primary index analysis, Index Doctor prints its overflow index analysis. Again, duplicate key strings are reported, as well as index gaps. (Remember that index gaps are "holes" that DDM places in the overflow to minimize the number of keys that must be moved down when a new key is added. See Chapter 3 for a compete explanation of index gaps.) The index gap detail line shows the relative key offset of the gap from the beginning of the overflow ("Index gap at:"), the number of keys that can fit in the gap ("Count:"), and the key immediately preceding the gap ("Key:").

Following the duplicate key and gap detail lines on the report are the total dup key strings, total gap keys, and total keys in the overflow. Deletes are omitted from the overflow index report because a deleted key in the overflow index is looked upon by DDM as a gap, and Index Doctor will report it as such. This also means that deleted key locations in the overflow index can be reused, whereas in the primary index they cannot because, as you may recall, keys are never added to the primary index.

Technical Note

Force a *real* keysort by using "KEYSORT filename,,,CHKDUP".

Taking Action

Here are some things to notice and possible action steps to take after reviewing an Index Doctor report. Most (but not all) of the following diagnoses are taken from the sample report in Figure 11.1.

Diagnosis 1: There is only one key slot available in the last gap. Adding a record with the key value "8070Y" won't cause a problem — a slot is available for it. But then adding "8070Z" will cause a pre-emptive keysort to degap the file. Subsequent record adds will cause the ripple-add effect.

Action: Force a real keysort on the file (KEYSORT filename,,,CHKDUP — as discussed in Chapter 3), reorganize the file, or enlarge the file.

Diagnosis 2: At the most, there are only 31 key slots available in any of the overflow gaps. If more than 31 records are added to the file with key values greater than "8012L," a pre-emptive keysort will be called to degap the file and subsequent adds will be very slow.

Action: Force a real keysort on the file or reorganize it. Also consider presorting input records to be added to this file in key sequence before adding them.

Diagnosis 3: There are few key slots available in the last gap (in this case, only one). This could indicate the file has not been allocated large enough.

Action: Consider reallocating the file with a larger size. This won't guarantee more gaps after subsequent adds, but it will probably help.

Diagnosis 4: There are large duplicate key strings.

Action: If possible, the file and applications should be modified to minimize the length of duplicate key strings. See Chapter 3 for a detailed discussion about the impact of duplicate keys and how to avoid them.

Diagnosis 5: When the "Keysort index" status is "yes," it indicates that the file may need to be keysorted. This is caused by many records being added to the file, resulting in a large overflow index area.

Action: Perform a real keysort or organize the file. If you notice this continually happening for a given file, consider performing a real keysort every night for the offending file.

Diagnosis 6: The file contains deleted keys but is not currently delete capable.

Action: Reorganize the file to remove deleted keys.

Diagnosis 7: The file contains duplicate keys but is not currently duplicate-key-capable.

Action: Reorganize the file to remove the duplicate keys or use COPYDATA to copy the file to a duplicate-key-capable file.

You don't need to use Index Doctor daily for every file, but consider using it once or twice a month on your large indexed files, just as a "check-up." By taking the recommended action steps, you'll increase interactive response time, reduce the time required for random indexed adds, and decrease job initiation time.

Keeping Storage Indexes Open

For every indexed file your applications use, the system must scan each file's entire index and build a storage index. Recall from Chapter 3 that a storage index is an index to the index. The storage index's table of index values and track numbers is used to narrow the index search down to the appropriate track. With a storage index, the system scans only the appropriate track looking for key values, which dramatically improves random record retrieval times.

However, using storage indexes has a cost: slower job initiation. For each indexed file your application uses — not already opened by another program — the system must scan the entire index to build a storage index. Building

storage indexes over and over again for the same large indexed files throughout the day results in seemingly random delays in program initiation and erratic response times. You can rectify this situation by forcing all frequently accessed indexed files to remain open throughout the day.

Remember that DDM keeps storage indexes in the System Queue Space — part of the non-swappable nucleus — and that all users of the file share the same storage index. Figure 11.2a shows PROGA using FILEA. When PROGA started, no other user was currently using FILEA and therefore there was currently no storage index in memory for FILEA. PROGA's initiation required that the storage index shown in Figure 11.2a be built. Figure 11.2b shows that when PROGB (which also uses FILEA) initiates, it will share the storage index created by PROGA's initiation. PROGB will initiate faster because the storage index it needs is already in memory. You can see that no user actually "owns" a storage index. Only the first user must endure the delay, caused when the storage index is built. The problem is that when the last program using the file ends, DDM discards the storage index. Then the next program to open the file must wait while DDM rebuilds a new storage index.

The trick, then, is to achieve persistent storage indexes by tricking oft-used, large indexed files into remaining open all day long. For example, let's say you had three large files you'd like to keep open all day so their storage indexes would persist for all users. One solution is to evoke a procedure like the following:

```
// FILE NAME-APTRANS,DISP-SHRRM,JOB-YES
// FILE NAME-APVEND,DISP-SHRRM,JOB-YES
// FILE NAME-CUMASTER,DISP-SHRRM,JOB-YES
// WAIT INTERVAL-080000
```

One limitation is that this method will neither open the files nor cause the storage indexes to be created. It will simply hold the files open for eight hours after the first real application opens them. Job initiation for the first user of each file will not be improved. A greater limitation is that this method can only be stopped interactively from the system console. Should any other application require exclusive use of any of these three files, operator intervention at the system console is required.

Another way to keep files open is to write a customized MRT-NEP program — a Multiple Requester Terminal program that has its Never-Ending-Program attribute set — with F-specs and a dummy read operation for each file you want to keep open all day. Unlike an ordinary Single-Requester-Terminal (SRT) program, which is not capable of releasing the requesting display station, a MRT-NEP program can release its requester and remain active. The MRT-NEP does not tie up a workstation or cause the system to perform unnecessary processing because once activated, the MRT-NEP remains in a suspended state.

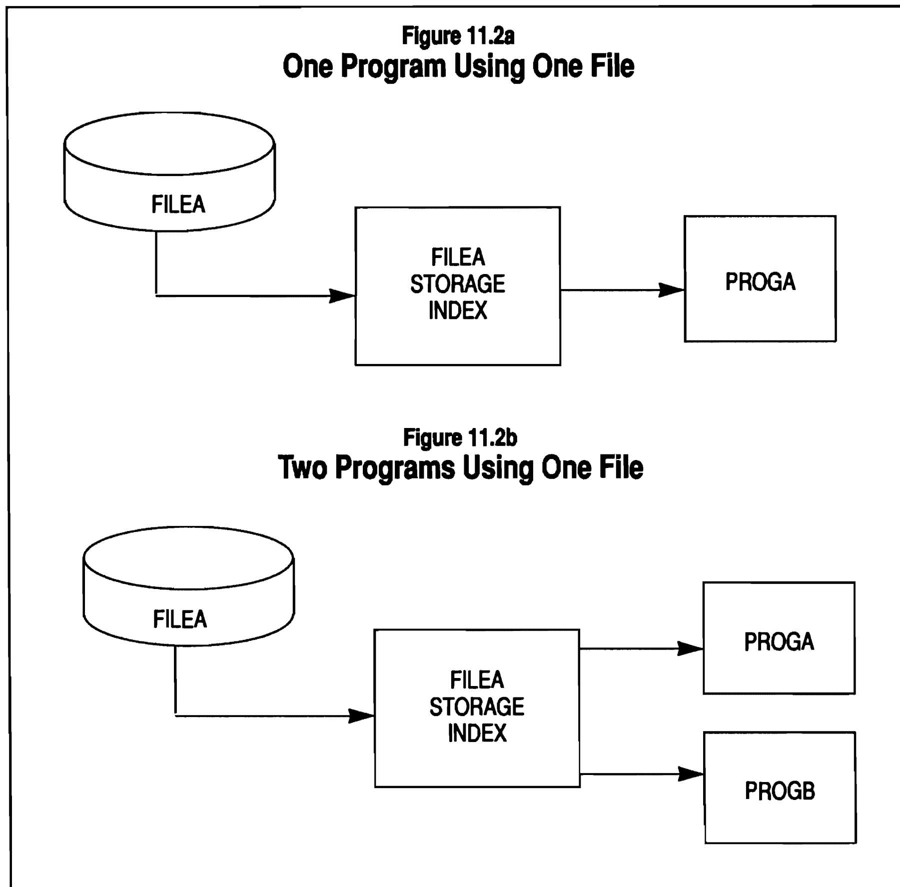


Figure 11.2a
One Program Using One File

Figure 11.2b
Two Programs Using One File

This method could be canceled from any workstation, but it requires recompiling the program each time you need to modify the list of files you want kept open. You're also limited to only 15 files per MRT program. To keep more than 15 files open would require additional MRT programs.

A better way to keep often-used indexed files open is with a combination of the two techniques. This method requires three short procedures and one short MRT-NEP program. Figure 11.3a shows the main procedure, KEEPOPEN. Initiating KEEPOPEN causes each referenced file to be opened (and remain open) and their storage indexes to persist as long as necessary. KEEPOPEN should include the following code for each file you want to keep open:

```
// FILE NAME-filename,DISP-SHRRM,JOB-YES
KOPENF filename,[storage index size in K]
```

Figure 11.3a
Procedure KEEPOPEN

```
*----------------------------------------
* Allocate specified files as JOB-YES and open the
* storage index for each one.
*----------------------------------------
* Insert a pair of lines as shown for each file you
* to keep open all day long
*
// FILE NAME-APTRANS,DISP-SHRRM,JOB-YES
KOPENF APTRANS
*
// FILE NAME-APVEND,DISP-SHRRM,JOB-YES
KOPENF APVEND
*
// FILE NAME-CUMASTER,DISP-SHRRM,JOB-YES
KOPENF CUMASTER
*
KPOPEN2
```

Figure 11.3b
Procedure KOPENF

```
// LOAD $COPY
// FILE NAME-COPYIN,LABEL-?1?,DISP-SHRRM,STORINDX-?2'YES'?
// FILE NAME-COPYO,LABEL-COPYTEMP,RECORDS-1,RETAIN-S
// RUN
// COPYFILE OUTPUT-DISK
// SELECT KEY,FROM-'¢',TO-'¢'
// END
```

Figure 11.3c
MRT Procedure KPOPEN2

```
// LOAD KPOPEN
// RUN
```

The first line uses the // FILE statement's JOB-YES parameter to enable the files to stay open after KEEPOPEN ends and the second line calls procedure KOPENF (Figure 11.3b). KOPENF uses $COPY to cause the specified file to be opened and a storage index created for it. The cent-sign character is used as the select key value because it is unlikely to be part of a character key and it cannot be part of a packed key. Using $COPY to force the file open is necessary

Figure 11.3d

MRT-NEP Program KPOPEN

```
H                                                                KPOPEN
F* +-----------------------------------------------------+
F* |                                                      |
F* |              I N D I C A T O R   U S A G E           |
F* |                                                      |
F* |   01  Blank input record, used when starting the program   |
F* |   02  Non-blank input record, used to cancel the program   |
F* |                                                      |
F* +-----------------------------------------------------+
FSCREEN  CP  F  80  80            WORKSTN
F                                             KFMTS   *NONE
ISCREEN  NS  01    1 C
I        OR  02
I*
C* +-----------------------------------------------------+
C* | A non-blank input record causes the program to be cancelled.|
C* +-----------------------------------------------------+
C          02          SETON                     LR
O* +-----------------------------------------------------+
O* | Release the requesting workstation.                  |
O* +-----------------------------------------------------+
OSCREEN  DR       01
```

because the // FILE statement alone does not cause a storage index to be built.

After the// FILE statement and call to KOPENF for each file in KEEP-OPEN, KEEPOPEN calls the MRT procedure KPOPEN2 (Figure 11.3c), which uses the MRT-NEP program KPOPEN (Figure 11.3d). KEEPOPEN ends after the call to KPOPEN2 but KPOPEN causes the job, and therefore the JOB-YES attribute, to persist until KPOPEN ends. As long as the MRT-NEP program KPOPEN is active, the files specified in KEEPOPEN remain open and their storage indexes persist.

Now comes the tricky part. There might be times during the day when an application needs exclusive use of a file being held open — say, to reorganize the file. You need a graceful way to end the MRT-NEP program KPOPEN from any workstation.

How do you end KPOPEN? By calling the KPOPEN2 procedure with any parameter value. Any data following the procedure name that initiates a MRT program can be read by that MRT program as its first input record. If KPOPEN2 were called with KPOPEN2 NOW IS THE TIME, the characters NOW IS THE TIME are passed to the KPOPEN program as the first input record. This technique will be used to end the KPOPEN program on demand. The scheme is simple: A blank first input record (a call to KPOPEN2 without any parameters) starts the program, and a non-blank record (a subsequent call to KPOPEN2 with any parameter value), ends the program. To end KPOPEN

from any workstation or any other procedure, simply use the line "KPOPEN2 CANCEL". "CANCEL" gets passed as a non-blank record to MRT program KPOPEN, causing it to end and the files being held open to close. Note that any value would work; "CANCEL" is used to aid readability.

The primary file in program KPOPEN is a WORKSTN file. KPOPEN does not read or write to the workstation file; all input for the workstation program actually comes from data passed as the first input record by the MRT procedure. The program always processes exactly one input record and releases the requester after handling this input record. Because the program never reads or writes to the workstation device, you don't need to define a screen format member; thus, in the F-specs, you code a KFMTS continuation line specifying *NONE.

Technical Note

To close all files being held open by the MRT program KPOPEN, end KPOPEN from any workstation or any other procedure by calling procedure KPOPEN2 with a first parameter value of CANCEL (KPOPEN2 CANCEL). This causes the MRT program KPOPEN, to end, which closes all files that are specified in procedure KEEPOPEN to be closed and causes their storage indexes to go away.

Because all existing references to files being kept open must allow file sharing, you may have to change file dispositions in a few existing procedures. If you have files that can't be shared, you can either modify the existing FILE statement to allow sharing, or, if the application absolutely requires dedicated use of a file, you can add the necessary OCL statements to cancel the KEEPOPEN procedure before continuing (e.g., adding // IF ACTIVE KEEPOPEN KOPEN2 CANCEL). If you have many programs that do not allow file sharing for large indexed files, you may need to make a lot of changes to your FILE OCL statements. But the performance improvements this technique provides are worth the effort.

You can optionally override the SSP default storage index size by specifying the maximum size of the storage index desired as the second parameter to the KOPENF procedure. For example,

```
KOPENF WILMA,16
```

would request a 16 K storage index for file WILMA. The maximum storage index size must be a number from one through 16. Based on memory availability, the SSP will attempt to use this value to allocate that size storage index,

but the actual index created may be smaller if enough memory isn't available. Unlike DBLOCK and IBLOCK record buffers, memory allocated for the storage index is never placed in a task work space; rather, storage indexes are always in the System Queue Space of the variable nucleus.

Technical Note

If you have one application that runs all day, that application might be doing what KEEP-OPEN would do: hold your largest indexed files open — causing their storage indexes to be available to all subsequent users. Typical applications like this include inquiries running on dedicated workstations or point-of-sale programs that run all day. Although chances are that no one application keeps all of your large indexed files open, examine which applications run all day and which files they keep open before implementing KEEPOPEN.

The Proof Is in the Performance

Figure 11.4 shows benchmark results of using KEEPOPEN on a dedicated S/36 5360 Model D with a frequently used interactive program that references a large indexed file (630,000 records) that has one alternate index. The program also references other, smaller files. When not using the KEEPOPEN technique, it takes about 22 seconds to initiate the interactive program on a dedicated system; if the system is being used by other jobs, the program takes approximately 47 seconds to initiate. With the KEEPOPEN technique, it takes less than two seconds to initiate the program on a dedicated system; on a non-dedicated system, initiation time is less than three seconds. If all the indexed files used by the program are already open, initiation time is less than one second on a dedicated system, and less than two seconds on a non-dedicated system.

Using KEEPOPEN saves 21 seconds on a dedicated system and 45 seconds on a typically loaded system. When you multiply the number of large indexed files on your system by the number of times you open those files each day, KEEPOPEN adds up to significant time savings with little programming effort.

Avoiding the Deadly Embrace

Even with the deficiencies we've discussed throughout this book, S/36's DDM deserves a lot of credit. While it's true that DDM is occasionally much slower than we'd like, it is *very* reliable, and it's easy to take that reliability for granted. Yet, as robust as DDM is, there are still times when a little defensive coding can avoid record contention problems. Consider two interactive programs

Figure 11.4
KEEPOPEN Performance Benchmarks

Conditions	Dedicated system	Typically loaded system
No storage index in memory	22 seconds	47 seconds
Indexes for the two larges files (>630,000 records) in memory	< 2 seconds	< 3 seconds
Indexes in memory for all files used by the program	< 1 second	< 2 seconds

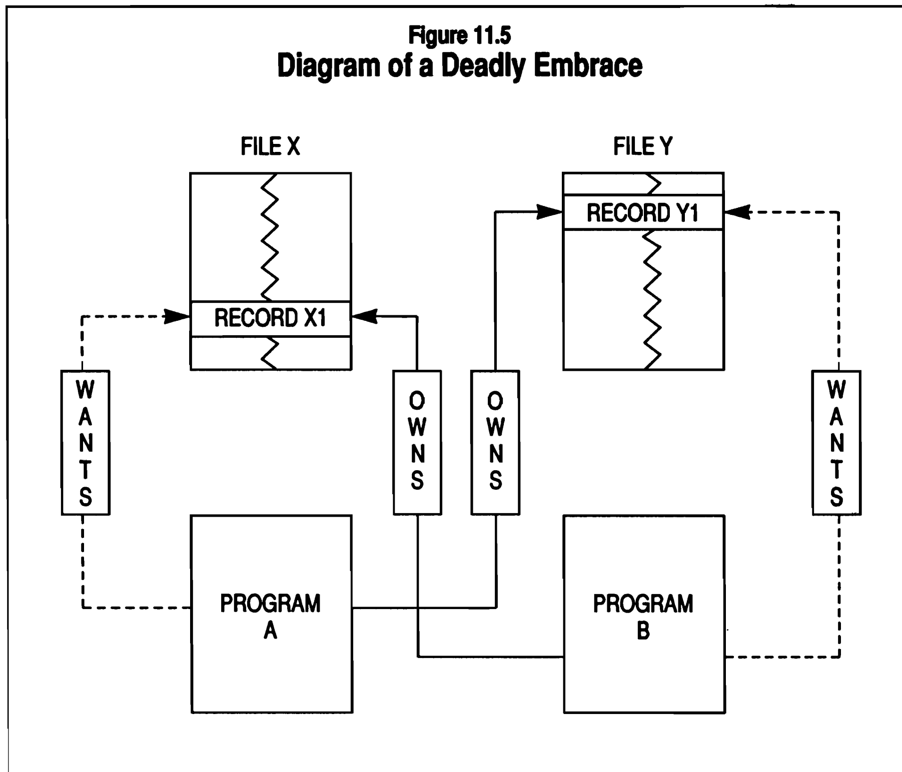
both reading the same two files for update. Figure 11.5 shows that program A has read file Y's record Y1 for update and that program B has read file X's record X1 for update. Each application has that record currently locked for update. After reading those records, program A attempts to read file X's record X1 for update, and shortly thereafter program B attempts to read file Y's record Y1 for update. Program A is waiting for record X1 and program B is waiting on record Y1. In this case, program A and program B can't proceed because they are both waiting on records each other has locked.

A *deadly embrace* has occurred. Program A is waiting on program B to release a record and program B is waiting on program A to release a record. These record releases are contingent upon each other and will never happen naturally. One of the two programs must die to resolve the conflict. The problem is especially nefarious because the deadly embrace triggers no messages or other explicit warnings — program A and program B just sit there waiting on the end of time.

A less critical, though often just as annoying, offense is a *one-way embrace*. Here, program A reads a record for update. That record is locked awaiting operator input and, of course, while that record is locked, no other applications can read it for update. If they try, they wait. Quietly. If the operator at program A goes to lunch with the locked record on her screen, other applications that request that record for update must wait for the operator to get back from lunch. Now, this isn't deadly. A natural, albeit slow, conclusion is possible. But with many operators pounding on the same files, the one-way embrace can be very annoying. Forget lunch: Other applications can be brought to a screeching halt if the operator simply walks away to check on an invoice.

Note also that avoiding the one-way embrace isn't just an operator problem. Your code probably provides a "back-up" command key to back out of an input screen that is updating a locked record. However, unless your code explicitly unlocks the most recently read record for update, the potential

Figure 11.5
Diagram of a Deadly Embrace



exists for a one-way embrace to persist, even if your operator thought she did the right thing by backing out of the field input screen.

The traditional coding sequence that leads to a deadly or one-way embrace looks something like:

```
Display screen to get key values
   Read a record from file A
   Read a record from file B
Display screen for new field values
   If CMD-3 go to previous screen
   Update file A
   Update file B
   Display previous screen
```

Not only is this code exposed to the deadly embrace, but even if the operator cancels the second screen with CMD-3, this code leaves the records read for update locked — exposing the one-way embrace. Those two records will stay

locked until this program reads (and locks) other records or until the operator ends the program.

Technical Note

Batch programs, though not totally immune, are unlikely to suffer from the deadly embrace. It takes a pause, a program waiting on something, to cause a deadly embrace. A pause such as one caused by a divide by zero or other such error could lead to a deadly embrace, but it's unlikely. Generally, avoiding the deadly embrace is a strategy you need only in your interactive programs.

The solution is simple and, as the programmer, it is your responsibility. You must write *all* your interactive programs with the No-Intention-To-Update (NITU) strategy. With NITU, each time a record is read for update, unless immediate updating will follow with no intervening wait on workstation I/O, the record is explicitly and immediately unlocked. Later, after your program has collected the values to write to the record for update, the record is reread and updated.

Figure 11.6 shows a coding algorithm that uses the NITU strategy to avoid the deadly and one-way embrace. Using the NITU strategy, records are always immediately updated or released after being read. Records are always locked for as little time as possible. With the code in Figure 11.6, either record embrace situation is impossible and the user is alerted to changes made by other users.

Figure 11.7 shows the NITU algorithm written in RPG. Note that after the fields for each update file are listed on the I-specs, as many 256-byte fields (RPG's character field limit) as necessary are also defined to define record contents prior to update. The INIT subroutine uses the DEFN opcode to define "holder" fields for these values and subroutines SVRECA and SVRECB are used to save the contents of each record. After each record has been reread, the VLRECA and VLRECB subroutines are used to determine whether changes have occurred since this application last read either record. Your application will determine the strategy required if the record has been modified. For some applications, it might be OK to continue with file update anyway; for others, you'll need to alert the operator and provide a way for the operator to re-enter the field values.

Using the NITU strategy is bothersome because it requires more code and more attention to detail than the sloppy, "embraceable" code does. However, if you take the time to implement the NITU strategy in your applications, they will be more robust and free of potential record conflicts.

Figure 11.6

An Algorithm Using the NITU Strategy

```
Display screen J to get key values
       Read a record from file A
       Save contents of record
       Release record
       Read a record from file B
       Save contents of record
       Release record
Display screen K for new field values
       If CMD-3 go to previous screen
       Reread record from file A
       If record value - saved record value
              update file A
       else
              another user has changed record, alert user
              endif
       Reread record from file B
       If record value - saved record value
              update file B
       else
              another user has changed record, alert user
       endif
       Display previous screen
```

Who is the Culprit?

Agreeing with the need for the NITU strategy is one thing; re-engineering all your applications to use it is another. You probably don't need to sit down this weekend, slogging through all your interactive applications and converting them to the NITU strategy. But each time you modify or fix (if your applications are like ours, a little bug spray is needed now and then) one of those applications, consider adding NITU coding to them. In the meantime, you'll need help spotting the nefarious deadly and one-way embraces we talked about. Utility SHOWUR is just what you need.

SHOWUR determines which records for a file are locked and which job is responsible. By using SHOWUR, you can zero in on the "embracing" culprit and force an end-of-job to the offending application. Without SHOWUR, you're left with few clues as to what's wrong, only that something is.

The utility comprises procedure SHOWUR, RPG program SHOWUR, assembler routine SUBRUR, and screen format member SHOWURFM. (All of the code is included on the "Desktop Guide" diskette.) To use the utility, simply key in SHOWUR followed by the name of the file you're interested in. The resulting screen (Figure 11.8) displays a list of jobs using the file, as well as other related

Figure 11.7

RPG Code Incorporating the NITU Strategy

```
....1....+....2....+....3....+....4....+....5....+....6....+....7....

H
FCRT     CD        200           WORKSTN
FFILEA   UC        512R 7AI     4 DISK
FFILEB   UC       1024R11AI     1 DISK
 *
IFILEA   NS
I                                        1   1 AFLD1
I                                        3   4 AFLD2
I                                        5   7 AFLD3
I                                              .
I                                              .
I                                              .
I                                      510 510 AFLD22
I                                      511 512 AFLD23
 * Define fields to save record value:
I                                        1 256 RECA1
I                                      257 512 RECA2
 *
IFILEB   NS
I                                        1   4 BFLD1
I                                        5   90BFLD2
I                                        9  11 BFLD3
I                                              .
I                                              .
I                                              .
I                                     10231023 BFLD41
I                                     10241024 BDLF42
 * Define fields to save record value:
I                                        1 256 RECB1
I                                      257 512 RECB2
I                                      513 768 RECB3
I                                      7691024 RECB4
 *
C           FIRST     IFNE '1'
C                     EXSR INIT
C                     MOVE '1'       FIRST   1
C                     END
 *
 * Perform screen I/O that brings in key values used
 * to randomly read records from file A and file B:
C                     EXCPT screen format
C                     READ  screen format
 *
C                     EXSR READFA                    Read file A record
C           RNFA      IFEQ '0'                       If record read
C                     EXSR SVRECA                    save record value
C                     EXCPT@RELA                     and release record
C                     ELSE
C                     ...Do processing here for record
C                     ...from file A not found
C                     END
```

Continued

Figure 11.7 Continued

```
....1....+....2....+....3....+....4....+....5....+....6....+....7....
 *
C                    EXSR READFB                     Read file B record
C          RNFB      IFEQ '0'                        If record read
C                    EXSR SVRECB                     save record value
C                    EXCPT@RELB                      and release record
C                    ELSE
C                    ...Do processing here for record
C                    ...from file B not found
C                    END
 *
 * Perform screen I/O to get new values for
 * fields for rec A and rec B here. . .
C                    EXCPT screen format
C                    READ  screen format
C   KC               ...cancel working on this format
 * Command key here canceling current work with the
 * previously read records is OK because the records
 * were released when they were read.
C                    EXSR  UPDFA                     Update file A record
C                    EXSR  UPDFB                     Update file B record
C                      .
C                      . program continues
 *                     .
 *
 *
 *--------------------------------------
 * Read FILEA
 *--------------------------------------
CSR        READFA    BEGSR
C          keyval    CHAINFILEA                 55   Read file A record
C...55               MOVE '1'       RNFA    1        Record not found
C  N55               MOVE '0'       RNFA             Record found
C                    ENDSR
 *--------------------------------------
 * Read FILEB
 *--------------------------------------
CSR        READFB    BEGSR
C          keyval    CHAINFILEB                 55   Read file B record
C   55               MOVE '1'       RNFB    1        Record not found
C  N55               MOVE '0'       RNFB             Record fount
C                    ENDSR
 *--------------------------------------
 * Save contents of record A
 *--------------------------------------
CSR        SVRECA    BEGSR
C                    MOVE RECA1     HRECA1
C                    MOVE RECA2     HRECA2
C                    ENDSR
 *--------------------------------------
 * Save contents of record B
 *--------------------------------------
CSR        SVRECB    BEGSR
C                    MOVE RECB1     HRECB1
C                    MOVE RECB2     HRECB2
C                    MOVE RECB3     HRECB3
```

Continued

Figure 11.7 Continued

```
....1....+....2....+....3....+....4....+....5....+....6....+....7....
C                   MOVE RECB4     HRECB4
C                   ENDSR
 *-------------------------------------
 * Validate contents of record A
 *-------------------------------------
CSR        VLRECA   BEGSR
 * Validate contents of record A
C                   MOVE '0'       RNVA    1        Record not valid
C          RECA1    COMP HRECA1                   55 Equal
C   55     RECA1    COMP HRECA2                   55 Equal
C  N55              MOVE '1'       RNVA
C                   ENDSR
 *-------------------------------------
 * Validate contents of record B
 *-------------------------------------
CSR        VLRECB   BEGSR
 * Validate contents of record B
C                   MOVE '0'       RNVB    1        Record not valid
C          RECA1    COMP HRECB1                   55 Equal
C   55     RECA1    COMP HRECB2                   55 Equal
C   55     RECA1    COMP HRECB3                   55 Equal
C   55     RECA1    COMP HRECB4                   55 Equal
C  N55              MOVE '1'       RNVB
C                   ENDSR
 *-------------------------------------
 * Update FILEA
 *-------------------------------------
CSR        UPDFA    BEGSR
C                   EXSR READFA                     Reread file A
C          RNFA     IFEQ '0'                        Record still there?
C                   EXSR VLRECA                     Record A still valid
C          RNVA     IFEQ '0'                        Record not changed
 * Move screen fields to output fields here
 * then update record A
 *                  EXCPT@UPDA                      Update record
C                   ELSE
C                   EXCPT@RELA                      Release record
C                   EXSR BUFCHG                     Notify user
 * Notify user here that buffer has been changed since
 * last read, then release record
C                   END
C                   ELSE
 * Unlikely, but possible that record was deleted
 * since this program read the record, perform that
 * error processing here
C                   EXSR FILERR
C                   END
C                   ENDSR
 *-------------------------------------
 * Update FILEB
 *-------------------------------------
CSR        UPDFB    BEGSR
C                   EXSR READFB                     Reread file B
C          RNFB     IFEQ '0'                        Record still there?
C                   EXSR VLRECB                     Record B still valid
```

Continued

Figure 11.7 Continued

```
....1....+....2....+....3....+....4....+....5....+....6....+....7....
C           RNVB      IFEQ 'Ø'
C           REC       IFEQ VOLDB                      Record not changed
 * Move screen fields to output fields here
 * then update record B
 *                    EXCPT@UPDB                      Update record
C                     ELSE
C                     EXCPT@RELB                      Release record
C                     EXSR BUFCHG                     Notify user
C                     END
C                     ELSE
 * Unlikely, but possible that record was deleted
 * since this program read the record, perform that
 * error processing here
C                     EXSR FILERR
C                     END
C                     ENDSR
 *-------------------------------------
 * Perform buffer-changed error
 *-------------------------------------
CSR         BUFCHG    BEGSR
 * Notify a user here that a record buffer was changed
 * prior to update. Your application will determine
 * exactly what strategy should be employed here.
 * It might be OK for the program to continue, or it might
 * need to step back to allow the operator to re-enter the
 * field values.
C                     ENDSR
 *-------------------------------------
 * Perform file error
 *-------------------------------------
CSR         FILERR    BEGSR
 * A record has been deleted since being read during  current
 * editing cycle. Abort processing here.
C                     ENDSR
 *-------------------------------------
 * Program initialization
 *-------------------------------------
CSR         INIT      BEGSR
 * Define fields to hold record contents during update
 * Buffer to hold record A contents
C           *LIKE     DEFN RECA1     HRECA1
C           *LIKE     DEFN RECA2     HRECA1
 * Buffer to hold record B contents
C           *LIKE     DEFN RECB1     HRECB1
C           *LIKE     DEFN RECB2     HRECB2
C           *LIKE     DEFN RECB3     HRECB3
C           *LIKE     DEFN RECB4     HRECB4
C                     ENDSR
 *-------------------------------------
ORECA    E               @RELA
 * Note no output fields on release output operation
ORECA    E               @UPDA
O                                    new field values here
O                                    new field values here
 *
```

Continued

Figure 11.7 Continued

```
....1....+....2....+....3....+....4....+....5....+....6....+....7....
ORECB    E              @RELB
 * Note no output fields on release output operation
ORECB    E              @UPDB
O                                 new field values here
O                                 new field values here
```

Figure 11.8

SHOWUR Screen

```
                    Records in use for file CUSTMAST

Job name   User    1st Proc    Cur Proc  Prog     RRN       Owned  Waiting
W2113118   Fred    LIBR#       FLEDIT    FLEDIT   00003241    Y
W3103434   Wilma   CUSBAL      CMAINT    CMAINT   00003241           Y
W4092251   Barney  CUSBAL      CMAINT    CMAINT   00000200    Y
W5103431   Betty   CUSBAL      CUPDAT    CUPDAT   00001565           Y
Y2103856   Betty   CUSBAL      CDELET    CDEL01   00001565    Y









**End**
Roll keys-page             Enter-update            CMD7-End program
```

information. The Roll keys let you page through the entries. If no data is shown on the screen, the specified file is either not on the system or not being used by any tasks. After displaying its initial screen for the file you entered, you may also change the name of the file to look for locks in other files.

Three columns in the display indicate where a record lock may exist. Column RRN shows which of the file's records a job has last read. If the job has not released the record after reading it with intent to update, a "Y" will appear in the OWNED column. If other jobs are trying to use the same record, a "Y" appears in the WAITING column. In such instances, as illustrated by Figure 11.8, a record lock exists. Other applications must wait until the program that caused the lock unlocks the record or until that application ends before they can acquire it.

With the information provided by SHOWUR, you can take the appropriate action to unlock the locked record. If the lock was caused by a one-way embrace at an operatorless workstation, unlocking the record is a simple matter of ending that application appropriately. If the lock was caused by a deadly embrace, a little program triage is required. Only one of the applications with a locked record can survive — the decision rests with you!

A Ray of Light

As we said at the beginning of this chapter, getting the most out of DDM means understanding its limitations and working around its weaknesses. This chapter has provided the ray of light you need to find DDM's weaknesses on your system and the tools and strategies you need to work around them.

We bit off a lot in this chapter. Don't worry about doing everything discussed here at once. For starters, load Index Doctor and run it against a couple of your large indexed files. With Index Doctor's reports and Chapter 3's DDM discussion, you should have a good handle on S/36 DDM and how it works. Then use Index Doctor to diagnose all your large, heavily used indexed files and take the appropriate action steps. You'll be amazed at the results that just a little explicit keysorting or more frequent file reorganization will bring. Then later, consider the other strategies mentioned in this chapter and what their merits might be for your applications. With a tweak here and a nudge there, S/36 DDM can actually work for you, not against you!

Chapter 12

A Baker's Dozen DDM Tips and Techniques

Chapter 11 provided you with four broad strategies for reigning Disk Data Management's (DDM's) choke hold on performance. In this chapter, which closes our section on DDM, we'll cover a baker's dozen tips and techniques to further help you tweak and tune DDM's performance.

Before we dig in, there is perhaps a fourteenth tip — one you've heard many times related to many subjects — we should discuss before we get to the baker's dozen. That tip: Don't fix it if it ain't broke! In every case, these tips and techniques provide ways for you to improve — sometimes offensively and sometimes defensively — DDM performance on your S/36. You probably have applications that don't run as fast as they *should* — but many of those probably run as fast as they *need* to. Before you roll up your sleeves and start applying all these tips, diagnose your applications, and their bottlenecks, and determine the overall effects of those bottlenecks on your system. Know what's broken before you start applying the fix. Use as many of the baker's dozen as you need, but "use 'em only where you need 'em."

1. Consider alternate indexes as an alternative to #GSORT

For those times when you need to process an entire file in a specified sequence in a batch process, consider using an alternate index instead of an ADDROUT or TAGALONG sort. Creating an alternate index with the BLDINDEX procedure or using an ADDROUT sort each take about the same amount of time. The TAGALONG sort comes in dead last. For example, to sort a 17,000-record file with a 320-byte record length and a 22-byte key took BLDINDEX 1:51, ADDROUT 1:56, and TAGALONG a sluggardly 10:24.

Simply creating the access path isn't the whole story, though. You also must consider the time required to actually read the data. For the 17,000-record file, it took 8:53 to read through the entire file via the alternate index, 14:12 via the ADDROUT file, and only 25 seconds via the TAGALONG sort file. (In each case, a 16 K data or index buffer was used.) The total times, then, to sort and process the file were:

	BLDINDEX	ADDROUT	TAGALONG
Create alt index or sort file	1:51	1:56	10:24
Read data file	8:24	14:12	0:25
Total time	10:15	16:08	10:49

Performance Tip

For batch processing a file in a specified sequence, use a TAGALONG sort if you have abundant disk space. It offers the best combination of flexibility and speed. If you often process *entire* files in a specified sequence, consider using an alternate index instead of #GSORT. Here you'll get the speed of the TAGALONG (or almost) without its voracious appetite for disk space.

The total performance using an alternate index edged out the TAGALONG as the fastest way to read a file in a specified order. Each method, though, has limitations that should not be overlooked. The ADDROUT, as you've seen, is slow. The alternate index method works only for those times when you need to process the entire file; you cannot create a conditional alternate index (e.g., alternate indexes always include all the records in the file). For large files, the TAGALONG method requires lots of free disk space. If you often process the entire file in your batch file processing, use the alternate index method; if you have lots of disk space and often need to conditionally include or omit records, use the TAGALONG method — if you have the disk space it always offers speed and flexibility. For disk-bound programmers who need to process just part of a file, the slow ADDROUT method is sometimes the only alternative.

2. Consider replacing indexed files with sequential files and alternate indexes

Consider sequential files with alternate indexes instead of traditional indexed files for your large files that require random access. Despite slight additional overhead, used prudently the alternate index is the best way to handle indexed files on the S/36. With alternate indexes, you gain key update ability. You can also put the parent file on a different drive than the alternate and thereby improve performance.

If the file must be reorganized, the "file" on which the COPYDATA procedure or $COPY program should run is the alternate index whose order, by the definition of its key, most closely approximates the order in which the file is most frequently accessed. The rule holds true regardless of whether or not multiple keys are defined for the file. A REORG would be specified, but the output file would be sequential (S). The sequential file would be created in key sequence order as determined by the alternate used to initiate the REORG. The closer the relationship between a file's most frequently used key order and that file's physical record sequence, the faster that file can be processed.

After the file has been reorganized, the "disorganized" file must be deleted. But before deleting the disorganized copy, you must delete its alternate indexes. After all alternates and the disorganized parent have been deleted, the newly organized parent may be renamed to the name of the original and its alternate index (or indexes) may be rebuilt.

After the file reorganization, processing the parent sequentially by "key" is *very* fast. There is no index being used to read the records, so you don't have to worry about providing buffer space (via IBLOCK) to index blocking; simply use as large a DBLOCK value as possible. Remember, though, to be prudent in the number of "permanent" alternates you maintain over the parent sequential file. Too many, and the technique will probably impede, not enhance, performance.

3. Keep alternate indexes to a minimum

At first, it might seem like this tip is in direct opposition to the first two. It's not. Use alternate indexes where you need to — they are very handy and easy to use — but only where you need to. Remember, keeping many *unnecessary* alternate indexes on a file can slow interactive performance. When a batch program adds records to a parent file while interactive programs are using an alternate index (or indexes) built on that parent file, interactive response time diminishes considerably. Remember, when records are added to a shared file, DDM must keep the overflow areas in key sequence. The DDM shuffling of the overflow of the opened alternate indexes will slow performance.

When records are added to a parent file and existing alternate indexes are not opened by any program, index entries are written in arrival sequence to the closed alternate index's overflow area — with the assumption that a subsequent keysort will reorder the overflow area. This is referred to as "delayed maintenance." Relying on delayed maintenance makes record addition to the closed alternate indexes fast, at least when compared to adding the records to opened alternate indexes; but the next application needing these stale alternates pays the price. That application will bide its time during initiation while DDM furtively keysorts the previously unopened alternate index's stale overflow area.

If possible, design your applications to build alternate indexes as needed and then delete them as soon as possible. For heavy-duty batch adds, you might even find that deleting and rebuilding an alternate index is faster than enduring the DDM-called keysort required at job initiation to freshen a stale overflow area.

As we've referred to elsewhere in this book, don't forget to minimize, or eliminate entirely, duplicate keys. Adding many records with duplicate keys will bring performance to its knees. See the tips in Chapter 3 for avoiding duplicate keys. Remember also that when a key value is changed in an alternate index, the effect is the same as adding a new duplicate key. Figure 12.1 shows a key value that has been changed to 46953. This key value is associated with relative record number 51. DDM must insert this new key in the overflow as though a new key were being added. The ripple-down add required to add this "changed" key will impede performance.

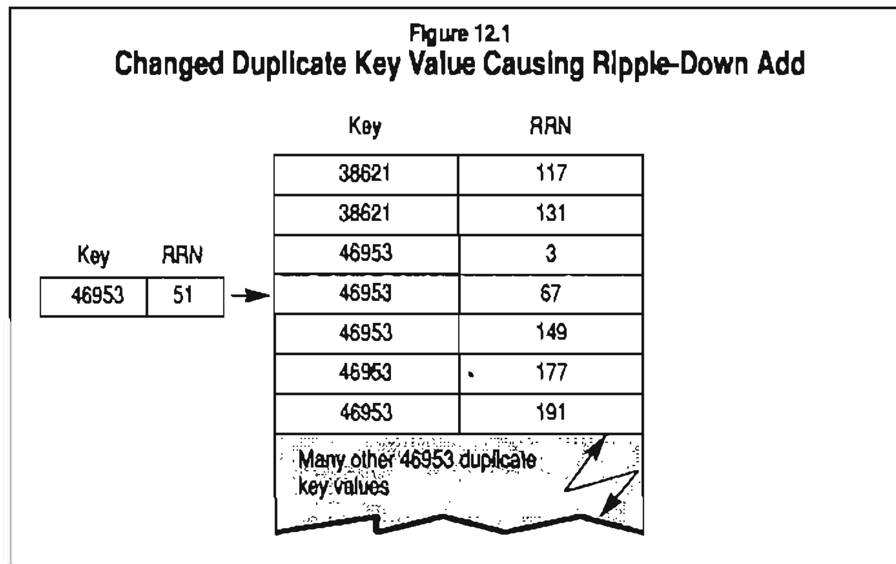
Performance Tip

Changing a duplicate key value in an alternate index has the same detrimental effect as adding duplicate keys. In either case, a ripple-down add will most likely be needed to add the records, dramatically affecting performance. The moral: Avoid duplicate keys in an alternate index.

4. Spindle placement is more important than file placement

Many S/36 performance tuners spend lots of time juggling the placement of files on a disk to minimize the disk seeks greater than 1/3, as reported by SMF. It's generally accepted that if 20 percent or 30 percent of your disk seeks are greater than 1/3 of the maximum seek distance, performance suffers. It is true that the less head movement, the better; but close file placement of related files is greatly overrated. Most of the disk seek time is spent starting and stopping the disk arm — the actual distance the disk arm moves is not really so important.

Figure 12.1
Changed Duplicate Key Value Causing Ripple-Down Add



Consider the following timings for a 200 MB 9332 drive:

Acceleration time	4.00 ms
Seek time	.01 ms/cyl
Settling time	4.00 ms
Rotational delay	10.00 ms

which result in the following times:

Seek over entire disk (1349 cylinders) takes 31.5 ms
$4 + (1349 * .01) + 4 + 10 = 31.5$ ms

Seek over 2/3 of disk (900 cylinders) takes 27.0 ms
$4 + (900 * .01) + 4 + 10 = 27.0$ ms

Seek over 1/3 of disk (450 cylinders) takes 22.5 ms
$4 + (450 * .01) + 4 + 10 = 22.5$ ms

Seek of one track takes 18.1 ms
$4 + .1 + 4 + 10 = 18.1$ ms

A seek over the entire disk takes 31.5 ms, yet a seek of just one track still takes 18.1 ms! Performance is measured in milliseconds and 18.1 ms is indeed better than 31.5 ms, but that is the most extreme example. In production environments, the disk seek variations are more likely to hover in the middle of the timings, where there just isn't much performance difference. So, yes, close placement of related files saves time, but it doesn't save a lot of time.

What is far more important than file location on a single drive is file placement on the appropriate drive. Your goal is to distribute file placement evenly across all drives. If your system has more than one spindle, check to make sure usage is balanced across all available spindles. No one drive usage should deviate more than 10 percent from the other drives. For a drive to vary more than that means it's doing more than its share of the work — and slowing your performance. One or more inactive spindles is far more damaging to performance than "incorrect" file placement across a single drive. Try to split all files across all drives. For example, when you use alternate indexes, try to group the alternates on one spindle and the parent on another. See Chapter 13 for specifics on using SMF to measure file placement and disk utilization.

Performance Tip

If your S/36 has more than one disk spindle, create alternate indexes on a spindle other than the one where the parent file is located. During access via alternate indexes, this will help eliminate the movement required by the disk heads to find index entries and records — thus improving performance.

Technical Note

Myth debunked: Close placement of related files on one spindle is greatly overrated! It is far more important to spread file usage evenly across all spindles than it is to worry about individual file location on a given spindle. Worry less about disk seeks greater than 1/3 and more about balancing your disk utilization. No one drive's utilization should deviate more than 10 percent from the other drives' utilization.

5. Share if you must

Another popular myth is that shared file disposition always negatively affects random record retrieval. The theory goes that using the DISP-SHRxx keyword automatically forces a reread of *every* blocked record on *every* READ or CHAIN operation — so don't DBLOCK a shared file. The truth is that the S/36 only rereads a block if another task updates a record *currently contained* in a data buffer. DDM is smart enough to know dynamically when a buffer is out of date and then, and only then, is a buffer refresh required — not simply because the file is using the DISP-SHR keyword in its // FILE statement. For those batch update-add programs that might occasionally share a file with an on-line inquiry program, go ahead and share the files. The convenience of having the inquiry program available probably overweighs the few times when its use might cause DDM to refresh the batch program's data buffer.

For a different reason, though, you might want to avoid sharing all files all the time. We've previously discussed the performance impact of adding random records to an indexed file. As you well know by now, when the file is shared, DDM must keep the overflow area in key sequence. When records largely out of key sequence are added to a shared indexed file, DDM huffs and puffs to keep the overflow ordered. When you add a large number of records to an *unshared* file, DDM simply squirts the new keys into the

index overflow in arrival sequence — not in key sequence. In this case, the record addition will be much faster, but the next person to use the index will wait while DDM uses delayed maintenance to keysort the overflow in key sequence. Consider adding random records to an unshared file, and then immediately forcing a preemptive real keysort on the updated file (see Chapter 3 for details about forcing a real keysort on a file).

6. Do some of DDM's work yourself

Another way to get a leg up on DDM is for you to do a little of its work. We've just discussed that by adding records to an unshared file, you defer, but don't avoid, some of the DDM huffing and puffing required to maintain the index overflow area. A way to avoid the huffing and puffing and to avoid delayed overflow maintenance is to presort the input file in key sequence before adding it to the indexed file. When the input file is in key sequence, keys are added to the overflow in key sequence; ripple-down adds will not occur and delayed maintenance is not required to maintain the overflow area. Presorting the input file will be as fast as if random records were being added to an unshared file without the penalty of delayed overflow maintenance.

Test results show that it takes 2:33 to add 1,000 random records to a shared indexed file without presorting them. A BLDINDEX to create an alternate index took 13 seconds (in effect, sorting the input file in key sequence) and the RPG program to add the records took 19 seconds — for a total of 32 seconds. Almost five times as fast! And that's for a relatively small number of records. Notice that is a perfect time to use an alternate index over #GSORT — no conditional inclusions or exclusions are required for the input file; just don't forget to delete the alternate index after the records have been added. The moral: When you can, presort input files in key sequence.

7. Take the bypass

When DDM attempts to add a record to a file, it scans the entire index (as well as the overflow area) to see if the key to be added already exists in the file. As more records are added to the file, this duplicate-key checking takes longer and longer.

If duplicate keys are not a concern, perhaps because you know that no duplicate keys will ever exist in the file (e.g., your program checks first before adding the new record), you can speed the addition of many records to an indexed file with the // FILE statement's BYPASS-YES parameter. Specifying YES for this parameter tells DDM not to check either index area (the primary or overflow) for a possible duplicate key, and causes the record to be quickly and directly added to the file. Note that for indexed files that are duplicate-key-capable, DDM always bypasses duplicate-key checking, even if BYPASS-NO is specified.

The larger a file becomes the longer duplicate-key checking takes. Therefore, record-adding performance is often contingent on the size of the file. Using BYPASS-YES for jobs that add lots of records to large indexed files will make them perform more consistently. At the end of each month, it won't take very much longer to add records to the file than it did at the beginning of the month. Consider using BYPASS-YES when the output file is DISP-OLD or when your application checks for duplicate keys before adding them. Performance will increase substantially when BYPASS-YES is used when many records are added to a large indexed file.

8. Avoid being underextended

As a way to avoid full files, the // FILE parameter EXTEND-x is a often a lifesaver. However, using too small an extend value (either with EXTEND-x or as the default value when the file is built) often can wreak pure havoc with performance.

For sequential, direct, and alternate index files, if enough disk space is available immediately after the file to accommodate the extend, that disk space will be used to extend the file. If enough disk space is not available, or if the file is an indexed file, when a file is extended it is copied to another place on disk that has been allocated storage based on the original file size plus the extend value. For all practical purposes, plan defensively for all file extends to require the entire file to be copied to another place on disk. The problem occurs when too small an extend value is specified, causing several extends to occur in one session.

Disk extends are annoying because they can take a long time to perform for a large file. But more importantly, without enough disk space and/or with EXTEND values too small, file extend operations can fail — and that is really annoying. Figure 12.2 illustrates the sequence of events that occurs when a file is extended. You want to avoid having the first extend fill most of your available contiguous disk space, leaving too little contiguous disk space available for subsequent file extends. Consider a system with 16,000 free contiguous blocks. When a file originally allocated at 8,000 blocks, with an extend value of 100 blocks, becomes full, its extended size requires 8,100 of the 16,000 current free contiguous blocks. If that same file exceeds its allocation a second time, there is not enough free contiguous disk space on your system for it to extend again. Your application will experience an untimely demise.

There are a couple of lessons here. First, strive to always have plenty of free disk space available. To the point, as we advocated in Chapter 3, of perhaps buying more. And compress disk space often. Lots of fragmented disk space is of little value. As much free contiguous disk space as possible minimizes the chances of a file-extend operation failing. Second, review all current extend values (either those specified when the file was created or

Figure 12.2
Sequence of Events for an Extended File

```
If the file is non-indexed or an alternate index then
      If additional space is available immediately after the file then
            Extend the file by moving the end-of-file pointer
            (extend in place)
      Else
            If a large enough contiguous area of disk space
            is available for the extended file then
                  Copy the file with its new size to that area
                  of disk
            Else
                  Issue file full message
            Endif
      Endif
Else
      If a large enough contiguous area of disk space
      is available for the extended file then
            Copy the file with its new size to that area of
            disk
            Extend all associated alternates
      Else
            Issue file full messsage
      Endif
Endif
```

those specified with the EXTEND-x parameter). Extend values should always extend the file at least by as many records or blocks as can be added to the file at one time. Note also that OCL EXTEND-x values override the default extend value used when the file was created.

9. Change default file extend value easily

The S/36 doesn't provide any direct way to change a file's default extend value without copying the file — which, for large files, requires lots of free disk space and takes a long time. Furthermore, there isn't an EXTEND parameter provided for the COPYDATA statement, and the manuals don't tell you how to change a file's default extend value using IBM-supplied programs. By making the default extend value hard to change, the S/36 locks you into managing extend values with the // FILE statement's EXTEND-x parameter.

To make it possible to quickly and easily change a file's default extend value, without copying the file, we have provided FILEXTND, a simple command-line utility to retrieve or directly change a file's default extend value. Remember that using the EXTEND-x keyword in OCL overrides the file's default extend value, so you can always temporarily override a file's default extend value. Use FILEXTND when you want to change the global default file extend value.

To retrieve a file's extend value, use:

```
FILEXTND file name,GET
```

To change a file's extend value, use:

```
FILEXTND file name,PUT,new extend value
```

If the specified file does not exist, no operation is performed and no message is displayed. When FILEXTND gets a file's extend value, it puts that value in positions 1-8 of the user area of the Local Data Area. As a bonus, when getting the file's extend value, FILEXTND also puts the file allocation type (R=records, B=blocks) in position 9, the number of records or blocks allocated in positions 10-17, and the number of records used in positions 18-25.

To change a file's extend value, use a 1- to 8-digit block or record value that specifies the amount of additional space to use for the extension. To remove a file's default extend value, specify 0 (zero) as the new extend value. With FILEXTND, your file extend strategy can now be managed in one place, rather than sprinkled throughout many inconsistent EXTEND-x parameters in several different procedures.

Technical Note

Although not documented anywhere, you can also change a file's default extend value by adding the // FILE statement's EXTEND parameter to $COPY's COPYO file statement:

```
// FILE NAME-COPYO,LABEL-label name,UNIT-F1,EXTEND-x
```

But remember, this requires disk space and time. It's much easier to change file extend values with FILEXTND.

10. Don't pack 'em in

The S/36 has native hardware instructions to perform zoned-decimal arithmetic. However, to perform arithmetic on RPG or COBOL packed or binary fields, additional processor-intensive routines are required to convert those fields to zoned-decimal format. Usually, packed and binary representation is used to save disk space; but as we've said before, you should have lots of disk space. The S/36 can deal fastest with zoned-decimal numeric values — there is no conversion necessary — with packed storage being second fastest and binary the slowest.

Consider the following performance timings when a program reads a 10,000-record sequential file with 12 9-digit numbers in each record and then provides a total of the 12 numbers for each record:

Zoned decimal	4:05
Packed	7:19
Binary	20:36

Packed storage almost doubles the time taken to process the file and binary storage goes off the chart! For pure performance purposes, if you have the disk space, design all your new applications with zoned-decimal storage. Zoned storage is faster and it's easier to use for conditional sorting. Consider using packed or binary storage when records are infrequently read and disk space is at a premium.

Technical Note

The AS/400 works fastest with packed data storage. Its performance is impeded by zoned-decimal storage. Conversion routines are required to transfer to memory data stored in zoned-decimal and binary. If you are designing new applications on the S/36 that you think are very likely to be migrated to the AS/400, consider the performance on each machine before you make a final data storage decision for your new applications.

11. Reorganize files often — the easy way

We have mentioned many reasons in this book why you should reorganize your indexed files frequently. However, with its orphan alternate indexes (quick now, how many alternate indexes do you have on your system, who are their parents, and what are their key descriptions?) and the long list of parameters that COPYDATA requires, the S/36 is certainly not reorganize-friendly.

To automate the process of reorganizing indexed files and rebuilding all dependent alternate indexes, we have included REORGX on diskette, an indexed file reorganization utility. REORGX will do everything that COPYDATA does (reorganize a parent file in key sequence, resize a file, optionally remove deleted records, and optionally include or exclude specific records). But REORGX does considerably more than COPYDATA does (and it also does more than RGZFILE in the IBM S/36 VASP package). REORGX's added features are:

- REORGX automatically deletes all dependent alternate indexes before reorganizing the parent file and then automatically rebuilds them after the reorganization. When REORGX initiates, it checks the VTOC to see what dependent alternate indexes currently exist over the parent — thus always deleting and rebuilding the alternate indexes that were on disk just before reorganizing the parent.

- In addition to letting you specify a preferred location for the parent file, REORGX also lets you specify the preferred location of the alternate indexes. If preferred locations are not specified, REORGX will attempt to locate alternate indexes on a spindle other than the one containing the parent file.

- REORGX provides a way to "grow" or "shrink" a parent as a percent of its current allocation.
- REORGX's last extra feature is that in addition to including records by specific positional character value (just as COPYDATA does), REGORX allows from/to record selection by relative record number or key value.

Using REORGX is very much like using COPYDATA — just a little more convenient. REORGX's two prompt screens are shown in Figures 12.3a and 12.3b. For most uses of REORGX, you'll rarely use the second prompt screen. Figure 12.4 shows REORGX's syntax when calling it from a procedure. With REORGX, it's easy to maintain your system's indexed files. Use REORGX frequently on your indexed files to clean up index overflow areas, to remove deleted records, and to resequence parent files in physical key sequence.

12. Provide lots of disk space for KEYSORT

Chapter 3 discussed how important it is for your S/36 to have plenty of disk space. One of the major reasons you need lots of disk space is to minimize the chances of KEYSORT performing an in-memory KEYSORT. Recall that there are actually two kinds of KEYSORT:

- A work-file KEYSORT used when enough disk space *is* available
- An in-memory keysort, used when enough contiguous disk workspace is *not* available for the KEYSORT's work files

An in-memory keysort will take much longer than a work-file keysort. In fact, literally, an in-memory keysort on a large file can take days! (See Chapter 3 for a formula to use to ensure that your system has enough free disk space to perform a work-file keysort on your largest file.) As if in-memory keysorts weren't bothersome enough, they are also sneaky. You are not alerted to the fact that one is taking place. When an in-memory keysort occurs, your S/36 just sort of saunters its way through the in-memory keysort, bringing performance to its knees along the way. Oh, you'll know something is wrong — but you won't know what.

The only way out of an in-memory keysort is to IPL your machine — in the middle of the offending keysort — bypassing file rebuild, clearing off enough disk space, and IPLing a second time — this time with file rebuild. The remedy is unnerving though, because you're never 100 percent sure what it is you're trying to cure. The S/36 provides no explicit way to determine if an in-memory keysort is taking place.

To solve the problem of how to detect an in-memory keysort, we've provided the SLOWKS command-line utility. SLOWKS takes just a second to run and returns a message indicating whether or not an in-memory keysort is occurring. When you suspect an in-memory keysort is taking place, first end

Figure 12.3a

REORGX Prompt Screen #1

```
                         REORGX Procedure                        Optional-*
           Reorganize an indexed file and rebuild related alternates
Name of file to be reorganized. . . . . . . . . . . . . . . . .

Creation date of file to be reorganized . . . . . . . . . . . .              *

Name of new file to contain copied records  . . . . . . . . . .

Reallocate unit or increase/decrease from current allocation. .              *
                                    BLOCKS, RECORDS, INCR, DECR

Size of file to be created or increase/decrease percent value .              *
                   BLOCKS/RECORDS: 1-8000000  INCR/DECR: 1 to 100

Preferred parent disk location .  . .  A1,A2,A3,A4,block number              *
Preferred alternates disk location  .  A1,A2,A3,A4,block number              *

Type of file retention  . . . . . . . . . . . . . . . . . . . T,J  T

Reorganize sequentially by key and/or omit deleted records  . .
                                                  NOREORG,REORG  NOREORG

CMD 3-Previous Menu  CMD 4-Put on job queue  CMD 14-More options
```

Figure 12.3b

REORGX Prompt Screen #2

```
                         REORGX Procedure                        Optional-*


Include or omit selected records  . . . . . . . .  INCLUDE,OMIT              *

Starting position of comparison characters  . . . . . .  1-4096              *

Condition for record selection  . . . . . . . EQ,NE,GE,GT,LE,LT              *

Comparison characters . . . . . . . . . . . . . .  'characters'

Select record, key or packed key. . . . . . . RECORD, KEY, PKY               *
     From value . . . .____________________________________________
     To value . . . . .____________________________________________

Record length of new file . . . . . . . . . . . . . . .  1-4096              *

Maximum number of records to copy . . . . . . . . . . . . . . .              *


CMD 2-Page back  CMD 4-Put on job queue
```

Figure 12.4

REORGX's Syntax When Calling from a Procedure

```
REORGX old file name,[mmddyy] new file name, {,               } .
                     |ddmmyy|                 |BLOCKS,blocks  |
                     [yymmdd]                 |RECORDS,records|
                                              |INCR,incr value|
                                              {DECR,decr value}

[A1 (parent)] ,[A1 (elts)] ,[T],[NOREORG],[INCLUDE],[position,][EQ] .
|A2         |  |A2       |  [J] [REORG  ] [OMIT   ]            |NE|
|A3         |  |A3       |                                     |LT|
|A4         |  |A4       |                                     |LE|
[block no   ]  [block no ]                                     |GT|
                                                               [GE]

['characters'] ,[record length] ,[maximum records]

[RECORD]  ,  [from value]  ,  [to value]
|KEY   |
[PKY   ]
```

normally all the active jobs you can (you won't be able to end the offending job that caused the in-memory keysort). After narrowing the list of possible offending jobs, type in:

```
SLOWKS jobname
```

for each remaining job. If one of the remaining jobs is being hung by an in-memory keysort, SLOWKS will tell you. After confirming that a slow keysort is running, perform the steps mentioned above to bail out of the slow keysort. By telling you exactly what is happening, SLOWKS makes that first unnerving IPL a little easier to perform.

13. Put memory to work for system programs

As programs and their associated data buffers need memory, the S/36 doles it out in 2 K chunks called pages. The combined pages of memory used by a program and its data buffers are referred to as the program's region. The default region size is 24 K and can be as large as 64 K. You can use the REGION OCL statement to change the default region size used for a job, or you can use the SET procedure to change the default region for an entire workstation session.

Increasing region size does *nothing* to improve the performance of your user programs. You can only provide more memory to them to improve performance through record and index blocking (see Chapter 11). However,

Performance Tip

Specifying a larger region size (with // REGION or with the SET procedure) provides more memory to #GSORT and $COPY to buffer disk I/O. Think of using a larger region size as a way to increase the record blocking factors for these system programs. The region size does not affect your application programs at all; they can only be given more memory for disk I/O buffering with explicit (via DBLOCK and IBLOCK or the program's F-spec) record or index buffering (see Chapter 11).

#GSORT and $COPY (the system programs used by COPYDATA, LISTDATA and LISTFILE, SAVE and RESTORE, and SAVENRD) can be speeded up by providing a larger region size to them. When a larger region size is available, these programs use it to buffer disk I/O.

Beware: Just as with record blocking for your application programs, larger region sizes can result in performance-inhibiting disk swapping. Remember, real memory is a finite resource and when a system program takes advantage of a larger region size, it might be doing so at the expense of an application program that also wants that memory. Generally, you should avoid using the SET procedure to change the default region size to 64 K for an entire session. It is much better to specify a larger region using the // REGION statement in your procedures that use #GSORT or $COPY.

Although a larger region size can benefit #GSORT and $COPY, the same cannot be said for all system programs. S/36 users have long debated whether or not a larger region size speeds up the COMPRESS procedure. It does not. For a system program to use memory made available by a larger region, the program must first make an explicit GET PAGE supervisor call to enable that program to use the larger region's additional memory. COMPRESS does not perform the GET PAGE supervisor call and therefore does not take advantage of the extra memory. Few other system programs are as disk intensive as #GSORT and $COPY and are far less likely, even if they perform the supervisor call, to have performance improved by a larger region size. If a system program can't take advantage of a larger region, the // REGION statement or the region as specified by the SET procedure is ignored.

Technical Note

Myth debunked! Record blocking (via DBLOCK) doesn't do anything to improve #GSORT performance. Record blocking only works with file I/O channeled through DDM and #GSORT bypasses DDM — using its own internal I/O routines. DBLOCK on a file statement in #GSORT OCL won't hurt, but it won't help either. To buffer #GSORT as much as possible, specify a 64 K region in the OCL that calls the #GSORT.

Recapping the Baker's Dozen

There you have it: 13 of our favorite tips and techniques for improving S/36 DDM performance and avoiding its pitfalls. Here's a quick recap:

1. Use alternate indexes as an alternate to #GSORT.
2. Consider replacing indexed files with sequential files and alternate indexes.

3. Keep alternate indexes to a minimum.
4. Spindle placement is more important than file placement.
5. Share files prudently.
6. Presort input files in key sequence before adding them to indexed files.
7. Bypass duplicate key sharing when you can.
8. Use large enough file EXTEND values.
9. Manage default file EXTEND values with FILEXTND.
10. Avoid packed and binary data storage.
11. User REORGX to reorganize your indexed files frequently.
12. Always have enough free disk space to avoid in-memory keysorts.
13. Use a 64 K region to improve #GSORT and $COPY performance.

We have spent a lot of time in this book harping on DDM performance — and for good reason. Early on we mentioned that in the time it takes for *one* disk I/O, as many as *35,000* machine instructions can be executed. Think about that. Where else on the S/36 can you performance tune with a 35,000-to-one improvement ratio? OK, we're being a bit melodramatic, but the reality is that your programs don't expend much *code* specifying disk I/O, and they expend most of their *time* performing it. As you prepare to tune your S/36 for more years of faithful service, carefully consider all areas of disk I/O — that's where most of your problems lurk and performance lies untapped.

Section V

Performance Measurement and Tuning

"You can observe a lot just by watching."

—Yogi Bera

They say a watched pot never boils. For S/36 managers, a different truism applies: An unwatched machine may boil, but it will never really get cooking. The key to any performance improvement plan is careful measurement and analysis of a few critical system variables.

Unfortunately, the extensive array of performance monitoring figures spewed out of SSP gives you the sinking feeling that there's just too much to watch! The average System Measurement Facility printout for just a few hours of operation can be hundreds of pages long. What S/36 analysts need is a way to cut through the noise and get their hands on those few critical variables.

The chapters in this section provide plenty of ways, in the form of tips, techniques, and tools. Chapter 13 reveals the secrets to getting and analyzing useful measurements with SMF, and includes a "cookbook" of recipes for solving particular performance problems using SMF. Chapter 14 shows you how to determine if you have enough memory (hint: you don't), and how to use more when you get it. Chapter 15 explains disk caching and provides pointers on how to squeeze the most speed out of this SSP feature. Chapter 16 lets you put your finger on the one measurement all your users instinctively feel: response time. After putting the information from this section into practice, you'll be "cook'n with gas."

Chapter 13

Using SMF

Previous chapters point out how adequate disk and memory capacity are essential for good S/36 performance. If you merely increased memory and disk capacity, and then hoped for the best, you'd undoubtedly see improved response times. You would not, however, be able to say just how much improvement took place or whether or not your system used the additional disk and memory as efficiently as possible. Getting those answers requires careful measurements before and after the upgrade. Making those measurements is the job of IBM's System Measurement Facility (SMF) utility.

Unfortunately, using SMF can be a daunting task. SMF produces such massive quantities of statistical output that many S/36 managers have come to think of it as System Measurement Fiasco. Any programmer worth his or her salt knows how to start and stop SMF, and how to print its reports. What is missing from IBM's documentation is a *strategy* for effectively running SMF and evaluating its output. This chapter provides such a strategy. You'll learn the secret to capturing useful SMF data, which key values to evaluate first, and how to quickly identify and solve performance problems. You'll also have a useful "cookbook" of procedures for using SMF in specific performance tuning situations, along with useful suggestions on advanced applications of SMF. While the chapter is not intended to give you the be-all, end-all explanation of SMF's nooks and crannies (see "References" at the end of the chapter for a list of articles that provide such explanations), you'll get a jump-start on tuning techniques with enough detail to handle most performance problems.

The Good Catch

Before you can evaluate SMF measurements, you must collect good data — a job that sounds simple but is full of traps for the unwary. Merely running SMF all day long and then printing out a 500-page report won't get you any closer to understanding your system's performance than will a divining rod or goat entrails. Here are the keys to capturing an accurate batch of statistics and producing useful reports:

Measure only during periods of uniform activity. SMF tracks averages, which lose their validity if the workload changes dramatically over the course of a single SMF run. Don't turn on SMF in the morning and let it run all day (right through morning break, lunch, afternoon tea, evening batch runs and

backup with David Letterman). Long periods of inactivity (or fierce activity) distort SMF's averages and hide peak values that could reveal serious problems. A better approach is to segment your day into different kinds of activity during which system usage is constant, then choose one of those segments and monitor it for several days. Watch out for such hidden wrenches as embedded backups, diskette usage, and batch reports. If necessary, turn off SMF during these interruptions and turn it on again afterward.

Use one-minute intervals. On the SMFSTART procedure, use the default snapshot interval of one minute, which provides samples frequently enough to be useful without incurring undue overhead. One-minute intervals also let you more easily relate snapshot values to per-minute usage statistics.

Collect I/O and SEC data by task. The default for this SMFSTART parameter is 'N', so you must override it to 'Y'. I/O and System Event Counter statistics by task let you identify programs that might be "hogs" — using so much of one or more resources that other programs can't get much time to run. Although the extra data consumes a significant amount of disk space in the SMF log file, you only use these files temporarily, so the cost in space shouldn't be a problem. With one-minute intervals, the extra CPU overhead for this option is negligible.

Don't collect data by file. User and system statistics for each file are only useful when you are moving files around to balance disk usage, so don't waste time and space collecting them (unless, of course, you're trying to balance your disks).

Enter the correct line speeds for communications. SMF can't determine the line speed at which your communications lines are operating, because this line speed usually is determined by the modem or other external equipment. Entering the correct operating speed on SMFSTART is the only way you'll get accurate line usage figures; so take the time to find out precisely the speed at which your modems, DSUs, or other external equipment actually operate.

Use a meaningful logfile name. A meaningful, and consistent, naming convention lets you quickly identify the correct file for a report once you begin accumulating many SMF logs on disk. One strategy is to identify the time period of the snapshot in the name. SSP already keeps track of the date a file is created — you can get the information from a CATALOG — so don't use up valuable name space with a six-digit date. Starting time is a more useful bit of information to capture in the file name. You might, for example, use a name of the form Smddhhmm, where *m* is a one-letter month identifier (J, F, M, etc.), *dd* is the day of the month, and *hhmm* is the start time of the snapshot.

Allocate plenty of file space. Make sure you specify a large enough logfile size on SMFSTART to accommodate all the data you plan to collect. If the file fills up, SMF stops itself, resulting in lost SMF collection data. You can estimate the amount of data using the following space requirements per snapshot:

- 1 sector per snapshot for system values
- 1 sector per three tasks
- 1 sector per two tasks when collecting by task
- 1 sector for every four comm lines when collecting communications usage
- 1 sector for every four files when collecting by file

For example, a two-hour run at one-minute intervals would require 120 snapshots. If you average 12 tasks and 40 files per snapshot, and collect both task and file statistics, you'll need 21 (1 + 4 + 6 + 10) sectors per snapshot, times 120, for a total of 2,520 sectors. Dividing that by 10 yields 252 disk blocks. In this example, you should specify 300 to be safe. You can "shrinkwrap" the file after the run by using COPYDATA, specifying a size just large enough to hold the actual records.

Don't print whole reports. Usually you'll only need a handful of values out of the thousands collected by SMF in a single run. Look at a two-page summary report first to determine if you need detail information. Then save paper and your local forest by using spool-viewing tools to examine the voluminous detail reports. Utilities such as IBM's COPYPRT CRT, KSI's QueueView, or the VASP (Value Added Software Package) WRKSPF (Work With Spool File) let you browse SMF reports online. You can get printkeys or small excerpts of a report to capture hardcopy records. And don't forget that SMFPRINT lets you specify a time range for detail reports; use that feature if you can isolate a problem to a small range of snapshots.

Following these guidelines will help you obtain accurate, useful measurements. You'll need to make several SMF runs on different days to be sure you've captured representative statistics and not just a fluke. The cardinal rule for evaluating SMF output is *never make tuning decisions based on one SMF run.*

The Quick Look

Once you've captured a good series of SMF log files, you're ready to begin analysis. You often can isolate performance problems with a "quick look" by evaluating a few key measurements. Start with a summary report (See "SMF Summary Report Part 1: Summary Usage," page 226, and "SMF Summary Report Part 2: Summary System Event Counters," page 227) and look for high values in these key areas. Ignore the peak measurements for now; it's the average values that reveal performance problems. A high peak might represent a few seconds of heavy system load; a high average represents continuous heavy loads.

Figure 13.1 lists key SMF measurements and the threshold values to watch for. Your goal in the quick look approach is to identify a bottleneck in one or more of the primary system resources (disk, memory, CPU). The figure

Figure 13.1
Key SMF Measurements

SMF Value	Normal Range	Take Action
MSP usage	up to 60%	>80%
CSP usage	up to 65%	>85%
Disk usage	up to 60%	>85%
Disk seeks > 1/3	up to 20%	>30%
User Area Disk Accesses	100-200/min	>400/min for 5360/62
		>300/min for 5363/64
Translated calls/loads	>5:1 ratio	<2:1 ratio
Comm line usage	90% batch	>95%
	50% interactive	>70%

shows normal ranges for seven key measurements, and "take action" thresholds that signal performance problems. If your SMF reports show any of these measures higher than normal but lower than the action threshold, you may still want to take steps to bring the values within normal range. Here are some tips for evaluating the key measurements listed in Figure 13.1:

MSP/CSP Usage Values ("SMF Summary Report Part 1: Summary Usage," number 1). These numbers represent how much of the available capacity you're using for the Main Storage Processor and Control Storage Processor, respectively. You might be tempted to think that 100 percent usage means you're getting the most out of your CPU, but in reality, values higher than 60 to 65 percent usually mean that the CPU isn't able to keep up with the work load. Keep in mind that these are *averages* — you want to keep at least 20 percent (and preferably 40 percent) CPU capacity in reserve to handle fluctuations in workload. If your measurements exceed the action threshold, check for the following conditions:

- For high MSP usage, run SMFPRINT with ALL and look for one or more programs consuming most of the CPU time. These programs may require lower-level tuning or rescheduling. You also can lower the priority on CPU "hogs" to reduce their impact on other jobs.
- For high CSP usage, check to see if any FORTRAN, BASIC, or Business Graphics Utility (BGU) programs are running. These all use the S/36 Scientific Instruction Set (SIS), and consume a lot of CSP time. Check also to see if the TRACE facility is running (using the D T command); TRACE might have inadvertently been left turned on after IBM

> or third-party software maintenance. The SSP reference manual explains how to turn TRACE off. If you're running communications without MLCA or ELCA, or have a 5362/5363 model without the local Workstation Expansion Feature, the CSP is carrying the workload of these optional processors. Upgrading your CPU with a communications or workstation processor will relieve this load. Finally, check swapping rates on the SMF Summary report System Event Counters page. The CSP performs swapping and other memory management chores, so excessive swap rates (higher than 50 per minute) can overwhelm the CSP. Additional memory is the best cure.

Disk Usage ("...Summary Usage," number 2). As with CPU usage, disk usage represents the percentage of available resource capacity in use — in this case, the resource is access time. Just as with CPU usage, you want disk usage well below 100 percent; a disk working at more than 60 percent capacity means many programs are waiting in line for disk I/O completion. If your system has more than one spindle, check to make sure usage is balanced across all drives: No one drive should deviate more than 10 percent from its companions. One or more loafing drives is a common performance bottleneck, but one easily corrected by moving files to balance disk accesses. The upcoming cookbook section explains the procedure. Another solution is to add more spindles and then perform disk balancing to spread out the workload. A performance consideration sometimes overlooked when upgrading CPUs is maintaining the same spindle count. For example, moving from a 5362 with two internal 60 MB drives and two external 200 MB drives to a 5360D with two 359 MB drives may result in a drastic increase in response time, because two disk actuators are trying to carry the same load formerly shared by four actuators. A better-performing configuration would be four 200 MB drives, which would provide even more disk capacity and would cost less. If you can't trace unbalanced disk usage to file placement, and the overworked drive is A1, check the swap rates shown on the SMF Summary report's System Event Counter page. The swap area is always kept on drive A1, so excessive swapping (greater than 50 per minute) can overwork this drive. The cure, of course, is more memory.

Disk Seeks > 1/3 ("SMF Summary Report Part 2: Summary System Event Counters," number 3). This measurement, in the I/O Counters section of the SMF Summary Report, indicates the percentage of time each drive moves the disk arm farther than one-third of the maximum seek distance. Although seek distance isn't the largest component of disk access time (simply moving the arm has the highest cost), long seeks prevent SSP from effectively carrying out its disk actuator scheduling algorithm. The result is inefficient disk usage, and

slower access, even when usage rates are within the normal range. Careful file placement can reduce the problem, but maintaining good placement is a difficult and time-consuming system management chore. A better solution is to add more spindles to place frequently accessed files on separate drives, allowing for simultaneous access with less actuator motion.

Technical Note

How the CSP schedules disk I/O. You might think that the S/36 honors disk I/O requests on a first-come, first-serve basis. It doesn't; instead, it plans ahead and "schedules" requests in a way that minimizes disk actuator motion. Because the disk is so much slower than the CPU, disk requests often stack up when multiple programs are running. During a single disk I/O, lasting about 35 milliseconds, a dozen or more programs may attempt disk operations. The CSP places these requests in queues — one for each spindle — maintaining the queues in order by disk location. The CSP than removes entries from the queue to process them in location order, which shortens seek distances and can even eliminate seeks. The scheduling algorithm calls for the CSP to sweep across the disk first from low to high locations, then from high to low, eliminating the need to seek back to the start of the disk upon reaching one end. The algorithm also allows for multistep disk operations where reordering could have an adverse impact on a particular application. For example, an indexed random get requires two disk operations: one to locate the key in the index and another to retrieve the data record. The *disk arm lock* feature lets disk data management keep the arm from moving between the two operations; if the second operation isn't requested in a certain amount of time, the arm lock expires and other requests get processed. One of the SMF Summary I/O Counters, expired disk locks, records the number of times locks fail. Although this value is primarily of informational value, frequent lock expirations are a sign of an overloaded MSP.

User Area Disk Accesses (UADA) ("...Summary System Event Counters," number 4). This value represents the number of disk accesses performed for memory management purposes. Although it appears as a separate item on the SMF report System Event Counters page, it's actually just the sum of *Translated Transfer Loads, Swaps In,* and *Swaps Out.* UADA is a better measure of memory usage than the Total Storage Commitment value from the Summary Usage page. Storage Commitment indicates how much virtual memory (VM) is *set aside* for use by various programs, but not whether those programs actually use the memory. External Program Calls in particular can inflate the Storage Commitment value while actually improving performance. UADA activity reflects disk I/O required to keep the current set of executing pages in memory —

regardless of how much VM is allocated. For example, a job that uses EPCs to call 3,000 K of subprograms that it uses only infrequently would register a huge overcommitment on a 1 MB system; a low UADA rate, however, would demonstrate that the programs actually are not placing a load on the system, but are only having their activations held open. If reported UADA exceeds the action threshold for your model CPU, you should add more memory. If UADA is very low, your existing memory is underutilized; you could start or increase cache usage to take advantage of the otherwise wasted capacity.

Translated Calls/Loads Ratio ("...Summary System Event Counters," number 5). Although you've already evaluated Translated Transfer (TT) Loads as part of UADA, another aspect of this value can help you isolate particular memory problems. Comparing the ratio of TT Loads to TT Calls reveals how frequently CSP must push system programs out of memory to accommodate other requests. When the ratio of loads to calls is 5:1 or greater, system programs are being reused directly in memory without loading, saving disk I/Os. If the ratio falls below 2:1, most system programs are being used only once before they get knocked out of memory. The solution is to add more memory, or upgrade to a CPU that accommodates more memory.

Technical Note

Unlike user programs, which are always swapped out to disk when their storage is needed, most system programs can simply release their storage when necessary. System programs having *re-entrant* or *refreshable* member attributes (shown in a library directory listing) have this property, because they do not store local program variables within the same address space as the program object code. When memory for such a system program must be taken, the CSP simply seizes it; later, if the system program is called again, the CSP reloads it directly from the original library. This technique reduces disk I/O by eliminating swap-outs and #SYSTASK disk usage by not requiring a disk swap area.

Communications Line Usage ("...Summary Usage," number 6). In contrast with other usage figures, higher is sometimes better where communications are concerned. Comm line usage represents how efficiently the external data channel is being used, not how much available CPU communications resources are being consumed. Because data communications is often many times slower than even disk I/O, the overhead of pumping data in and out via this path is minimal. For batch communications, you definitely want to see usages above 90 percent, which shows you are using most of the external line bandwidth. For remote workstations, you want to keep some line bandwidth in reserve for heavy interactive use. When one user presses Enter on a remote

workstation, the resulting message uses the entire line bandwidth. If two users both press Enter simultaneously, the resulting messages must share the line, reducing the effective throughput for each message by half. If you constantly run interactive lines at more than 50 percent usage, users will see slow response times due to line sharing.

A good fix for this problem is installing a Multiline Communications Adapter (MLCA) or Eight-Line Communications Adapter (ELCA) and increasing line speed. If you already have MLCA/ELCA and have high interactive usage, consider splitting your work across two CPUs so that communications run on one machine and batch work on another. A negative, and insidious, cause of high line usages is comm retries. Most line protocols used on the S/36 incorporate error detection and correction; when a message is received corrupted, the receiver requests retransmission, and the sender executes a *retry* operation. Running on a marginal network can result in many retries, effectively using up the available bandwidth and showing high usage rates. You should periodically review the Communication Line Error rates on the Summary Usage page. Line conditions can vary over time, and a regular review can help you catch line quality problems early. Here are a few other comm management tips:

- If you're running SNA with batch file transfer or APPC, and line usage is low, increase the pacing count (also called window size) in the communications configuration (using the SETCOMM procedure). The pacing count specifies how many messages can be "in the pipeline" before SNA must wait for an acknowledgment. With good line quality, you should be able to use the highest pacing value available. Beware, though: A large pacing value on a low-quality line will increase the number of retries, degrading performance while appearing to increase usage. Verify that line errors don't rise significantly after upping the pace value.
- When running creaky, old BSC applications (using RPG T-specs), consider that short record lengths degrade performance because each record must be acknowledged. Use blocking in such programs (on the BSCA F-spec) to improve line efficiency. Better yet, update the program code to use APPC; you'll see vastly improved throughput, better reliability, and less need for operator intervention.
- Interactive programs running from remote workstations should avoid outputting a series of small screens, because a great deal of line overhead accompanies every workstation PUT operation. For example, programs that use the Starting Line Number (SLNO) feature of workstation data management to output detail lines one at a time should be rewritten to use internal arrays and a single workstation PUT.
- Sometimes reducing line speed improves performance. A line experi-

encing many errors at a particular speed might well run error-free at a lower speed, particularly if the modem uses a different modulation (analog signaling) technique at lower speeds. For example, if a 9,600 bps line experiences a 50 percent error rate, it actually has its efficiency cut by a factor of four; dropping to a 4,800 bps speed error-free would move data twice as quickly as the "flaky" 9,600 bps line did.

The Art of Tuning

Successfully tuning a S/36 requires scrupulous adherence to a few basic rules. Those rules are:

1. Make several SMF runs before changing anything.
2. Change only one variable at a time.
3. Repeat measurements after every change, under the same conditions.
4. Compare expected and actual results and keep records of your progress.

If you stick with those rules, you'll make steady progress toward improved performance. Break the rules and you'll quickly lose your way, with no point of reference against which to evaluate your progress. Under such conditions, the best you can hope for is unappreciated improvement; but you might just as easily make things much worse.

Beyond these simple rules, good tuning becomes more art than science. The next section attempts to capture the experience and knowledge of many S/36 gurus for particular tuning situations where the procedures to follow aren't obvious. Keep careful notes and expect to make your own contributions to this body of expertise. As an adjunct to SMF, you may want to use the CACHIQ (cache performance monitor), RTIMER (response time monitor), MMETER (memory meter) and SNAP (SMF snapshot display) utilities discussed elsewhere in this book.

SMF Cookbook

Each of the following six tuning "recipes" focuses on a particular problem and its solution. As with other recipes, you may need to add salt or sugar to taste. The only rules are the basic tenets outlined above — beyond that, whatever works, works!

Problem 1: High disk usage

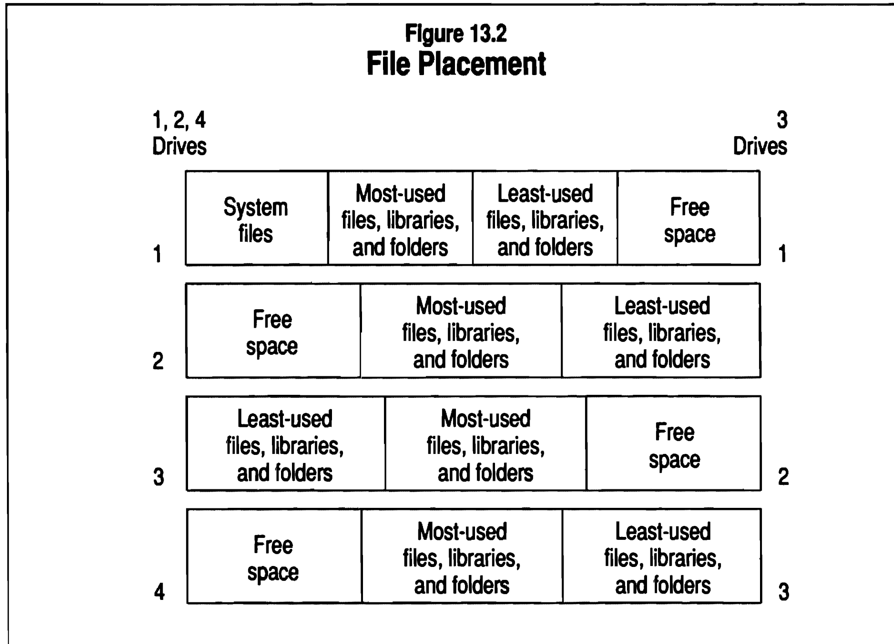
If you've read much of the other material in this book, you know that disk I/O is most often the performance bottleneck in any S/36. The easiest way to reduce disk I/O is to add more memory — SSP automatically takes advantage of whatever extra memory you install to keep reducing swapping and keep more system programs resident. This only goes so far, however: You may add so much memory that SSP's automatic usage doesn't use it all. Fortunately, you

can explicitly take advantage of additional memory with several techniques. All you need to do is make careful measurements while putting the techniques into play and you'll deduce a combination that works well in your environment. The procedure is:

1. Run SMF under various conditions to establish baseline values for your installation. Disk usage rates greater than 60 percent indicate a disk bottleneck.
2. Install as much additional memory as your model can accommodate (used memory boards are inexpensive).
3. Run SMF again under the same conditions as the baseline runs and note changes in all six primary SMF measures. You should see either no change or a reduction in MSP usage, a reduction in CSP usage and disk usage (particularly for drive A1), and lower UADA values. The translated calls/loads ratio should increase to 5:1 or better.
4. If your SMF reports show the improvements outlined above, and your UADA is very low (less than 50 per minute), you can take the following steps to better use the additional memory. Remember to take one step at a time and collect follow-up SMF measurements.
5. If you run multiple copies of the same interactive applications, turn on Memory Resident Screen Formats (MRSF) via IBM's CNFIGSSP procedure. MRSF caches screen formats the first time any job accesses them, so that jobs accessing the same formats later avoid rereading them from disk.
6. If you run concurrent batch and interactive jobs, use the CACHE procedure to activate disk caching. Disk cache may or may not work well with your job mix. The only way to know for sure is to try it. (See Chapter 15, "Caching in on Extra Memory," for detailed instructions).
7. If your interactive applications consist of many programs interconnected by OCL statements and LDA communication, convert the applications to use External Program Calls (EPCs) instead. EPCs eliminate all the disk I/O associated with OCL processing, job initialization and termination, and LDA access. (See Chapters 7 and 8 for discussions on using EPCs).

Problem 2: Unbalanced disk usage

If you have one or more "loafing" drives — indicated by uneven disk usage figures — then moving files around to even out the workload could buy back considerable performance. IBM recommends a particular general file organization for various spindle configurations. Whether the configuration ends up being the best for your installation, it's a good place to begin if you've done no



file placement so far. Figure 13.2 depicts IBM's recommendations. Each box represents a spindle, with divisions inside each box indicating the relative positions of various kinds of data. Read down the left side if you have 1, 2, or 4 spindles and the right side if you have 3 spindles. IBM's *SMF Guide* (SC21-9025), Chapter 4, explains how to physically move files, libraries, and folders to a particular drive. Identify your most-used disk objects via the SMF START option to collect I/O counters by file. After you've used those procedures to organize your disk into IBM's general plan, follow these steps to fine-tune your placement:

Performance Tip

See Chapter 5 for an alternate method of file placement using the "smart" COMPRESS strategy.

1. Run SMF and check to see if usage across drives has improved.
2. If drive A1 still has more usage than any other drive, ensure that you are not experiencing a memory bottleneck (see Problem 1). Attempting to tune a memory-constrained machine is a waste of time. Abnormal UADA or translated load/call ratios indicate memory constraints.
3. If a drive other than A1 shows excessive usage, use SMF's data collection by file to locate the most-used file, library or folder on that drive and move it to the least-used drive.

Problem 3: High CSP usage

Assuming you are not running FORTRAN, BASIC or BGU programs, which unavoidably stress the CSP, the main causes for high CSP usage are excessive

swapping or insufficient auxiliary hardware. To reduce swapping, add memory. If you're already maxed-out on memory on a smaller CPU, you should consider upgrading to the 5360 model D. If swapping isn't excessive, the problem may be that your model CPU is trying to use the CSP for too many chores that could be performed by auxiliary processors. Follow these steps to see what hardware could reduce the CSP's load:

1. On 5362/5363 models, a workstation expansion feature provides a separate workstation controller to perform polling, validity checking, and keystroke processing for local workstations (a separate workstation controller is standard equipment on 5360s). You can tell whether or not you have the workstation expansion installed by checking the Workstation Controller usage item on the Summary Usage page; if the feature is not installed, the value will read zero. To see if the workstation expansion feature can reduce CSP usage, check the Local Display Station Ops and Local Printer Ops values on the Summary I/O Counters page. If the combined counts are greater than 25 per minute, then the CSP is spending significant time on workstation I/O. Adding the workstation expansion feature can reduce CSP usage by up to 15 percent.
2. Models 5360 and 5362 support the Data Storage Controller (DSC) — an add-on processor that independently moderates data transfers between disk and memory, and between disk and tape. For both machines, the DSC comes as part of the reel-to-reel or cartridge tape drive attachment. To see if adding a DSC will reduce CSP usage, check the Disk Usage figures on the Summary Usage page. If any spindle shows usage greater than 25 percent, the CSP is spending significant time on disk data transfers, and adding a DSC can reduce CSP usage as much as 15 percent. Further, if you use streaming tape for backup, the DSC will greatly reduce response time degradation that normally occurs during tape I/O, letting you run backups concurrently with interactive jobs.
3. Models 5360 and 5362 also support MLCA (and on the 5360, ELCA) communications processors. Without these processors, the CSP must perform data link and polling chores. If you use any kind of data communications, either of these processors can reduce CSP usage by up to 10 percent.

Problem 4: High MSP usage

The MSP processes user programs (e.g., RPG applications) as well as SSP utilities (e.g., $MAINT, $COPY). High MSP usage usually stems from one or two programs performing intensive computations. To reduce MSP usage, you must identify the computationally intensive programs and modify them if possible. Use SMF's option to collect MSP usage by program to pick out the top MSP

Figure 13.3

Sample SNAP Utility Display

```
SMF SNAPSHOT SPY                                                    CURRENT TASKS
                                                ----DISK-----   PRT       WS
JOBNAME     1ST PROC    PROGRAM     MSP(S)    READ    SCAN     WRITE     OPS     OPS
            SYS TASK    CMD-PR         .00
            SYS TASK    SYS ERR        .00
W1120121    GLB010      GLB010       47.52     113     218
W1120122    GLB022      GLB022        9.72      40      55
W1120123    GLB030      GLB030       10.88      41      57
W1120124    GLB040      GLB040       10.64      45      63
W1120125    GLB044      GLB044       12.02      18      61
W1120126    GLB060      GLB060       11.19       9      50
W312015B    STS         STS           6.27
W5120247    LIBR#       TESTCQ        2.46
W2120549    LIBR#       MMETER         .00
W2121310    LOAD1       LOAD1        27.46       3     110
W1122153    SYS TASK    SMF            .02       2                 1
W1122140    SNAP        SNAP          6.70       1                                    28

**END**

Roll-Page  Cmd1-Current/Terminated tasks  Cmd10-Next screen  Cmd7-End
```

contenders. The SNAP utility (Figure 13.3, included on diskette) lets you display MSP and other usage values instantaneously without running SMF. (Simply run the SNAP procedure with no parameters to view usage statistics for all tasks). Keep in mind that SNAP does not display averages; but when identifying MSP "hogs," instantaneous readings work just as well as SMF averages. Once you've identified high-MSP-usage programs, go through the following checklist to isolate and replace the computationally intensive code:

1. Does an RPG program use binary fields or arrays, or a COBOL program COMPUTATIONAL-3 (binary) data? Binary-to-decimal conversion on input and decimal-to-binary conversion on output make intensive use of the MSP. Usually, binary representation is used to save disk space; but with the cost of disk drives so low you're probably better off rewriting programs and changing file layouts to use zoned-decimal rather than binary, even though zoned representation uses twice as much space.
2. Does an RPG program use packed fields or arrays, or a COBOL program COMPUTATIONAL-2 (packed) data? While not as MSP-intensive as binary conversion, packed data conversion can consume a lot of MSP time, especially where packed arrays are concerned. As with binary data, the solution is to change file layouts and applications to use zoned rather than packed format.

3. If the program works with character strings, does it use single-byte-element arrays, variable array indexes, and the LOKUP and MOVEA operations to perform string operations? RPG array handling with variable array elements is notoriously expensive in MSP usage, because a subroutine is called to incrementally locate the array entry based on the index variable. This represents hundreds of machine instructions *per character* processed! You can rewrite string handling to use assembler subroutines for concatenation, substring, searching and other chores (see Chapter 17, "Harnessing the Power of Assembler Routines"), or you can use the string handling operations built into ASNA's 400/RPG compiler. Either method is portable to the AS/400, which supports string operations in the native RPG compiler.

4. Does an RPG program use the SORTA operation? SORTA must be used with care, as it uses an algorithm having "quadratic" execution time (i.e., execution time increases as the square of the number of elements to be sorted). A common oversight is using SORTA on a large array that only "uses" a few elements. Regardless of the number of elements into which the RPG program actually stores data, SORTA sorts the entire array. If the sort is ascending, usually all high-order elements containing blanks will end up being sorted to the beginning of the array. You can eliminate this MSP hog by limiting SORTA to small arrays (less than 100 elements). For larger arrays, an indexed file, or a sequential file with an alternate index, will be less MSP-intensive than SORTA.

Problem 5: Sudden, unexplained response time degradation

Occasionally you may have noticed intermittent incidents of extremely poor response time that can't be explained by increased workload. Some users see such symptoms only a few times in a month, but degradation can occur several times per day in severe cases. The phenomenon has several possible causes, and you can use SMF's reporting capabilities, along with other utilities provided with this book (SNAP, SHOWUR, INDEXDR, SLOWKS), to track down the culprit. Follow these steps to determine the causes in your situation:

1. At the first report of degraded response time, run SNAP and look for one or more tasks having higher MSP or I/O usage than other tasks. This step helps rule out run-away or hog programs that might be saturating the MSP without your knowledge. If you do find a particular program that seems to have excessive MSP or I/O, check that program's SNAP measurements again once response time returns to normal. If several iterations of this testing show that poor response time correlates with high resource usage by this program, then you have a hog program on your hands. The text under Problem 4 explains how

Figure 13.4

I/O Counters Summary

```
----- S U M M A R Y   S Y S T E M   E V E N T   C O U N T E R S ------
                                                              TIME
                                          PER               MAXIMUM
                                TOTAL    MINUTE   MAXIMUM   OCCURRED
MAIN STORAGE TRANSIENT CALLS      63        3.1      23     12.02.19.358
TRANSLATED TRANSFER CALLS       2969      146.8     569     12.02.19.358
•
•
•
DISK 1 TOTAL OPS                8946      442.2     704     12.02.19.358
DISK 1 READ OPS                 3521      174.0     355     12.02.19.358
DISK 1 WRITE OPS                 278       13.7      87     12.02.19.358
DISK 1 SCAN OPS                 5147      254.4     296     12.03.19.864
DISK 1 SEEK OPS                 8074      399.1     649     12.02.19.358
DISK 2 TOTAL OPS                2003       96.3     442     12.02.19.358
DISK 2 READ OPS                 1202       48.0     201     12.02.19.358
DISK 2 WRITE OPS                 101        7.1      62     12.02.19.358
DISK 2 SCAN OPS                  700       33.8     104     12.02.19.358
DISK 2 SEEK OPS                    0        0.0       0     00.00.00.000
DISK 3 TOTAL OPS                   0        0.0       0     00.00.00.000
DISK 3 READ OPS                    0        0.0       0     00.00.00.000
DISK 3 WRITE OPS                   0        0.0       0     00.00.00.000
DISK 3 SCAN OPS                    0        0.0       0     00.00.00.000
DISK 3 SEEK OPS                    0        0.0       0     00.00.00.000
DISK 4 TOTAL OPS                   0        0.0       0     00.00.00.000
DISK 4 READ OPS                    0        0.0       0     00.00.00.000
DISK 4 WRITE OPS                   0        0.0       0     00.00.00.000
DISK 4 SCAN OPS                    0        0.0       0     00.00.00.000
DISK 4 SEEK OPS                    0        0.0       0     00.00.00.000
DISKETTE 1 READ OPS                0        0.0       0     00.00.00.000
DISKETTE 2D READ OPS               0        0.0       0     00.00.00.000
DISKETTE 1 WRITE OPS               0        0.0       0     00.00.00.000
DISKETTE 2D WRITE OPS              0        0.0       0     00.00.00.000
DISKETTE SEEK OPS                  0        0.0       0     00.00.00.000
•
•
•
TAPE 1 READ BYTES                  0 K      0.0 K     0 K   00.00.00.000
TAPE 1 WRITE BYTES                 0 K      0.0 K     0 K   00.00.00.000
TAPE 1 REWIND OPS                  0        0.0       0     00.00.00.000
TAPE 1 HITCHBACK OPS               0        0.0       0     00.00.00.000
TAPE 2 READ BYTES                  0 K      0.0 K     0 K   00.00.00.000
TAPE 2 WRITE BYTES                 0 K      0.0 K     0 K   00.00.00.000
TAPE 2 REWIND OPS                  0        0.0       0     00.00.00.000
TAPE 2 HITCHBACK OPS               0        0.0       0     00.00.00.000
•
•
•
```

to improve performance for such programs.

2. If a single program isn't responsible for the problem, run SMF during a period of slow response time (you may need to leave SMF running "pre-emptively" throughout the day if slow response periods are unpredictable). Obtain a summary report for the time period of the problem.

3. In the summary report, I/O counters section (Figure 13.4), look for high values for a particular disk drive, or for diskette or tape I/O counts. Uneven disk usage for a particular drive indicates one file is probably having an abnormally high number of reads or writes; you'll need to get an ALL report to track down the problem file (see step 4). Any diskette or tape I/O during interactive operations can have a severe impact on response time if you don't have a DSC installed. If you see disk or tape activity, but no DSC usage on the Device Usage Rates page, then you don't have a DSC installed (only the 5362 and 5360 support the DSC). You must either reschedule the diskette or tape I/O, or upgrade to a DSC-equipped CPU.

4. If a particular disk drive has much higher I/O than its companions, obtain an ALL report, which displays access counters for each file and identifies the drive containing the file (Figure 13.5a). Find the file (or files) on the problem drive having the highest disk activity. Then examine the Task Status page (Figure 13.5b), checking the disk I/O counters for the problem drive on each program that accesses the problem files. One or two programs probably account for the majority of accesses.

5. Now you must determine if the excessive I/O for the problem programs identified in step 4 are related to program logic, access conflicts, or DDM maintenance actions. The cause could be a high volume of record accesses on the part of your program, record-waits due to conflicting batch jobs, or the result of DDM invoking a slow index maintenance function.

6. If the cause is a high volume of record accesses, SMF's File Access Counters for the problem file will show high numbers in the GET, UPDATE, DELETE or ADD operations. You should examine the program's logic to see under what conditions it can perform excessive file operations. Profiling the program (see Chapter 18, "Profiling and Advanced Debugging") is one way to empirically identify the problem section of code. You may discover a coding error that results in superfluous disk I/O, in which case repairing the bug should fix the problem. Otherwise, you will have to change the program's design to reduce disk I/O.

Figure 13.5a

File Access Counters by File from SMFPRINT ALL

-------------------------------- U S E R   F I L E   A C C E S S   C O U N T E R S --------------------------------

FILE LABEL	DATE CREATED	JDBNAME	FILE TYPE	FILE DRG	BLOCK LDC	LENGTH	DISK LDC	--DATA--- READ	WRTE	----INDEX----- READ	SCAN	WRTE	REC WTS	--GET--- LDG	PHYS	-UPDATE- LOG	PHYS	-DELETE- LOG	PHYS	--ADD--- LOG	PHYS
GLBTRAN	92/09/12		R	I	22682	614	A1	101	0	0	247	0	0	7498	348	0	0	0	0	0	0
GLBMAST	92/09/12		R	I	10377	101	A1	5	0	0	5	0	0	220	10	0	0	0	0	0	0
GLBTBL	92/09/12		R	I	18055	17	A1	1	0	0	0	0	0	1	1	0	0	0	0	0	0
GLBDETL	92/09/12		R	I	84202	90	A2	88	0	0	50	0	0	300	138	0	0	0	0	0	0
GLBFORM	92/09/12		R	I	88540	33	A2	12	0	0	12	0	0	66	24	0	0	0	0	0	0

Figure 13.5b

File Access Counters by Task from SMFPRINT ALL

-- T A S K   S T A T U S --

.
.
.

	JOB	PROC/TYPE	MSP USAGE	-----DISK 1------ READ	SCAN	WRITE	-----DISK 2------ READ	SCAN	WRITE	-----DISK 3------ READ	SCAN	WRITE	-----DISK 4------ READ	SCAN	WRITE
1		CMD-PR	0%	0	0	0	0	0	0	0	0	0	0	0	0
2		SYS ERR	0%	0	0	0	0	0	0	0	0	0	0	0	0
3	W2120118	SMF	0%	0	0	1	0	0	0	0	0	0	0	0	0
4	W1120912	STS	5%	0	0	0	0	0	0	0	0	0	0	0	0
5	W3120158	STS	3%	0	0	0	0	0	0	0	0	0	0	0	0
6	W2120549	LIBR#	0%	0	0	0	0	0	0	0	0	0	0	0	0
7	W1120121	GLB010	32%	46	89	0	0	0	0	0	0	0	0	0	0
8	W5120247	LIBR#	2%	0	0	0	0	0	0	0	0	0	0	0	0
9	W1120123	GLB030	8%	20	28	0	0	0	0	0	0	0	0	0	0
10	W1120128	GLB080	10%	5	25	0	0	0	0	0	0	0	0	0	0
11	W1120124	GLB040	8%	21	29	0	0	0	0	0	0	0	0	0	0
12	W1120125	GLB044	10%	8	28	0	0	0	0	0	0	0	0	0	0
13	W1120122	GLB022	6%	18	25	0	100	62	0	0	0	0	0	0	0
14	W2121310	LOAD1	10%	0	23	0	0	0	0	0	0	0	0	0	0

7. The second cause leads to a large number in the Record Waits (REC WTS) column of the File Access Counters for the problem file. Record waits occur when one or more programs try to read a record for update and another program already owns the record. Concurrently running two or more batch programs that both update the same file can result in a "lockstep" phenomenon, in which none of the programs actually encounter a "deadly embrace" (permanent lockout), but each program attempts to access the same records for update at about the same time (Chapter 11 provides a detailed discussion about avoiding the deadly embrace). DDM forces the programs to read and update the records serially, resulting in the high Record Wait count. DDM's internal record-lock resolution algorithm uses considerable disk and MSP resources, so a high number of waits affects overall system

performance. The solution is to run the programs separately.

8. The last cause — DDM index maintenance — is particularly tricky to isolate. Four situations exist where DDM can perform excessive disk I/O without prior warning, and no SMF values are of any help identifying the culprit. (Chapter 3, "Inside Disk Data Management," provides additional information on each of these situations).

   In the first situation, a program opens a file that previously had delayed index maintenance — keys were added to the overflow but not sorted. DDM must sort the overflow area before giving the program control, a process that can take several minutes for a few thousand added keys. The solution is to keysort files manually when possible to avoid unexpected program initiation delays. The INDEXDR utility (Chapter 11) lets you identify such files.

   The second situation occurs when a program updates a parent file record that updates a key in one of the alternates defined over the parent. If the changed key value has a large number of duplicates already in the index, DDM must move many keys to insert the new key in RRN sequence within the duplicate key string. The most common example of this is changing an alternate-index key field to blanks. If hundreds or thousands of duplicate keys with the same value exist, the insert operation can take from minutes to hours to complete. The solution is to avoid generating long strings of duplicate keys in alternate indexes.

   The third situation results when DDM tries to insert a key in an index that has gaps in it, but there are no gaps left between the insertion point and the end of the index. The index is not really full, so DDM can't return "file full" to the application. DDM is forced to perform a "degapping operation" in which all the index entries in the overflow area are rewritten to collect free space at the end of the index. To accomplish this in a timely manner, DDM suspends the operation of all other programs using the file until the degapping is completed. Degapping usually takes only a few seconds to a few minutes, even for large files; during that time, however, the system can appear to be "hung." The solution is to allocate large indexed files with at least 10 percent "breathing room" to reduce the chance that embedded gaps get used up.

   The fourth situation occurs during keysort, when SSP can't find enough disk space for a sort work area. In this situation, SSP reverts to an "in-memory" keysort that needs no sort workfile. Unfortunately, an in-memory keysort is both MSP-intensive and extremely inefficient — it can take days to complete. The SLOWKS utility

provided with this book will identify an in-memory keysort. See Chapter 12 for details about using SLOWKS to solve this problem.

Problem 6: File and index blocking don't seem to be helping performance
Chapter 10 explains how to use DBLOCK and IBLOCK to improve individual program performance. To be sure that you're using effective blocking factors, though, you should measure program I/O usage and overall system I/O usage before and after each blocking change. The following procedure shows how this is done:

1. Before setting any blocking factors, run SMF to collect statistics by task and file while the program is executing over a known dataset *with no blocking* specified. The easiest way to do this is to use IBLOCK-1 and DBLOCK-1 for the file in question. Don't try to measure blocking effectiveness for more than one file at a time, and do not let other programs use the file during your test (you may leave the file as DISP-SHR, though).
2. Record the logical and physical counts from the User File Access Counters page for the file in question. These are the counts under the GET, UPDATE, DELETE and ADD column headings. Also record the physical I/O counters for each disk drive for the program in question.
3. Set DBLOCK and IBLOCK to factors determined by the guidelines in Chapter 10. Then rerun the SMF collection and tests, taking care to use exactly the same input data with no other programs using the file under test.
4. Compare the statistics resulting from your second test run with the baseline values you collected in step 1. The number of logical file operations should remain the same, while the total physical operation counts decline. The total number of disk operations for the program should also decline. If the total physical file operations remain the same, then this program isn't a good candidate for blocking. If the physical file operations counts decline, but total disk operations for the program goes up, blocking is causing increased disk I/O in some other aspect of the program's operation (Chapter 10 delineates the cases where this occurs). You should repeat steps 3 and 4 every time you try a different set of IBLOCK and DBLOCK values.

What You Don't Know *Can* Hurt You

If there's any one truism to learn from this chapter, it's that you can't tune a system in the dark. Achieving good performance requires planned, accurate

measurements and informed analysis. The procedures outlined here let you cut through the complexity of SMF and make effective tuning decisions.

References

Willkomm, Kenneth L. "Taking Your System's Measure," *NEWS 3X/400*, December, 1990.

——. "Check and Balance," *NEWS 3X/400*, January, 1991.

——. "Time to Worry?" *NEWS 3X/400*, February, 1991.

——. "Secondary Considerations," *NEWS 3X/400*, March, 1991.

——. "Up Close and Down the Road," *NEWS 3X/400*, April, 1991.

——. "Beyond SMF," *NEWS 3X/400*, May, 1991.

——. "Tips for Top Performance," *NEWS 3X/400*, June 1991.

SMF Summary Report Part 1: Summary Usage

```
----------------------  S U M M A R Y   U S A G E  --------------------
                                   AVERAGE  MAXIMUM  TIME MAXIMUM OCCURRED
①  MAIN STORAGE PROCESSOR            92 %    96 %    12.18.20.113
   CONTROL STORAGE PROCESSOR         76 %    79 %    12.00.21.374
   WORKSTATION CONTROLLER QUEUE.      0 %     0 %    00.00.00.000
   WORKSTATION CONTROLLER             0 %     0 %    00.00.00.000
   WORKSTATION CONTROLLER 2 QUEUE     0 %     0 %    00.00.00.000
   WORKSTATION CONTROLLER 2           0 %     0 %    00.00.00.000
   PC PROCESSOR                       0 %     0 %    00.00.00.000
   DATA STORAGE CONTROLLER            0 %     0 %    00.00.00.000
   DATA STORAGE ATTACHMENT            7 %    10 %    12.02.19.358
②  DISK 1                            71 %    86 %    12.21.32.853
   DISK 2                            14 %    30 %    12.21.32.863
   DISK 3                             0 %     0 %    00.00.00.000
   DISK 4                             0 %     0 %    00.00.00.000
③  COMMUNICATION LINE 1              40 %    98 %    12.00.21.374
   COMMUNICATION LINE 2              ** %    ** %    NOT ACTIVE/COLLECTED
   COMMUNICATION LINE 3              ** %    ** %    NOT ACTIVE/COLLECTED
   COMMUNICATION LINE 4              ** %    ** %    NOT ACTIVE/COLLECTED
   COMMUNICATION LINE 6              ** %    ** %    NOT ACTIVE/COLLECTED
   COMMUNICATION LINE 6              ** %    ** %    NOT ACTIVE/COLLECTED
   COMMUNICATION LINE 7              ** %    ** %    NOT ACTIVE/COLLECTED
   COMMUNICATION LINE 8              ** %    ** %    NOT ACTIVE/COLLECTED
   COMMUNICATION LINE 1 ERRORS       ** %    ** %    NOT ACTIVE/COLLECTED
   COMMUNICATION LINE 2 ERRORS       ** %    ** %    NOT ACTIVE/COLLECTED
   COMMUNICATION LINE 3 ERRORS       ** %    ** %    NOT ACTIVE/COLLECTED
   COMMUNICATION LINE 4 ERRORS       ** %    ** %    NOT ACTIVE/COLLECTED
   COMMUNICATION LINE 5 ERRORS       ** %    ** %    NOT ACTIVE/COLLECTED
   COMMUNICATION LINE 6 ERRORS       ** %    ** %    NOT ACTIVE/COLLECTED
   COMMUNICATION LINE 7 ERRORS       ** %    ** %    NOT ACTIVE/COLLECTED
   COMMUNICATION LINE 8 ERRORS       ** %    ** %    NOT ACTIVE/COLLECTED
   TASK WORK AREA                    23 %    24 %    12.13.24.131
   DISK CACHE                         0 %     0 %    00.00.00.000
   ASSIGN/FREE SPACE                 74 %    80 %    12.04.20.411
   TOTAL STORAGE COMMITMENT          23 %    37 %    12.21.27.090
   ACTIVE STORAGE COMMITMENT         87 %    91 %    12.13.24.131
   ACTUAL STORAGE COMMITMENT         23 %    37 %    12.21.27.090
```

SMF Summary Report Part 2: Summary System Event Counters

------- S U M M A R Y   S Y S T E M   E V E N T   C O U N T E R S -------

	TOTAL	PER MINUTE	MAXIMUM	TIME MAXIMUM OCCURRED
MAIN STORAGE TRANSIENT CALLS	63	3.1	23	12.02.19.358
⑤ TRANSLATED TRANSFER CALLS	2969	146.8	569	12.02.19.358
ASYNCHRONOUS TRANSFER CALLS	71	3.5	14	12.02.19.358
MAIN STORAGE TRANSIENT LOADS	28	1.4	9	12.02.19.358
⑤ TRANSLATED TRANSFER LOADS	60	3.0	22	12.02.19.358
MAIN STORAGE LOADER REQUESTS	29	1.4	9	12.02.19.358
SWAPS IN	110	5.4	12	12.02.19.358
SWAPS OUT	0	0.0	0	00.00.00.000
SWAPS OUT, FORCED	0	0.0	0	00.00.00.000
TASK WORK AREA READ OPS	121	6.0	39	12.02.19.358
TASK WORK AREA WRITE OPS	124	6.1	49	12.02.19.358
MAIN STORAGE CLEAR OPS	113	5.6	46	12.02.19.358
CONTROL STORAGE TRANSIENT CALLS	2474	122.3	646	12.02.19.358
CONTROL STORAGE TRANSIENT LOADS	393	19.4	53	12.02.19.358
CONTROL STORAGE LOADER REQUESTS	0	0.0	0	00.00.00.000
SPOOL SEGMENTS ALLOCATED	0	0.0	0	00.00.00.000
SPOOL ENTRIES ALLOCATED	0	0.0	0	00.00.00.000
SPOOL EXTENTS ALLOCATED	0	0.0	0	00.00.00.000
GENERAL WAITS	425	21.0	154	12.12.23.797
DISK RECORD WAITS	0	0.0	0	00.00.00.000
TASK WORK AREA EXTENTS	0	0.0	0	00.00.00.000
JOB INITIATIONS	12	0.6	3	12.06.21.374
JOB STEP INITIATIONS	22	1.1	8	12.02.19.358
MRT ATTACHES	0	0.0	0	00.00.00.000
MRT LOADS	0	0.0	0	00.00.00.000
JOB TERMINATIONS	9	0.4	2	12.06.21.374
JOB STEP TERMINATIONS	3	0.1	1	12.02.19.358
ABNORMAL TERMINATIONS	6	0.3	2	12.09.22.541
DISK LOCKS SATISFIED	1610	79.6	103	12.03.19.864
DISK LOCKS EXPIRED	3338	165.0	176	12.03.19.864
ASSIGN/FREE EXTENSIONS	14	0.7	6	12.02.19.358
ASSIGN/FREE REDUCTIONS	6	0.3	2	12.07.21.790
PREEMPTIVE TASK DISPATCHES	200431	9907.6	11877	12.19.26.430
RESOURCE TIMEOUTS	1954	96.6	129	12.03.19.864
MAIN STORAGE PROCESSOR TIMEOUTS	2058	101.7	124	12.19.26.430
WKSTN BUFFER READ RETRIES	3	0.1	1	12.02.19.358
L-1 STORAGE RELEASES W/O SWAP	0	0.0	0	00.00.00.000
L-1 STORAGE RELEASES W/ SWAP	0	0.0	0	00.00.00.000
L-2 STORAGE RELEASES W/O SWAP	0	0.0	0	00.00.00.000
L-2 STORAGE RELEASES W/ SWAP	0	0.0	0	00.00.00.000
L-3 STORAGE RELEASES W/O SWAP	0	0.0	0	00.00.00.000
L-3 STORAGE RELEASES W/ SWAP	0	0.0	0	00.00.00.000
L-4 STORAGE RELEASES W/O SWAP	0	0.0	0	00.00.00.000
L-4 STORAGE RELEASES W/ SWAP	0	0.0	0	00.00.00.000
MEMORY RESIDENT OVERLAY LOADS	0	0.0	0	00.00.00.000
MEMORY RESIDENT OVERLAY MAPS	0	0.0	0	00.00.00.000
DISK CACHE HITS	0	0.0	0	00.00.00.000
DISK CACHE MISSES	0	0.0	0	00.00.00.000
④ USER AREA DISK ACTIVITY	170	8.4	34	12.02.19.368

Continued

SMF Summary Report Part 2: Summary System Event Counters *(continued)*

------- S U M M A R Y   S Y S T E M   E V E N T   C O U N T E R S -------

	TOTAL	PER MINUTE	MAXIMUM	TIME MAXIMUM OCCURRED
NOT USED	0	0.0	0	00.00.00.000
NOT USED	0	0.0	0	00.00.00.000
NOT USED	0	0.0	0	00.00.00.000
NOT USED	0	0.0	0	00.00.00.000
NOT USED	0	0.0	0	00.00.00.000
NOT USED	0	0.0	0	00.00.00.000
NOT USED	0	0.0	0	00.00.00.000
NOT USED	0	0.0	0	00.00.00.000
NOT USED	0	0.0	0	00.00.00.000
NOT USED	0	0.0	0	00.00.00.000
NOT USED	0	0.0	0	00.00.00.000
DISK 1 TOTAL OPS	8946	442.2	704	12.02.19.358
DISK 1 READ OPS	3521	174.0	355	12.02.19.358
DISK 1 WRITE OPS	278	13.7	87	12.02.19.358
DISK 1 SCAN OPS	5147	254.4	296	12.03.19.864
DISK 1 SEEK OPS	8074	399.1	649	12.02.19.358
DISK 2 TOTAL OPS	0	0.0	0	00.00.00.000
DISK 2 READ OPS	0	0.0	0	00.00.00.000
DISK 2 WRITE OPS	0	0.0	0	00.00.00.000
DISK 2 SCAN OPS	0	0.0	0	00.00.00.000
DISK 2 SEEK OPS	0	0.0	0	00.00.00.000
DISK 3 TOTAL OPS	0	0.0	0	00.00.00.000
DISK 3 READ OPS	0	0.0	0	00.00.00.000
DISK 3 WRITE OPS	0	0.0	0	00.00.00.000
DISK 3 SCAN OPS	0	0.0	0	00.00.00.000
DISK 3 SEEK OPS	0	0.0	0	00.00.00.000
DISK 4 TOTAL OPS	0	0.0	0	00.00.00.000
DISK 4 READ OPS	0	0.0	0	00.00.00.000
DISK 4 WRITE OPS	0	0.0	0	00.00.00.000
DISK 4 SCAN OPS	0	0.0	0	00.00.00.000
DISK 4 SEEK OPS	0	0.0	0	00.00.00.000
DISKETTE 1 READ OPS	0	0.0	0	00.00.00.000
DISKETTE 2D READ OPS	0	0.0	0	00.00.00.000
DISKETTE 1 WRITE OPS	0	0.0	0	00.00.00.000
DISKETTE 2D WRITE OPS	0	0.0	0	00.00.00.000
DISKETTE SEEK OPS	0	0.0	0	00.00.00.000
72MD AUTO LOADER REQUESTS	0	0.0	0	00.00.00.000
DISKETTE HEAD CONTACT REVS	0	0.0	0	00.00.00.000
LOCAL DISPLAY STATION OPS	956	47.3	73	12.06.21.374
LOCAL PRINTER OPS	0	0.0	0	00.00.00.000
REMOTE DISPLAY STATION OPS	0	0.0	0	00.00.00.000
REMOTE PRINTER OPS	0	0.0	0	00.00.00.000
3262 PRINTER OPS	0	0.0	0	00.00.00.000
1255 MICR OPS	0	0.0	0	00.00.00.000
TAPE 1 READ BYTES	0 K	0.0 K	0 K	00.00.00.000
TAPE 1 WRITE BYTES	0 K	0.0 K	0 K	00.00.00.000
TAPE 1 REWIND OPS	0	0.0	0	00.00.00.000
TAPE 1 HITCHBACK OPS	0	0.0	0	00.00.00.000
TAPE 2 READ BYTES	0 K	0.0 K	0 K	00.00.00.000

Continued

SMF Summary Report Part 2: Summary System Event Counters *(continued)*

```
------- S U M M A R Y   S Y S T E M   E V E N T   C O U N T E R S -------
                                                                TIME
                                                PER            MAXIMUM
                                      TOTAL   MINUTE  MAXIMUM  OCCURRED
  TAPE 2 WRITE BYTES                    0 K     0.0 K    0 K   00.00.00.000
  TAPE 2 REWIND OPS                     0       0.0      0     00.00.00.000
  TAPE 2 HITCHBACK OPS                  0       0.0      0     00.00.00.000
③DISK 1 SEEK OPS GT 1/3 DISK         20.7 %    ****   46.2 %   12.02.19.358
  DISK 2 SEEK OPS GT 1/3 DISK         0.0 %    ****    0.0 %   00.00.00.000
  DISK 3 SEEK OPS GT 1/3 DISK         0.0 %    ****    0.0 %   00.00.00.000
  DISK 4 SEEK OPS GT 1/3 DISK         0.0 %    ****    0.0 %   00.00.00.000
  DISK 1 AVERAGE SEEK LENGTH        153 CYL    ****  294 CYL   12.02.19.358
  DISK 2 AVERAGE SEEK LENGTH          0 CYL    ****    0 CYL   00.00.00.000
  DISK 3 AVERAGE SEEK LENGTH          0 CYL    ****    0 CYL   00.00.00.000
  DISK 4 AVERAGE SEEK LENGTH          0 CYL    ****    0 CYL   00.00.00.000
```

Chapter 14

Do You Need More Memory?

We've already advocated that adding memory is the cheapest, easiest way to improve performance on the S/36. However, it's unwise to blindly believe that adding more memory will solve *all* your performance problems. It's also unwise to install memory to alleviate problems that can be solved by a little application tuning. What you really need is a memory dipstick — something to confirm your sense that your S/36 is a quart low on memory.

There may be times when your application memory requirements simply exceed all the real memory available — causing performance-robbing memory swapping. This may be due to improper record blocking, programs that are too large, or improper job scheduling. Understanding exactly how the S/36 uses memory and determining whether or not your system has enough memory are daunting tasks. But with a basic knowledge of how the S/36 uses memory, you'll be in a position to evaluate when it needs more memory and how to make the best use of that additional memory. Think of this chapter as your memory dipstick.

Where Does it All Go?

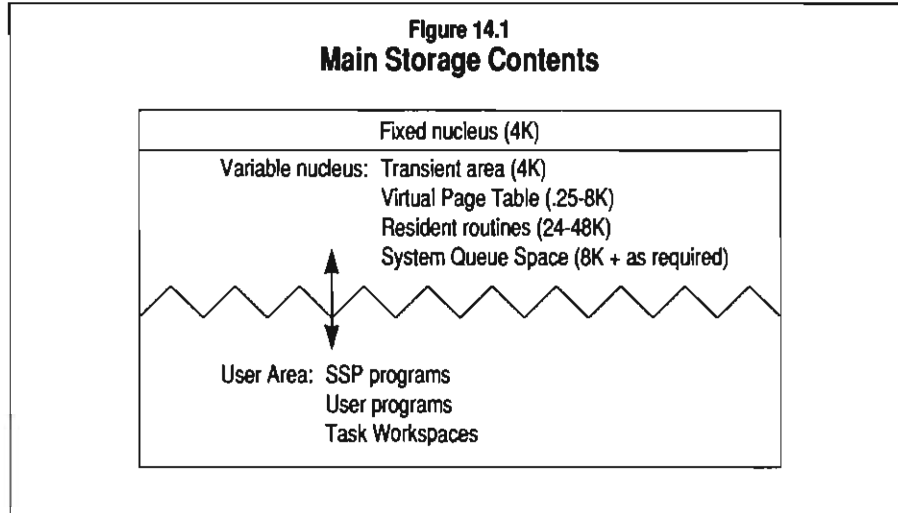
Before deciding whether or not you need more memory, let's briefly review Chapter 2's discussion of how S/36 memory is organized. If you'll recall, the S/36's main memory is logically organized into three distinct areas (Figure 14.1). First, the fixed nucleus, which occupies the first 4 K of main memory, contains variables and data structures needed by the SSP and CSP. Next, the variable nucleus contains the transient area, virtual page table, frequently used memory-resident SSP routines such as Disk Data Management and Workstation Data Management, and System Queue Space. Finally, the user area, the largest of the three, contains user programs, non-resident SSP programs, file buffers, screen formats, and other various temporary system and user objects. If you didn't get a firm understanding of main memory organization the first time through, now would be a good time to review Chapter 2 before you continue.

Performance Tip

Cache can yield performance improvements. But beware that allocating memory to cache reduces memory available for user programs. See Chapter 15 for a detailed explanation of using cache effectively.

Let's look next at some specific instances where memory can be chewed up in a big way. One likely culprit is the cache. Although caching can yield dramatic performance benefits, it can also impede performance just as substantially if too much memory is allocated to the cache. When the cache is overallocated, the system wastes a lot of time maintaining "stale" cache pages; overallocation also increases User Area Disk Activity (more on this in a minute). It's easy to overallocate memory to cache, but fortunately you can use a number of methods to determine whether or not you're getting

Figure 14.1
Main Storage Contents



the most "bang for your byte" from cache. Refer to Chapter 15 for detailed information about using cache effectively.

IBM's office products, such as DisplayWrite/36 (DW/36), Query/36 and Personal Services/36 all consume a large amount of virtual memory (VM). Especially DW/36, which assigns a large amount of VM for each concurrent user. As you already know, the S/36 can commit almost unlimited amounts of memory using the built-in virtual paging mechanism. But this can also degrade performance. If not enough real memory is available, the system will constantly page programs, system transients, and buffers in and out of real memory, significantly increasing disk activity. For this reason, you should keep concurrent office product users to a minimum if at all possible to avoid costly memory overcommitments.

As you read in Section III, External Program Calls (EPCs) can also chew up a lot of VM. Although subprograms are only paged in when they're called, if more real memory is available, these subprograms, and the system transients that call them, stand a better chance of staying in memory longer, rather than constantly being paged in and out to real memory.

Another potential VM problem only affects users of ASNA's ACCELER8 product. Since ACCELER8 uses a B-tree method of indexing, it keeps "nodes" of index entries in each file's index area. When these files are opened by your programs, the system ignores the IBLOCK value in the // FILE OCL statement, and instead allocates its own fixed-length index block buffer to hold the "root" node. The problem is that ACCELER8's fixed-length buffer is 2 K for every indexed file or alternate in your program. And this buffer is assigned regardless of whether or not there's any IBLOCK value in the OCL. This means that a

program that once ran in, say, 50 K with no Task Work Space (TWS) buffers may now run in 60 K with 10 K of TWS buffers. (See Chapters 3 and 10 for more information on TWS buffers.) Just as with DW/36 and EPCs, TWS buffers can cause paging, thus degrading performance. The TWS problem may very well outweigh the performance gains you may see with ACCELER8; so one solution would be to purchase additional memory, take a better look at those programs that are going into TWS, and either make them more modular or decrease their DBLOCK values.

Other likely culprits for eating up memory are your own application programs. Inefficient use of DBLOCK or IBLOCK can cause your programs to be much larger than they might normally need to be. Refer to Chapter 10 for a complete look at using DBLOCK and IBLOCK effectively.

Finally, and probably most obviously, if your system has a large number of online users and batch jobs, it's a natural assumption that all those programs are going to take up memory. Especially if you have several programmers at work who generally use dual-session displays or PCs with emulation cards. Each programmer can easily be running several jobs at once such as compiles, edits, test programs, debug sessions, and so on. If this gets to be a problem, it's probably time to consider getting the programmers a S/36 of their own. (See Chapter 6 for a complete discussion on suggested "programmer" machines.)

Performance Tip

If you experience Task Work Area extent messages, before you do anything else, use CNFIGSSP to ensure that TWA is set to its maximum of 6553 blocks. See Chapter 7 for more about the suggested size of TWA.

The Warning Signs

A few red flags should not be overlooked — they may indicate your system needs more memory. First, response time will be sluggish. If response times increase and decrease as the day wears on, this may well be a sign of memory overcommitment. Second, batch processing will be slow. Just as with interactive response times, if you find batch jobs are taking longer than usual, this is another red flag. Next, Task Work Area (TWA) extent messages may appear at the system console. If this starts happening frequently, you should either investigate your usage of DW/36 jobs or EPCs, or pick up the phone and start calling hardware brokers for a S/36 memory quote.

Finally, pay close attention to certain values on your SMF reports. Specifically, if the value for the User Area Disk Activity (UADA) is larger than normal, this is one of the surest signs that the system needs more memory. UADA is really the sum of Translated Transfer Loads, Swaps In, and Swaps Out. If this number exceeds the action threshold for your system, there's a problem. Another SMF value to pay close attention to is the Translated Transfer Loads-to-Calls ratio. This value reflects the number of times the CSP had to kick system programs out of memory to make room for something else. If this ratio is 2:1 or less, systems programs are only being used once, or not at all, before they get kicked out of memory. The solution is to add more memory. Refer to Chapter 13 for a complete discussion on the meanings of these

important values and detailed instructions for getting the most from SMF and its reports.

Memory Meter

In addition to using SMF, we've included a handy interactive memory monitoring utility on the "Desktop Guide" diskette called MMETER. MMETER gives a real-time account of how nucleus pages, user main and subprograms, system programs, and system workspaces are using your system's memory. MMETER's real-time memory account helps you track down intermittent memory-related performance problems that are sometimes difficult to spot through SMF reports. For example, if you experience occasional drastic increases in response time at unpredictable intervals of days, or even weeks, you may not be able to establish a useful measurement with SMF. To check your system's "normal" memory use, use MMETER when system response time is good. You then can compare memory use during good response times to memory use during slow response times, easily determining if memory overcommitment is a possible source of trouble.

Performance Tip

MMETER reports ***actual*** main storage size, not configured main storage size. Use MMETER after additional memory is installed to verify proper installation.

To invoke MMETER, key the procedure name MMETER and the screen shown in Figure 14.2 appears. To update the screen, press Enter; to exit, press Command Key 7. Let's examine the MMETER screen in detail.

At the top of the screen are three column headings. *Count* is the number of programs, pages, or workspaces for the line item; *Total committed* refers to the total kilobytes and percentage of virtual storage for each line item; and *Currently paged in* refers to the kilobytes and percentage of real memory currently used by each line item. Note that *Total committed* can considerably exceed the total amount of real memory physically installed on the machine — a prime example of the S/36's virtual memory management scheme at work.

The five detailed line items are defined as follows: *Nucleus pages* consist of the fixed and variable nucleus areas, which are used by the system and always reside in real storage. The amount of memory that nucleus pages consume changes as the system gives and takes pages to and from the user area.

User programs consist of all user application programs and SSP utilities such as $MAINT, $COPY, or compilers. *System programs* are SSP programs (e.g., spool writers, the initiator, the command processor, system transients) called by other system programs to perform repetitive tasks. They run transparent to the user, and as a result, are excluded from the D U display. The fourth line item, *Task workspaces*, is of interest only to those using External Program Calls. Generally, "stock" S/36 application programs do not have any subprograms, and determining the memory used by subprograms is not possible through any IBM-supplied utility.

The last line item is *System workspaces*, pages of memory the system uses for storing various tables, such as the active procedure list, active screen

Figure 14.2
Memory Meter Screen

System/36 Memory Meter

	Count	Total committed		Currently paged in	
Nucleus pages.........	74	148K	29%	148K	28%
User programs.........	12	508K	99%	92K	17%
System programs.......	195	448K	88%	60K	12%
Task workspaces.......	40	1280K	250%	204K	40%
System workspaces.....	7	54K	11%	8K	2%
TOTALS................		2,438K	476%	512K	100%
(unused memory)....				0K	
(main storage size)				512K	

formats, and program buffers like those created when a program exceeds its 64 K address space and must place file buffers into the TWA.

MMETER's individual line items help you isolate the memory requirement for user programs, called programs, or system activity. A high memory overcommitment for either the User programs or Task Workspaces is due to the application workload directly under your control — reducing the workload will help even out response-time peaks. Excessive overcommitment in the System workspaces line item usually results from heavy use of IBM Office products such as DW/36. To improve performance, either you must add more memory or schedule heavy Office product use for off-peak hours. Overcommitment of memory to the Nucleus pages or System programs line item usually is caused by high SSP activity, either through a large volume of procedure interpretation, or a large number of medium-lived System Queue Space items. If you don't add memory to relieve the high memory requirement, you probably need to modify your programs to reduce their dependency on SQS or procedure execution.

The totals for the line items show the cumulative kilobytes and percentage of committed memory and the kilobytes and percentage of real memory currently being used by the system. As previously stated, the amount committed can be many times the amount used. Also, the total unused memory is often 0 K; but having no used memory is not necessarily cause for concern. It usually means that your installed memory is being used to its fullest potential, thereby providing maximum benefits in throughput and response times.

However, if the total number of kilobytes is consistently high, you might want to consider adding more memory to your machine. For example,

if the normal total memory commitment for your system is 230 percent, but it increases to 500 percent during slow response times, purchasing additional memory will probably help alleviate the problem.

Chapter 15

Caching in on Extra Memory

Although SSP automatically takes advantage of extra memory to improve performance, it can only go so far on its own. As you've read in other chapters, you can direct SSP to use additional memory for specific purposes, such as memory-resident screen formats, data and index blocking, and external program calls. Even with these added uses, however, it's possible to leave memory a-wasting, particularly on 5360 machines with more than 2 MB of memory. When memory is not being used to its fullest, it's a good time to consider turning on S/36 disk cache, which can put otherwise-unused memory to work reducing clock-killing disk I/O operations. In this chapter you'll learn how disk cache works and under what conditions it improves performance. You'll also learn how to monitor cache performance with a clever utility called CACHIQ, and how to automatically change the cache configuration to accommodate changing workloads.

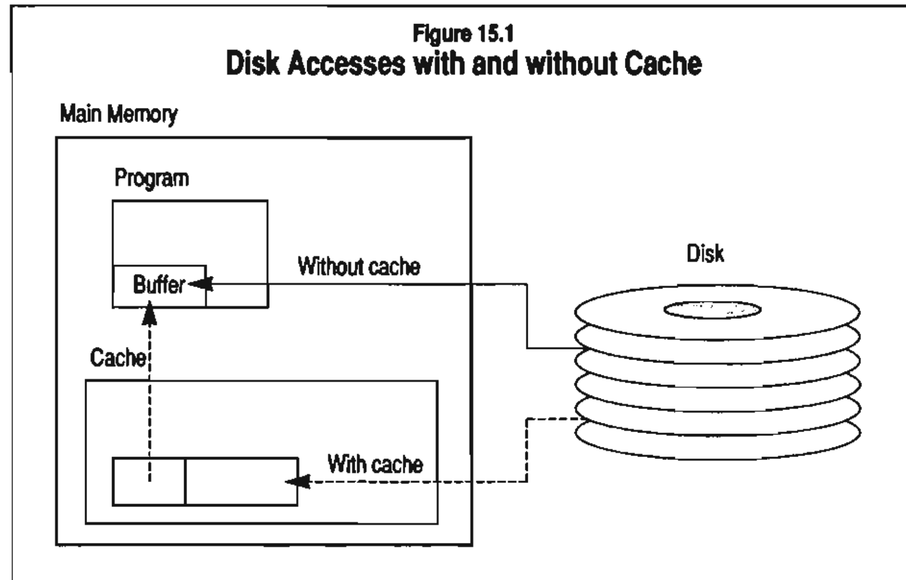
How Cache Works

The disk cache is a buffer in main storage that stands between the disk drives and your program's data buffers (Figure 15.1). With disk cache enabled, instead of reading data straight into a program's storage, SSP reads the data into the cache and then copies the data into the program buffer. Later, if another program needs that same data, SSP can simply copy it from the cache, saving the time of a disk read operation. Each disk read request satisfied by copying from the cache is called a *cache hit.* Whenever a disk read can't be found in the cache, the data is read from disk into the cache, displacing previously read data; this is called a *cache miss.* A cache hit is about 10 times faster than a cache miss. Although a read from disk under cache takes longer than if cache weren't turned on, just one cache hit makes up for that extra time in spades, resulting in improved performance. If no other program ever needs the cached data, however, the extra work of caching is wasted and performance suffers. Unlike blocked records, which are owned by an application, cached data is shared by all users — no one application owns the cache.

A phenomenon called *locality* makes caching effective. The idea of locality comes from the observation that disk accesses tend to be clustered, so that once a location is referenced, it's likely that nearby locations will be referenced in the near future. By looking at how cache runs under the covers, you can see how it takes advantage of the locality effect.

SSP organizes the disk cache as a group of fixed-size chunks called

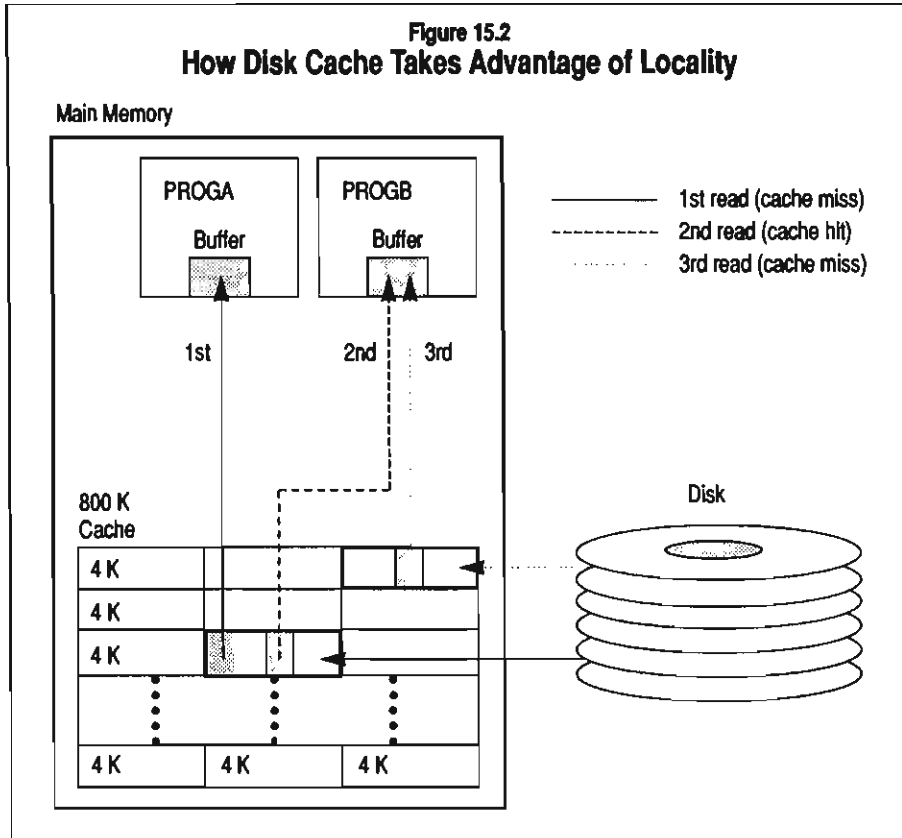
Figure 15.1
Disk Accesses with and without Cache



pages. A page can be as small as 1 K or as large as 16 K, but all pages within the cache have the same size. Figure 15.2 shows an 800 K disk cache having 4 K pages. In this example, two programs are running, PROGA and PROGB. When PROGA reads a 256-byte record from disk, SSP actually reads a whole page — 4 K — into the cache, and then copies the first 256 bytes from that page into PROGA. Because of the fast transfer rate of S/36 disks, reading 4 K of data requires only a little more time than reading 256 bytes. If PROGB then reads a different 256-byte record that happens to fall within the 4 K chunk previously stored in the cache, a cache hit occurs and SSP saves a disk I/O. If PROGB reads data not in the cache, a new 4 K page gets cached in addition to the 4 K already cached, increasing the probability that a future read can be satisfied by the cache. Each cached page represents a *locale* that stays in memory waiting for a hit. Unlike blocking, which contains only one physically contiguous chunk of data, the cache contains multiple locales, each holding data from a different disk location. Combined with the locality effect, multiple pages improve the probability of a cache hit. The example's 800 K cache holds 200 pages, each a separate locale.

Each cache miss results in a new locale being read into a cache page. SSP tracks the number of hits and misses for each page and how recently each page was accessed. Once all the pages are occupied, SSP decides via a least-recently-used algorithm which pages get overwritten to hold new locales. In this way, SSP ensures that pages only retain locales having a proven track record of hits vs. misses.

Figure 15.2
How Disk Cache Takes Advantage of Locality



The S/36 disk cache uses a technique called *write-through* to ensure that the cache always reflects the actual contents of disk. Whenever a program writes to disk, if the data written is partially or completely contained in the cache, SSP first updates the cache before performing the write operation. If other programs read data from the same locale before the write operation completes, they get the data from cache, avoiding a wait for the disk while still maintaining data consistency. If the data written to disk isn't contained in the cache, then nothing unusual happens — the data goes straight to disk without involving the cache.

Caching is fundamentally different from blocking. A block buffer holds a *single* locale with the intent that a single program will read all or most of the data in the buffer. Caching holds many locales with the intent that a few locales will be able to satisfy many read requests — possibly spread over several programs. Blocking improves both read and write performance, while cache improves performance only for disk reads. Popular myth holds that

blocking and cache are mutually exclusive performance tools — that you should not use blocking with cache enabled. In reality, though, the two complement each other — blocking can improve performance for individual applications while cache provides overall improvement. Systematic planning and careful measurement are the keys to optimizing performance with either tool.

Starting Out with Cache

No magic formula exists that tells you when cache will help and how to configure it to best advantage. The only sure path to success with cache is through experimentation. To conduct useful experiments that yield results you can use, you must know how to control cache, how to measure its performance, and how to evaluate those measurements for your environment.

You start, stop, and change cache with the CACHE procedure, which has the following syntax:

CACHE *command,size,pagesize*

where *command* is START to turn cache on, STOP to turn it off, or ALTER to change the cache configuration. The *size* parameter specifies how much real memory to set aside for the cache; *pagesize* sets the size of each page (from 1 K to 16 K). As a general rule, use 1/4 the size of installed memory as a starting point for cache size. For example, on a 2 MB system, you could start with a cache size of 512 K.

Choose a page size that results in at least 32 cache pages if your total installed memory is 2 MB or less. For larger memory sizes, the number of pages should be at least 1/64th of the memory size. On a 5 MB system, for instance, the cache should have at least 80 (5,120/64) pages; if you use 8 K pages, that means a cache of at least 640 K. You may end up using far more or far less memory for cache after you've done some performance tracking; until then, these guidelines give you a good starting point.

Before experimenting with cache, it's a good idea to obtain baseline SMF measurements as described in Chapter 13. You should record the value of User Area Disk Activity (UADA) under each set of working conditions for which you plan to use cache. For example, if you have a period during the day that is primarily interactive activity, make an SMF run to establish baseline values for that period. If later in the day you perform batch work alongside the interactive, consider that a separate period and establish a baseline for it, too. After you have collected these baseline measurements, you can enable cache to see whether performance improves.

You should activate cache before starting the programs you want to evaluate, then initiate the programs, and finally run SMFSTART to activate SMF. Follow the guidelines in Chapter 13 for setting SMF parameters. By starting SMF last, you avoid tainting your measurements with one-time initialization activity.

SMF delivers quite a lot of information that you don't need to evaluate cache performance, so you may want to use the CACHIQ utility, described later in this chapter, in place of SMF. CACHIQ provides a real-time display of cache performance as well as accumulated averages; it also stores its measurements in a disk file that you can save for later analysis.

Note that IBM's CACHE procedure cannot run via // EVOKE or // JOBQ — you must run it on the command side of the console workstation. The SETCACHE utility, described later, gets around this problem, making it much easier for you to automate cache start-up and tailoring. Note also that every time you change the cache configuration using CACHE ALTER, SSP completely empties and reconstructs the cache buffer. Changing the cache too frequently (more often than every few minutes) can degrade performance and pollute your measurements.

After you've completed a measurement period with cache, run SMFSTOP. Whether or not you stop cache or your applications is up to you. If performance seems perceptibly better, you might as well leave cache turned on until the next measurement period.

Counting Cache

Once you've collected cache statistics for one measurement period, you should analyze them before making further test runs. Run an SMFPRINT SUMMARY report; the items to examine appear under Summary System Event Counters (Figure 15.3): Disk Cache Hits, Disk Cache Misses, and User Area Disk Activity (UADA).

The primary measure of cache efficiency is the hits-to-misses ratio, which you must compute by dividing the Cache Hits by the sum of Cache Hits and Cache Misses. In the sample report, the hit/miss ratio for 755 hits to 250 misses is .7512, which translates to an efficiency value of 75.12 percent. An efficiency under 50 percent indicates that more time is being spent managing cache than is gained in increased performance. Anything above 50 percent is good — the higher the better.

However, simply achieving a high cache efficiency doesn't mean your overall performance is any better. When you activate cache, SSP takes memory away from the user area to accommodate the cache buffer. This could cause increased paging activity, as reflected in the UADA value. Your objective is to maximize cache efficiency by experimenting with various combinations of cache size and page size, *without* increasing UADA significantly. This is where your baseline UADA measurement is important: If the UADA count per minute increases by an amount greater than the cache hits per minute, then cache is costing you more than it's saving, even though you might be seeing high cache efficiency. Cache hits per minute represents the number of disk operations cache is saving; if the increase in UADA per minute exceeds this,

Figure 15.3

SMF Summary System Event Counters for Cache Evaluation

-------------- S U M M A R Y	S Y S T E M TOTAL	E V E N T PER MINUTE	C O U N T E R S MAXIMUM	-------------- TIME MAXIMUM OCCU
MAIN STORAGE TRANSIENT CALLS. .	63	3.1	23	12.02.19.358
TRANSLATED TRANSFER CALLS . . .	2969	146.8	569	12.02.19.358
ASYNCHRONOUS TRANSFER CALLS . .	71	3.5	14	12.02.19.358
MAIN STORAGE TRANSIENT LOADS. .	28	1.4	9	12.02.19.358
TRANSLATED TRANSFER LOADS . . .	60	3.0	22	12.02.19.358
MAIN STORAGE LOADER REQUESTS. .	29	1.4	9	12.02.19.358
•				
•				
•				
MEMORY RESIDENT OVERLAY LOADS .	0	0.0	0	00.00.00.000
MEMORY RESIDENT OVERLAY MAPS. .	0	0.0	0	00.00.00.000
DISK CACHE HITS.	755	66.0	84	12.02.19.358
DISK CACHE MISSES.	260	20.0	40	12.04.29.213
USER AREA DISK ACTIVITY	70	4.4	34	12.02.19.358
NOT USED.	0	0.0	0	00.00.00.000
NOT USED.	0	0.0	0	00.00.00.000
NOT USED.	0	0.0	0	00.00.00.000
NOT USED.	0	0.0	0	00.00.00.000
NOT USED.	0	0.0	0	00.00.00.000
NOT USED.	0	0.0	0	00.00.00.000

you're on sinking sand. Try reducing the size of the cache by 50 percent and rerunning your tests.

If you get low to medium cache efficiency (51 percent to 75 percent), and UADA has shown little or no increase, try enlarging the cache by 50 percent and rerunning your tests. The unchanging UADA indicates that the cache hasn't yet encroached on the working set of virtual pages. You also should try increasing the cache page size after making a test run with the larger cache. Low efficiency may be due to the locale not being large enough to accommodate the size of an access cluster on your system.

After each test run and evaluation, continue changing cache size or page size by 50 percent. That value isn't accidental: It takes you through the shortest number of tests to find an optimal value for the conditions you're testing. You're actually performing a binary search, cutting the difference by half on each iteration between your current settings and the optimal ones. Suppose you start out with a cache size of 1 MB on a 4 MB system. After the first pass, if performance decreases, you'll reduce the 1 MB cache size by 50 percent to 500 K; on the second pass, if performance increases, you'll increase the cache to 750 K (500 K plus 50 percent of 500 K); on the next pass, if performance decreases, you'll decrease the cache to 625 K (750 K minus 50 percent of 250 K); and on the last pass you may reduce the cache to 562 K (625 K minus 50 percent of 125 K). On that last pass you'll find that performance worsens and decide that 625 K is optimal. Once you've settled on an optimal cache size, follow the same process to determine page size.

If you're lucky enough to see high cache efficiency without increased UADA on your first run, congratulate yourself on being a shrewd judge of numbers. Then run some confirming tests to bracket your selections, varying both cache size and page size by 10 percent above and below the original settings.

Quick Cache

Running these tests using SMF quickly becomes tedious, because you need to name and manage files, generate reports, and perform efficiency calculations. To make the task easier, we created the CACHIQ utility, which displays the current and per-minute values for cache hits, cache misses, and UADA (CACHIQ is included on the "Desktop Guide" diskette). CACHIQ also computes cache efficiency using the hit/miss ratio. You don't need to run any reports or perform any calculations, and you see the results in real time. CACHIQ also logs the information to a summary file as a permanent record for later analysis, should you need that capability. The only caveat for using CACHIQ is that you can't run SMF at the same time, because CACHIQ reads and resets the same system event counters as SMF.

The syntax for the CACHIQ procedure is:

CACHIQ *snapinterval,snapcount,display,logfile*

where *snapinterval* is the time interval between snapshots in hhmmss format (the default is 000100, or one minute); *snapcount* is the number of snapshots to take (the default of 0 indicates that snapshots are taken until the operator terminates the utility); *display* is *NODISPLAY if you don't want to view results on the screen in real time, *DISPLAY if you do (the default is *DISPLAY); and *logfile* is the name of a file to which log records should be written (if *logfile* is blank, no log records are written).

If you run CACHIQ with no parameters, the utility displays cache statistics in real time at one-minute intervals until you terminate it by taking SysReq option number 4 (set inquiry latch). Figure 15.4 shows how CACHIQ's display looks. Note that the current snapshot interval must expire before CACHIQ "sees" the termination request and quits. Ending CACHIQ with SysReq option 3 (cancel job) is no problem as long as you aren't logging snapshots. If you don't want CACHIQ to automatically display at fixed intervals, specify 0 (zero) for *snapinterval*; CACHIQ will update its display each time you press ENTER.

The parameters on CACHIQ let you use it in a batch environment to collect statistics without tying up a display station; for example, by using // EVOKE to start CACHIQ as an independent job. For instance, you could run:

```
// EVOKE CACHIQ 000500,144,*NODISPLAY,IQLOG
```

to create an independent task that runs CACHIQ for 12 hours (144 five-minute snapshots) and writes each snapshot to a file named IQLOG. Figure 15.5

Figure 15.4
CACHIQ Display

```
                CACHE IQ                              Logfile: IQFILE
          Cache Status Monitor

     Cache active:     Y
       SMF active:     N
 Total cache size:   300K
  Cache page size:     4K     Per Minute
       Cache hits:  1540       620.4
     Cache misses:   411        52.7
       Efficiency:    78.9%     92.1%

             UADA:    19        16.4

Snapshot interval:     1:00
 Current snapshot: 22:11:58        Count: 129
       Start time: 20:02:53           Of: 500
     Elapsed time:  2:09:05    Remaining: 371

Press ATTN and take option 4 to interrupt
```

describes the logfile record format. To stop CACHIQ under these conditions, enter a STOP SYSTEM command, which CACHIQ will recognize and obey after the current snapshot interval expires. Alternatively, you can use the CANCEL console command to kill the CACHIQ job (but use option 2 if you want to keep the log file).

In addition to the log file, CACHIQ always outputs the values of its last snapshot into the *system* LDA, in the same format as the logfile record, starting in system LDA position 1. Thus, incorporating the following into your cache control procedure:

```
CACHIQ 000100,1,*NODISPLAY
```

causes CACHIQ to wait for one minute, then take a snapshot and quit, leaving the snapshot results in the system LDA. To access the system LDA via OCL, insert // LOCAL AREA-SYSTEM into your procedure; all LDA references after this statement will be to the system LDA. Your procedure or a program you write could evaluate the snapshot results and then compute new (presumably better) cache parameters.

When using CACHIQ interactively, keep in mind that its per-minute rates are computed over the time period that CACHIQ runs. To reset CACHIQ's per-minute counters, simply end and restart it. CACHIQ is much easier to use than SMF when you only need to examine cache management system event counters.

Figure 15.5

CACHIQ Logfile Record Format

Record length is 64 bytes:

Start Pos	End Pos	Dec Pos	Description
1	1		Cache active (Y or N)
2	2		SMF active (Y or N)
3	6	0	Cache size (K)
7	8	0	Cache page size (K)
9	13	0	Cache hits
14	18	1	Cache hits per min
19	23	0	Cache misses
24	28	1	Cache misses per min
29	33	1	Cache efficiency (%)
34	38	1	Cache efficiency per min (%)
39	43	0	UADA
44	48	1	UADA per minute
49	54	0	Snapshot interval
55	60	0	Last snapshot time
61	66	0	Start time
67	72	0	Elapsed time
73	77	0	Number of snapshots taken

Dynamically Controlling Cache

You probably will find that a single cache configuration doesn't meet the needs of all your working environments. One set of cache values might work well for morning processing, but running a few parallel batch jobs over lunch (with few active interactive users) might require a much larger cache. A third cache configuration might work well for afternoon mixed interactive and batch processing, and yet a fourth for evening processing.

IBM's requirement that the CACHE procedure be run interactively from the console means you must have somebody manually enter the commands for each change. We've developed a handy utility that gets around this problem nicely. SETCACHE (provided on diskette) is a procedure and program that accepts the same parameters as IBM's CACHE procedure, but that can be run in batch mode via // JOBQ or // EVOKE. Using SETCACHE you can create a cache-control procedure like the one shown in Figure 15.6 that automatically changes the cache at fixed times of day.

If you're more ambitious, you might devise an algorithm to automatically compute new cache configuration values to adapt to a changing system workload. You could use CACHIQ to take a cache performance snapshot and then invoke a program to read the snapshot values from the LDA and make a cache change determination based on cache efficiency and UADA trends. Figure 15.7 illustrates such a procedure, although the program XCACHE, which

Figure 15.6

Sample Procedure to Automatically Change Cache Configuration on a Predetermined Schedule

```
*
* Automatice Cache Scheduler
*
SETCACHE START,4,300
// WAIT TIME-120000   At noon, change to big cache for multibatch
SETCACHE ALTER,8,1024
// WAIT TIME-130000   At 1:00pm, change to medium cache for mixed jobs
SETCACHE ALTER,2,600
// WAIT TIME-183000   At 6:30pm, change to huge cache for single batch
SETCACHE ALTER,16,2048
// WAIT TIME-235959   At midnight, turn cache off for backup
SETCACHE STOP
```

Figure 15.7

Sample Procedure to Automatically Change Cache in Response to Changing Environment

```
*
* Automatic Cache Controller
*
// LOCAL AREA-SYSTEM
// LOCAL OFFSET-1,BLANK-256
*
* Repeat this loop indefinitely
*
// TAG LOOP
*
* Get a cache performance snapshot into the LDA every five minutes
*
CACHIQ 000500,1,*NODISPLAY
*
* Run XCACHE to evaluate the snapshot from the LDA
*   (LDA also used to hold trend data)
*
// LOAD XCACHE
// RUN
*
* If XCACHE decided to change the cache config, carry out the change
*
// IF ?L'201,5'/ALTER SETCACHE ALTER ?L'211,4'?,?L'215,2'?
*
* Repeat
*
// GOTO LOOP
```

makes the cache configuration decision, is left as an exercise for the reader. (Such a program must take into account your specific operating conditions; hence, a general-purpose solution isn't recommended.)

Value-Added Cache

In the fall of 1992, IBM released its Value Added Software Package (VASP) for the S/36. VASP is free to any S/36 user holding a legal SSP license. VASP adds a fourth parameter to the CACHE procedure (which SETCACHE also supports) to enable or disable caching for disk scan operations. Specifying SCAN enables caching of scans; NOSCAN (the default) disables them. SSP performs disk scans when searching an index for an indexed random read (e.g., an RPG CHAIN operation). Before VASP, cache would always cache scan operations that read more than a single sector of disk. In practice, this turned out to increase the number of cache misses, because scans seldom started at the same sector in an index. With VASP's CACHE modifications, scans are by default no longer cached. If you're having trouble getting acceptable cache efficiency, your problem could be scan-induced cache misses. Installing VASP will change the cache to not include scan operations, and it will enable the SCAN/NOSCAN parameter on both the CACHE and SETCACHE procedures.

To Cache or Not to Cache

While the most certain method for finding out if cache will help system performance is to try it out, there are some truisms to consider when selecting work environments for each measurement period. Generally, cache is most effective on systems having underutilized memory, many ***data file*** disk accesses, a high read-to-write ratio, and shared files with a moderate amount of sequential access. Given those requirements, you might think that a single program running on a dedicated system accessing a few files randomly couldn't take advantage of cache. Even in this situation, though, cache can bring huge performance benefits if the application exhibits locality with a large percentage of its disk accesses. For example, if the program randomly retrieved a small number of frequently used inventory records widely scattered in a master file, cache would capture the most frequently accessed records and eliminate disk I/O whenever the program needed them. Business programs often exhibit this kind of behavior; but experimentation is important because it's not always easy to identify such programs by inspection.

When you first use cache, you might be discouraged by seeing high cache efficiency but correspondingly high increases in UADA, negating the value of cache. All is not lost, however — this is an indication that cache would be helpful if you could add more memory. Memory is one of the cheapest commodities in S/36 upgrading; you should seriously consider adding as much memory as your model CPU can accommodate. If cache still shows high efficiency with unacceptable UADA, and you're running on a smaller S/36 model, consider upgrading to a 5360 D model, which supports up to 8 MB of memory (see Chapter 5, "The Importance of Memory and Disk Space," for information on adding extra memory).

A few situations warrant turning cache off. One of these is a dedicated batch program performing purely sequential processing. Such a program never will take advantage of cache, because it never processes the same record twice. In fact, any number of concurrently running, sequential batch jobs that don't share files won't be helped by cache. Cache will, in fact, greatly degrade performance due to the overhead of constantly refilling cache pages. If the programs update records, the degradation will be even worse!

Cache also is not useful when you are performing backup operations or device-to-device copies (which usually take place on a dedicated machine). Cache slows these operations because the files can't be shared and won't exhibit any locality. An exception to this scenario is when the system is reorganizing a file that is extremely out of key sequence; in this case, cache can help if clusters of ascending keys exist throughout the file. Without cache, the $COPY utility must physically read each record randomly by key as the keys appear in the index ($COPY does not use blocking when performing a reorg). With cache, however, each physical read operation brings in the locale immediately following the record. If that record is the start of a cluster of records containing ascending keys, subsequent physical reads will be satisfied from the cache, reducing disk I/O. If you run dedicated reorganizes on large files (e.g., during the night) and find the reorgs don't finish in the allotted time, try using a large cache with the page size set to contain as many records as typically occur in a cluster of ascending keys.

No matter what your job mix, chances are that cache — given adequate memory — can improve performance for certain processing scenarios. Your job is to identify those scenarios, perform systematic cache experiments to determine which cache configurations give the best performance advantage, and then automate management of your cache to use the right cache configuration at the right time. Don't make the mistake of ignoring cache, however, or of trying to use cache on a memory-constrained system. Spend a little money to maximize your memory, then let cache take a load off your system.

Chapter 16

Is Response Time Fast Enough?

Users tend to have a narrow view of system performance. While you may be tickled pink to achieve low UADA counts, improved memory utilization, and balanced disk drives, users care about one thing only: response time. There are good reasons for user concern over response time. First, of course, is the frustration of waiting for the computer when trying to get work done. But more importantly, human interface studies have shown that when a computer takes too long to respond to user actions, the users in turn take longer to respond when the computer finally does react. For character-based computer dialogs — the category into which S/36 interactive programs fall — when response time is longer than a second or so, user attention begins to wander. The end result is a steeply worsening user productivity curve.

Subsecond response time thus is a goal worth pursuing. Although achieving subsecond response time might seem hopelessly impossible, there is much you can do to achieve this goal. This chapter will show you the ropes, explaining how to measure response time, what application design pitfalls to avoid, and how to harness external program calls (EPC) to dramatically cut processing time.

Response-Time Speedometer

As Chapter 13's discussion on SMF points out, any kind of performance tuning requires careful measurements before, during, and after the tuning process. Because the objective of performance tuning is to make the machine faster, one of the most important measurements is the response time of interactive applications. Unfortunately, this is one measurement that the stock S/36 doesn't provide automatically. Sure, you can use a stopwatch and clipboard to log response time measurements, but besides being just plain inconvenient, this manual technique isn't very accurate. Further, simultaneously recording the response times of many workstations requires an army of observers.

IBM does offer a product for sale — the Response Time Measurement Facility (RMF) — that collects and reports S/36 response time values automatically. But the utility is expensive and difficult to use. It turns out, though, that IBM already supplies the basic tools necessary to measure workstation response times. Utility RTIMER, supplied on the diskette with this book, takes advantage of those built-in tools to collect, analyze, and report response-time performance figures. Armed with these objective measurements, you can directly evaluate the effectiveness of your system tuning efforts. Even better, RTIMER lets you give users concrete proof when your tuning efforts pay off.

Performance Tip

User complaints about response-time degradation are often based on subjective, not objective, measurements. When users complain about poor response times, use RTIMER to get the real story.

RTIMER takes advantage of an undocumented feature built into SSP to cause it to collect response times and log them to disk, similar to the way SMF collects various performance values and logs them to disk. SSP measures response time as the interval from when the user presses ENTER (or some other function key) to when the program next outputs a display screen. RTIMER's reporting component — analogous to the SMFPRINT component of SMF — then analyzes the logged measurements and prints a report.

Using RTIMER is a four-step operation. First, you start up response-time collection via the RTIMER procedure:

RTIMER START,*blocks*

where *blocks* is the number of blocks to allocate for the data collection file. Each block holds about 80 response-time entries. If you omit this parameter, RTIMER uses a default of 10. Next, you run the mix of interactive applications you're interested in evaluating. Be sure to process enough transactions to accumulate a significant number of measurements (several dozen for each screen format you're evaluating), and then stop response-time collection:

```
RTIMER STOP
```

Finally, you run procedure RTIMEP to analyze the collected statistics and print a report. The report details every application-related workstation event (Figure 16.1) unless you specify NODETAIL in procedure parameter one, which results in a summary-line-only report (Figure 16.2). The detail report shows the clock time of each workstation transaction (input followed by output) and the response time. The summary report shows only total transaction counts and response-time averages. Both reports are sorted by program name, screen format name, and workstation ID, and show response-time averages by workstation and screen format. Both reports also show a final, overall response-time average. RTIMEP sorts first by program name in case some formats are shared among several programs or in case screen format names are not unique among programs. Sorting by screen format name and workstation ID lets you spot, and possibly factor out, performance differences among workstations. For example, workstations accessing remotely stored files via DDM will have long response times due to delays accessing records over a communications network. Another point to consider when reading the report is that format names identify the format displayed *at the start* of each workstation operation. Thus, program initiation — the transition from menu to first screen — doesn't falsely bias averages for any given screen format. This also means, however, that you must take care when evaluating response times if your application design methodology results in frequent program initiations. RTIMER may not show these program initiation delays; be assured, however, that your users experience them nonetheless!

Figure 16.1
Sample Detail Report from Utility RTIMER

2/20/91 Response Time Analysis Page 2

Prog	Format	WS	Job Name	Time	R-time	WS Cnt	WS Avg	FM Cnt	FM Avg
MEL	MTMT79	W1	1910213	17:50:26	11.69	1	11.69	1	11.69
MEL	QVUB02	W2	2910213	17:50:32	1.11	1	1.11	1	1.11
QVUBRS	QVUBX080	W2	W2172508	17:53:10	4.94	1	4.94	1	4.94
QVUBRS	QVUBX132	W1	W1173149	17:55:49	7.83				
QVUBRS	QVUBX132	W1	W1173149	17:56:21	2.58				
QVUBRS	QVUBX132	W1	W1173149	17:56:57	.45	3	3.62		
QVUBRS	QVUBX132	W2	W2172508	17:50:47	.52				
QVUBRS	QVUBX132	W2	W2172508	17:53:12	.06	2	.29	5	2.28
QVUBRS	QVUB02	W1	W1173149	17:56:04	1.50				
QVUBRS	QVUB02	W1	W1173149	17:56:07	.95				
QVUBRS	QVUB02	W1	W1173149	17:56:11	2.03				
QVUBRS	QVUB02	W1	W1173149	17:56:14	3.47	4	1.98		
QVUBRS	QVUB02	W2	W2172508	17:50:34	.72				
QVUBRS	QVUB02	W2	W2172508	17:52:13	.58				
QVUBRS	QVUB02	W2	W2172508	17:52:14	.52				
QVUBRS	QVUB02	W2	W2172508	17:52:35	.62				
QVUBRS	QVUB02	W2	W2172508	17:53:13	.52				
QVUBRS	QVUB02	W2	W2172508	17:53:15	.55				
QVUBRS	QVUB02	W2	W2172508	17:53:28	.52				
QVUBRS	QVUB02	W2	W2172508	17:53:28	.52				
QVUBRS	QVUB02	W2	W2172508	17:53:29	.55				
QVUBRS	QVUB02	W2	W2172508	17:53:29	.55				
QVUBRS	QVUB02	W2	W2172508	17:53:32	.52				
QVUBRS	QVUB02	W2	W2172508	17:53:32	.52				
QVUBRS	QVUB02	W2	W2172508	17:53:34	.52				
QVUBRS	QVUB02	W2	W2172508	17:53:34	.52	14	.55	18	.87
QVUBRS	QVUB03	W1	W1173149	17:56:53	1.21				
QVUBRS	QVUB03	W1	W1173149	17:56:56	.65				
QVUBRS	QVUB03	W1	W1173149	17:57:00	1.04	3	.96		
QVUBRS	QVUB03	W2	W2172508	17:50:42	1.14				
QVUBRS	QVUB03	W2	W2172508	17:50:45	.65				
QVUBRS	QVUB03	W2	W2172508	17:54:59	1.08				
QVUBRS	QVUB03	W2	W2172508	17:54:59	1.08	4	.98	7	.97
						Overall Average:		73	1.94

Looking for Delay

With an objective tool for measuring response time, you're ready to look for application problems that could be causing slow response times. There are two major areas to examine: application design and program implementation.

Keep in mind that one of your goals in achieving subsecond response time is to avoid having users become inattentive between transactions. If each press of the Enter key currently sends the computer off on a five- or 10-second processing mission, you might consider breaking the transaction into

Figure 16.2
Sample Summary Report from Utility RTIMER

2/20/91 Response Time Analysis Page 1

Prog	Format	WS	Job Name	Time	R-time	WS Cnt	WS Avg	FM Cnt	FM Avg
FSED2	COPY	W1				1	.72	1	.72
FSED2	MTMTLC79	W2				2	.99	2	.99
FSED2	MTMT79	W1				18	1.16		
FSED2	MTMT79	W2				9	2.28	27	1.53
FSED3	MTMT79	W1				3	.94	3	.94
FSED4	SOPTIONS	W1				2	3.57	2	3.57
MEL	MTMT79	W1				1	11.69	1	11.69
MEL	QVUB02	W2				1	1.11	1	1.11
QVUBRS	QVUBX080	W2				1	4.94	1	4.94
QVUBRS	QVUBX132	W1				3	3.62		
QVUBRS	QVUBX132	W2				2	.29	5	2.28

smaller steps to reduce the elapsed time between system responses. There are no medals given in the Programmer's Hall of Fame for cramming the most data into a single screen. Spreading the details of a transaction across several screens has the added advantage of reducing the amount of information users must digest at one time.

If you break a transaction at natural pause points for the user, you'll not only achieve faster response times, you'll take better advantage of the "think" time during which the computer is normally idle. For example, a user transcribing data from a multipage input document has natural pause points when turning pages; your application should take advantage of those pauses by making the data entry for each page a separate screen. As a more concrete example, consider a telephone-driven order-entry process, with phone operators interactively developing orders based on conversation with customers. The operator/customer dialog has natural pauses (e.g., when the operator must ask the customer for the next item) during which the system could be processing previous input. If the order-entry application forces the operator to key an entire order on one screen before pressing Enter, the resulting delay to close out the order could be quite long. During this time, conversation usually turns to the incompetence of the data processing technical staff. If, on the other hand, the operator had been pressing Enter after each order item, by the time the order is finished little additional processing remains, letting the operator give the order total to the customer and end the transaction.

This kind of application design requires careful attention to exception conditions. The user must, for example, be able to back up to a previous step in the input process at any time — forcing the user to start over is bad user-interface technique. You also should strive to provide continuity from screen to screen, carrying some information over from previous screens. In the order-entry example, you should display previously entered items and let the user make changes as required. Infrequently used data items shouldn't clutter up the screen; instead, let the user call up exception screens to enter or access these items. Consider using pop-up overlays for such screens; such overlays let the user maintain a point of reference during the exception procedure.

Figure 16.3 shows an all-in-one order-entry screen in which the user enters an entire order before sending it to the system. The screen is complex and requires complex navigation. It displays every conceivable bit of data associated with an order: customer name, bill-to and ship-to addresses, credit status, special instructions, order history, discount rates, tax and payment terms, line items, and notes for each line item. Figures 16.4a through 16.4e show one way this single screen could be divided into several interactive steps. In step (a), the user deals only with customer name and address information. If the operator must check the client's credit history, pressing a command key calls up the appropriate information in a pop-up overlay (step b), which can take advantage of additional screen space to present data more clearly. In step (c), the operator enters the ship-to address if it differs from the bill-to address; the operator can use a similar exception procedure to enter special instructions or review discount rates. When entering line items (d), only basic customer identifiers are carried over from the previous screen; the operator usually does not need to refer to address or other information during item entry, so that data can safely be dropped. Note, however, that the operator can quickly back up to the previous screen via a standard command key. During item entry, the operator presses ENTER after each item; prior items appear above and the input area moves down the screen as each item is processed. Item notes only occupy a line if the operator enters them, so most items take only one line — not two lines as in the original screen — further reducing screen clutter. Data about each item not normally required during order entry (e.g., stock date) is eliminated; the operator can call it up via a command key if necessary. After ending the order, an overlay screen (e) presents summary information the operator needs to confirm the order with the customer.

The revised application design performs less processing with each step, letting the operator get feedback more frequently and keeping response times down. The new design is also easier to use, and easier to modify when the time comes for those inevitable enhancements. This example should give you some ideas for improving the interfaces of your own applications as a means to achieving subsecond response time.

Figure 16.3

Typical All-In-One Order Entry Panel

```
  Cust#: 015788                           Ship-to: Clark's Printing
   Name: Clark's Printing                          Attn: Journeyman
Bill-to: 101 Gregory Drive                         400 Heidelberg Road
         Ventura, CA 93003                         Ventura, CA 93003
  Phone: 805 647-3125                      Credit: OK    Lims: 1003/1500
                                           Rating: G     AvPy: 30
Sp inst: ____________________________   Discounts: 5.5%    7.1%    11.0%
         ____________________________         Tax: 6.25%
  Terms: Net 30                        Last order: 10/15/92 Amt: $  550
Bal due:    300.00    Age: 30 days      YTD sales: $  2250

 Item#  Description           Units  Qty  QOH  B/O  StockDt   Unit Pr     Total
 14039  Engraving, plate       Ea     10  878    0  06/15/92    33.40    334.00
  notes: ____________________
 14042  Cards, detailed        Ea    890 9250    0  06/19/92     1.00    890.00
  notes: ____________________
 14044  Stock ink green        Drum    1  550    0  05/01/92   423.90    423.90
  notes: ____________________
 14052  Rack mounts 3"         Ea    100 9132    0  07/03/92     3.99    399.00
  notes: Don't use glass_____
 14054  Stock ink red          Drum    1    0    1  03/30/92   480.22    480.22
  notes: ____________________

 Press enter when finished.  Roll keys to scroll   Ck7: End job   Ck9: Cancel
```

Figure 16.4a

Customer Name and Address Information Screen

```
  Cust#: 015788
   Name: Clark's Printing
Bill-to: 101 Gregory Drive
         Ventura, CA 93003

  Phone: 805 647-3125









 Ck3: Back        Ck4: Ship-to        Ck5: Credit      Ck6: Discounts
 Ck7: End job     Ck8: Special Inst   Ck9: Cancel
```

Figure 16.4b
Customer Credit History Screen

```
  Cust#: 015788                      ----------Credit Information-----------
   Name: Clark's Printing            Credit allowed . . . . . . . . Yes
Bill-to: 101 Gregory Drive           Max credit line  . . . . . . . $ 1500
         Ventura, CA 93003           Available credit line  . . . . $ 1003
                                     Rating . . . . . . . . . . . . G (good)
  Phone: 805 647-3125                Average payment in . . . . . . 30 days

                                     -----------Financial History-----------
                                     Balance due  . . . . . . . . . $  300
                                                                    30 days
                                     Last order . . . . . . . . . . $  550
                                                                    10/15/92
                                     YTD sales  . . . . . . . . . . $ 2250

 Ck3: Back        Ck4: Ship-to       Ck5: Credit      Ck6: Discounts
 Ck7: End job     Ck8: Special Inst  Ck9: Cancel
```

Figure 16.4c
Ship-To Address Screen

```
  Cust#: 015788
   Name: Clark's Printing
Bill-to: 101 Gregory Drive
         Ventura, CA 93003

  Phone: 805 647-3125

Ship-to: Clark's Printing
         Attn: Journeyman
         400 Heidelberg Road
         Ventura, CA 93003

 Ck3: Back        Ck4: Ship-to       Ck5: Credit      Ck6: Discounts
 Ck7: End job     Ck8: Special Inst  Ck9: Cancel
```

Figure 16.4d

Customer Line-Item Order Screen

```
 Cust#: 015788 Clark's Printing

Item#  Description             Units   Qty   QOH  B/O  Unit Price     Total
14039  Engraving, plate         Ea      10   878   0     33.40       334.00
14042  Cards, detailed          Ea     890  9250   0      1.00       890.00
14044  Stock ink green          Drum     1   550   0    423.90       423.90
14052  Rack mounts 3"           Ea     100  9132   0      3.99       399.00
 notes: don't use glass
14054  Stock ink red            Drum     1     0   1    480.22       480.22

_____  ____________________    ______ ____
 notes: ____________________

 Ck3: Back           Ck7: End job       Ck8: Item Detail      Ck9: Cancel
Roll: Scroll items  Ck10: End order
```

Figure 16.4e

Summary Order Information Screen

```
 Cust#: 015788 Clark's Printing

Item#  Description             Units   Qty   QOH  B/O  Unit Price     Total
14039  Engraving, plate         Ea      10   878   0     33.40       334.00
14042  Cards, detailed          Ea     890  9250   0      1.00       890.00
14044  Stock ink gr******************************************        423.90
14052  Rack mounts *           Order Completed              *        399.00
 notes: don't use g*                                        *
14054  Stock ink re*    Subtotal . . . . . . . $ 2527.12    *        480.22
                   *    Tax ( 6.25%) . . . . . .  157.95    *
                   *    Shipping . . . . . . . .   55.00    *
                   *    Total due  . . . . . . . 2740.07    *
                   *                                        *
                   ******************************************

 Ck3: Back        Ck7: End job        Ck9: Cancel       Ck10: Accept order
```

Getting SSP Out of the Loop

Modifying program design to reduce the amount of application work done in each interactive step is one way to improve response time. Another way is by reducing the amount of *system* work performed when linking between programs. Most S/36 applications consist of several programs, with control being passed from program to program depending on user actions. For example, when a user needs an alphabetic search function from within an order-entry program, the order-entry program passes control to a separate interactive alpha-search program, along with the user's search argument. When the alpha-search program finishes, it passes control back to the order-entry program, along with the results of the search.

Traditional S/36 application design uses OCL statements to pass control between programs and the Local Data Area (LDA) to exchange data arguments. Figure 16.5 illustrates the process. OCL statements within a procedure load and run program MAIN, which opens the files it uses and begins processing. Later, when MAIN needs to invoke program ALPHA, it stores the search argument in the LDA and terminates, closing its files and returning control to the procedure. The procedure then loads and runs program ALPHA, which opens its own files and uses the argument passed in the LDA to carry out the search. When the search is completed, ALPHA stores the search result in the LDA and terminates in turn, closing files and returning control to the procedure, which then can reload program MAIN (and reopen files), which executes logic to pick up the returned search result from the LDA and resume control at the previous point.

Whew! The traditional OCL-based program linkage clearly is convoluted. Worse, though, it's expensive in terms of response time; because OCL processing, program initiation, termination, and LDA access all result in time-consuming disk accesses. Figure 16.6 shows the number of disk operations required for each step in the process of invoking a program using this technique. The total number of operations depends upon the number of files used in each application, but even the best-case scenario, with one file in each program, results in 17 disk accesses — more than a half-second of overhead. A more realistic situation, with a half-dozen files in each program, consumes nearly 50 disk accesses and a second and a half of overhead. Even if the programs themselves run in zero time (not likely), subsecond response time is obviously out of the question.

S/36 programmers, though, are an ingenious lot; over the years they've cooked up a number of clever techniques — read-under-format workstation I/O, Multiple Requester Terminals (MRTs), and Never-Ending Programs (NEPs) to name a few — to reduce this overhead. Unfortunately, while these improvements do reduce the invocation overhead, they make coding even more convoluted than before.

Figure 16.5
Traditional Program Linkage Using OCL and LDA

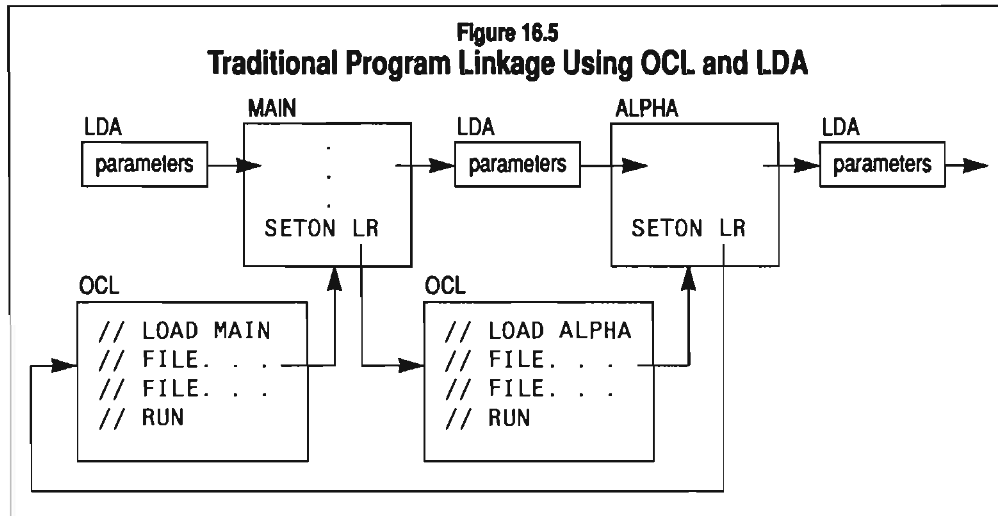


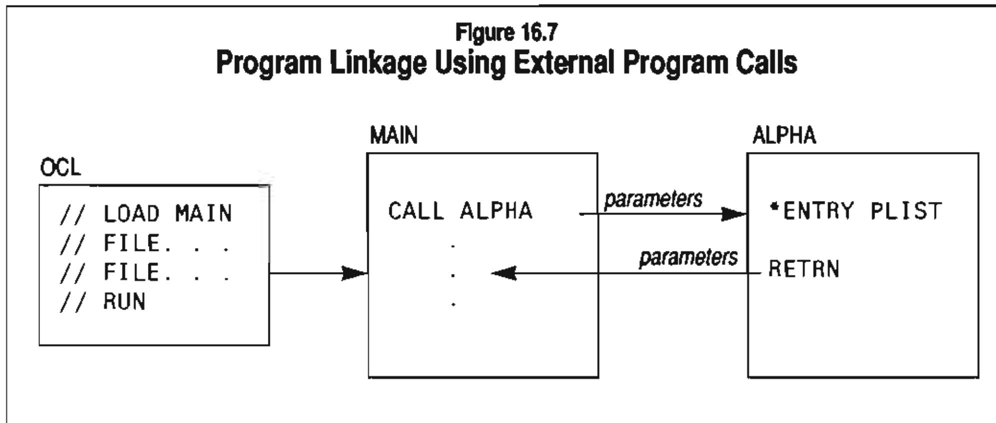
Figure 16.6
Disk Overhead for Terminating/Initiating Task Sequence

Description	Number of Disk Ops
Terminate existing task	
Close files	2 per file
Save LDA	1
Release swap area	2
Initiate new task	
Read OCL statements	2
Locate new program in library directory	2
Create swap area	1
Load new program	1
Restore LDA	1
Allocate files	1 per file
Open files	2 per file
Locate screen format member	2

A much better solution is the use of external program calls (EPCs) to invoke programs. EPC improves invocation performance in three ways. First, it eliminates OCL processing (except for the main program); second, it passes parameters privately via memory-to-memory transfers, bypassing the disk-based LDA; third, it lets multiple programs remain active within a job step, greatly reducing the need to constantly initiate and terminate programs.

Figure 16.7 illustrates the process for the previous example. Program

Figure 16.7
Program Linkage Using External Program Calls



MAIN is still loaded via OCL statements, and still requires the overhead of opening files in preparation for processing. However, when MAIN invokes program ALPHA, it does so via an RPG CALL statement, which loads ALPHA without first closing terminating main, and without processing any OCL statements. The overhead of closing files and OCL termination is saved. ALPHA begins execution by opening its files, as before, but receives the search argument without accessing the disk-based LDA, saving that I/O. When ALPHA finishes its work, it returns control to MAIN via an RPG RETRN statement, passing the search result back as a memory-based parameter. Program ALPHA doesn't terminate, though, but simply enters a suspended state while MAIN resumes execution where it left off, without being reloaded and without reopening its files.

The story isn't over yet, though. The first call to ALPHA using EPCs required only a few disk I/Os where the OCL-based method required many. But subsequent EPC calls to ALPHA will require no disk accesses at all! When MAIN calls ALPHA after the first time, ALPHA simply wakes up from its suspended state; when ALPHA returns, it "sleeps" again while MAIN wakes up. No OCL interpretation, program initiation or termination, or file opens and closes need be done. If neither program is paged out to disk, the invocation can be as fast as 4 milliseconds — a fraction of the time of a 35-millisecond disk access.

EPC clearly is the path to subsecond response time. For the full story on understanding and using EPC, turn to the chapters in Section III (External Program Calls). You'll have to expend some effort redesigning and reworking your existing applications, but the effort will pay off handsomely in improved response time and better user productivity.

Performance Tip

For even better use of EPC to eliminate program initiations and terminations, consider replacing menus with programs that display the menu screens and then use CALL to invoke requested applications. Then, when the user exits an application, use RETRN to return control to the menu program. This way, applications will remain active and appear with zero delay when the user next requests them from the menu program.

Section VI

Advanced Topics

"For every problem there is one solution which is simple, neat, and wrong."
—H.L. Mencken

IBM is always touting the improved programmer productivity possible with its current midrange contender, the AS/400. And indeed, compared with the S/36 as shipped by IBM, the AS/400 is a markedly superior machine. The AS/400 provides advanced file operations, better system access from high-level languages such as RPG, and source-level debugging and performance tools, to name just a few of its niftier features.

Fortunately, you don't have to settle for a machine the way IBM shrink-wraps it. A whole host of free-use and commercial add-ons bring many of the AS/400's advantages to the lowly S/36. Commercial add-ons, such as third-party RPG compilers and EPC products, are discussed elsewhere in this book. This section focuses on things you can use immediately, without spending any more money or getting anybody's permission. All of the enhancements described are contained in their entirety on the diskette accompanying this book.

Chapter 17 presents more than two dozen powerful assembler language routines that provide string handling, file manipulation, and system-level access similar to that available on the AS/400. This chapter also gives you the straight scoop on how these routines affect the future migratability of your programs. Chapter 18 shows you how two tools — a dump debugger and a code profiler — can help you perform source-level program debugging and tuning. Chapter 19 is a handbook for making your applications more portable to future platforms; topics include externally described files, careful RPG coding practices, sensible file design.

If you've got a tough coding problem and think the AS/400 is your only way of escape, check out this section. The S/36 might surprise you.

- System management
- Device control

It's important to keep in mind the following rules and conventions when using these or any other assembler subroutines:

- All the subroutines in this chapter have been written to be called from RPG. However, unless otherwise specified in the "Notes" section, COBOL programs may also call these subroutines by using a "bridge" routine, also included, called RBRIDG (see "RBRIDG," the last section in this chapter.)
- When calling an assembler subroutine, you must specify all the RLABLs defined in the calling sequence, whether or not your program needs them for a particular function. The reason for this is that an RLABL does not generate executable instructions, but rather a length and offset to the corresponding parameter within the program. When the subroutine returns control to your program, it must "jump around" the parameters, and there is no way for the subroutine to dynamically tell how many parameters exist.
- All RLABLs should be field names unless the parameter definition explicitly specifies that a given RLABL should be an array or Data Structure (DS). In the case of a DS, be sure the name you specify is the name in positions 7-12 of the I-spec line that defines the DS, and not just a subfield that redefines the entire DS.
- RLABLs can be named anything you want. They don't have to match the names in the calling sequence, because they are referenced by address, not by name.
- A question mark (?) in the length area of an RLABL definition denotes that the field can be a variable length, depending on the operation to be performed. If a question mark is not present, the RLABL must be the length specified in the calling sequence.
- Many of these subroutines require that the MSP be in "privileged mode" when they are called. All subroutines that require privileged mode have an attribute set in their library directory entry that ensures privileged mode is on when your program is run. This attribute should not be modified, or a task dump will occur when the subroutine is called.

STRING HANDLING FUNCTIONS

Of all the subroutines in this chapter, the seven that follow stand the best chance of migrating to the AS/400, RS/6000, or any other computer, for that matter. This is because none of them have any machine dependencies. Every computer language supports some kind of string handling capabilities. Even RPG/400 now supports a wide range of string operations such as CAT, SCAN, and SUBST.

These string-handling subroutines can enhance the performance of

any RPG program by virtually eliminating the need for variable index array processing, which is the most common, and most likely the only, means of getting the job done in RPG. Because variable index array processing often involves hundreds of machine cycles, any kind of searching or modification of an array causes unnecessary seconds, or even minutes, to be wasted. By using an assembler subroutine, you can usually eliminate the arrays entirely and have the subroutine work directly on the field you want to manipulate or search, thereby dramatically increasing performance.

SUBR$F

Description

Locates an arbitrary substring within a target string.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                    EXIT SUBR$F                      Used for
     C                    RLABL          OP       1        Input
     C                    RLABL          RESULT  30        Output
     C                    RLABL          ARGMNT   ?        Input
     C                    RLABL          TARGET   ?        Input
     C                    RLABL          LEFTP   30        Input
     C                    RLABL          RIGHTP  30        Input
```

Parameter definition

OP — A 1-byte field that contains the operation to perform.
I = Initial search
R = Repeat search

RESULT — A 3-byte numeric field that will contain the leftmost position of the search string in the target field if a match was found, 0 if the string was not found, and -1 if you made a coding error in the search parameters (e.g., ARGMNT larger than TARGET; LEFTP greater than RIGHTP).

ARGMNT — A field up to 256 bytes long that contains the search argument. The argument ends with the first blank character unless you enclose the actual search argument in single quotation marks. If you enclose the argument in double quotation marks, both upper and lower case characters in the target string will match (i.e., case is ignored).

TARGET — A field up to 256 bytes long that contains the target string to search.

LEFTP — A 3-byte numeric field that specifies the leftmost position in TARGET to be considered in the search. If LEFTP is zero, the value 1 is assumed.

RIGHTP A 3-byte numeric field that specifies the rightmost position in TARGET to be considered in the search. If RIGHTP is zero, the search string must start at position LEFTP in the target string to match the argument; this is called an "anchored" search and is much faster than a general search, because only one compare needs to be performed rather than a test of all possible positions.

Notes

- An initial search is used every time you want to change the search arguments. A repeat search is used to repeat a search using the same argument you used previously but with different data in the target field. A repeat search is much faster than an initial search because all the initialization code in SUBR$F is not executed.

SUBR$C

Description

Concatenate a variable number of fields.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                      EXIT SUBR$C                     Used for
     C                      RLABL           OUTFLD  ?       Output
     C                      RLABL           INFLD1  ?       Input
     C                      RLABL           INFLD2  ?       Input
     C                      RLABL           INFLDn  ?       Input
     C                      RLABL           OUTFLD  ?       Output
```

Parameter definition

OUTFLD A field up to 256 bytes long that will contain the concatenated contents of all the INFLDn fields. This parameter must again be specified as the last RLABL to "bracket" the variable number of input fields for SUBR$C.

INFLDn A variable number of fields up to 256 bytes long that contains the text to be concatenated.

Notes

- To insert a blank between fields, place an empty field in the output string where the blank should occur. To insert more than one blank, you must place multiple blank fields in the output string (i.e., one blank field per blank).
- The contents of the output string is not initialized to blanks before concatenation.

SUBR$X

Description

Extracts an arbitrary substring within a target string.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                    EXIT SUBR$X                    Used for
     C                    RLABL          FRMPOS  30      Input
     C                    RLABL          TOPOS   30      Input
     C                    RLABL          LEN     30      Input
     C                    RLABL          SOURCE  ?       Input
     C                    RLABL          TARGET  ?       Output
     C                    RLABL          RCODE   1       Output
```

Parameter definition

FRMPOS A 3-byte numeric field that contains the leftmost position of the string to copy from SOURCE (1 through 256 are allowed).

TOPOS A 3-byte numeric field that contains the leftmost position indicating where to copy the substring into TARGET (1 through 256 are allowed).

LEN A 3-byte numeric field that contains the number of bytes to copy (1 through 256 are allowed).

SOURCE A field up to 256 bytes long that contains the string to copy.

TARGET A field up to 256 bytes long that will contain the copied substring. Note that this field is not cleared prior to copying the substring from SOURCE. All positions not affected by the copy will remain intact.

RCODE A 1-byte field to contain the return code.
0 = Normal return
1 = FRMPOS, TOPOS or LEN value is not from 1 to 256
2 = FRMPOS/TOPOS plus LEN is invalid

Notes

- This subroutine can be used in conjunction with SUBR$F to find and extract a substring. Simply use the same source string and position returned by SUBR$F in RESULT as input to field FRMPOS in SUBR$X (see the following example).

Example

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                    MOVE 'I'       FUNC            Initial search
     C                    MOVE 'FIND ME' ARGMNT          Search argument
     C                    EXIT SUBR$F                    Perform the search
     C                    RLABL          FUNC    1
     C                    RLABL          FRMPOS  30
     C                    RLABL          ARGMNT  7
```

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                      RLABL          TARGET256
     C                      RLABL          LEFTP   30
     C                      RLABL          RIGHTP  30
     C*
     C              FRMPOS  IFGT 0                          If argument found
     C                      Z-ADD1         TOPOS            Set "to" position
     C                      Z-ADD7         LEN              Set length to copy
     C                      EXIT SUBR$X                     Extract substring
     C                      RLABL          FRMPOS  30
     C                      RLABL          TOPOS   30
     C                      RLABL          LEN     30
     C                      RLABL          TARGET
     C                      RLABL          SUBSTR256
     C                      RLABL          RCODE   1
     C            RCODE     IFEQ '0'                        If okay return
     C                      .
     C                      . Do something with the substring
     C                      .
     C                      END
     C                      END
```

SUBRAT

Description

Left-, right-, or center-adjust text within a field.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                      EXIT SUBRAT                     Used for
     C                      RLABL          OP      1        Input
     C                      RLABL          TEXT    ?        Input/Output
     C                      RLABL          RCODE   1        Output
```

Parameter definition

OP — A 1-byte field that contains the operation to perform.
  L = Left-adjust text
  R = Right-adjust text
  C = Center text

TEXT — A field up to 256 bytes long that contains the text to be adjusted.

RCODE — A 1-byte field to contain the return code.
  0 = Normal return
  1 = Invalid operation

Notes

- When centering text, if the text cannot be exactly centered, there will be an extra blank on the right end of the field. For example, if you attempt to center the string "NOW IS THE TIME" (i.e., 15 bytes) in a 20-byte field, there will be two blanks on the left and three on the right.

SUBRBX

Description

Convert binary to hex or hex to binary. For example, if converting from binary to hex, a 2-byte binary value of X'1234' will be converted to a 4-byte value of X'F1F2F3F4'.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                    EXIT SUBRBX                       Used for
     C                    RLABL          OP       1         Input
     C                    RLABL          BIN      ?         Input/Output
     C                    RLABL          HEX      ?         Input/Output
     C                    RLABL          RCODE    1         Output
```

Parameter definition

OP — A 1-byte field that contains the operation to perform.
- 1 = Convert binary to hex
- 2 = Convert hex to binary

BIN — A field up to 128 bytes long that contains the binary input or result.

HEX — A field up to 256 bytes long that contains the hex input or result. This field should be exactly twice as long as BIN.

RCODE — A 1-byte field to contain the return code.
- 0 = Normal return
- 1 = Invalid operation
- 2 = Field BIN is longer than 128 bytes
- 3 = Field HEX is not twice the length of BIN

Notes

- If converting from binary to hex, BIN should contain the binary input and HEX will contain the hex result. If converting from hex to binary, HEX contains the input and BIN will contain the result.

SUBRCS

Description

Convert text from upper case to lower case or lower case to upper case.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                    EXIT SUBRCS                       Used for
     C                    RLABL          OP       1         Input
     C                    RLABL          TEXT     ?         Input/Output
     C                    RLABL          RCODE    1         Output
```

Parameter definition

OP — A 1-byte field that contains the operation to perform.
L = Convert upper case to lower case
U = Convert lower case to upper case

TEXT — A field up to 256 bytes long that contains the text to be converted.

RCODE — A 1-byte field to contain the return code.
0 = Normal return
1 = Invalid operation

SUBRUP

Description

Unpack or pack a field.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                      EXIT SUBRUP                     Used for
     C                      RLABL           OP       1      Input
     C                      RLABL           INPUT    ?      Input
     C                      RLABL           OUTPUT   ?      Output
     C                      RLABL           RCODE    1      Output
```

Parameter definition

OP — A 1-byte field that contains the operation to perform.
1 = Unpack
2 = Pack

INPUT — A field up to 15 bytes long that contains the packed or unpacked input.

OUTPUT — A field up to 15 bytes long that will contain the packed or unpacked output.

RCODE — A 1-byte field to contain the return code.
0 = Normal return
1 = Invalid operation

Notes

- This routine performs signed unpacking/packing, such as that used in RPG programs.
- It is the caller's responsibility to ensure the output field is large enough to contain the resulting data.

LIBRARY MANIPULATION FUNCTIONS

The library function subroutines give you access to libraries and library members. The first two — SUBRLD and SUBRLR — provide the same capabilities as LISTLIBR procedure calls, except that with LISTLIBR the output must be placed in an intermediate disk file. SUBRSG is especially useful because it provides simultaneous access to an unlimited number of source or procedure members. These routines are especially useful for building software development tools because of their fast access to libraries and library members.

SUBRLD

Description

Return information about an individual library member (i.e., the directory entry) or an entire library.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                    EXIT SUBRLD                     Used for
     C                    RLABL      LIBNAM  8            Input
     C                    RLABL      MEMNAM  8            Input
     C                    RLABL      MEMTYP  1            Input
     C                    RLABL      DIRDS                Output
     C                    RLABL      RCODE   1            Output
```

Parameter definition

`LIBNAM` An 8-byte field that contains the library name where the member resides.

`MEMNAM` An 8-byte field that contains the member name.

- If MEMNAM is blank, the next directory entry is read.
- If MEMNAM contains a member name, the directory information for that member is returned.
- If MEMNAM contains a partial member name, the directory information for the next member matching the partial name is returned. A partial name is followed by an asterisk. (e.g., SUBR*)
- If MEMNAM contains *LIBR, information about the entire library is returned.

`MEMTYP` A 1-byte field that contains the member type. If MEMNAM contains *LIBR, the value in this field is ignored.

O = Object
P = Procedure
R = Subroutine
S = Source

DIRDS A data structure that contains the directory information returned. The format of the data structure and the field definitions for individual members are as follows:

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     IDIRDS       DS
     I                                          1   1 DRTYPE
     I                                          2   9 DRNAME
     I                                         10  15 DRADDR
     I                                         16  18 DR#TXT
     I                                         19  22 DRLINK
     I                                         23  27 DR#STM
     I                                         28  31 DRSCA
     I                                         32  33 DRRLD
     I                                         34  36 DRCORE
     I                                         37  37 DRATR1
     I                                         38  38 DRATR2
     I                                         39  39 DRATR3
     I                                         40  41 DRMRT
     I                                         42  43 DRREL
     I                                         44  46 DRTOTL
     I                                         47  47 DRATR4
     I                                         48  53 DRMOD
     I                                         54  59 DRDATE
     I                                         60  63 DRTIME
     I                                         64  65 DRATR5
     I                                         66  69 DRPTF@
     I                                         70  70 DRATR6
```

DRTYPE Member type
DRNAME Member name
DRADDR Disk address of member
DR#TXT Number of text sectors -- types O and R
Record length -- types S and P
DRLINK Link edit address
DR#STM Number of statements in member -- types S and P
DRSCA Start control address, entry point
DRRLD RLD displacement
DRCORE Core required, in sectors
DRATR1 Attribute byte 1
DRATR2 Attribute byte 2
DRATR3 Attribute byte 3
DRMRT MRTMAX count -- type O
MRT proc -- type P (will be hex FF)
DRREL Release level
DRTOTL Total number of sectors in module
DRATR4 Attribute byte 4
DRMOD Reference number

DRDATE Date member was changed/created (YYMMDD)
DRTIME Time member was changed/created (HHMM)
DRATR5 Attribute byte 5, module subtype
DRPTF@ Displacement of PTF table
DRATR6 Attribute byte 6

The format of the data structure and the field definitions for the entire library (*LIBR in MEMNAM) is as follows:

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     IDIRDS       DS
     I                                                1   6 LBFMT1
     I                                                7  11 LBLBSZ
     I                                               12  15 LBDRSZ
     I                                               16  21 LBUSEC
     I                                               22  27 LBASEC
     I                                               28  32 LBUDIR
     I                                               33  37 LBADIR
     I                                               38  43 LBBLIB
     I                                               44  49 LBELIB
     I                                               50  55 LBBDIR
     I                                               56  61 LBEDIR
     I                                               62  67 LBBMEM
     I                                               68  73 LBEMEM
     I                                               74  79 LBNMEM
     I                                               80  80 LBEXTN
```

LBFMT1 Format-1 address
LBLBSZ Library size in blocks
LBDRSZ Directory size in sectors
LBUSEC Used member sectors
LBASEC Available member sectors
LBUDIR Used directory entries
LBADIR Available directory entries
LBBLIB First sector of library
LBELIB Last sector of library
LBBDIR First sector of directory
LBEDIR Last sector of directory
LBBMEM First sector of members
LBEMEM Last sector of members
LBNMEM Next available member sector
LBEXTN Contains a Y if a library extent exists

RCODE A 1-byte field to contain the return code.
0 = Normal return
1 = Library not found
2 = Member not found or end of members
3 = DIRDS data structure too small

Example

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C*
     C*-- Read sequentially through all member types
     C*
     C                    MOVE 'TESTLIB 'LIBNAM  8        Set library name
     C                    MOVE *BLANKS   MEMNAM  8        Sequential search
     C*
     C                    MOVE 'O'       MEMTYP  1        Object
     C                    EXSR GETDIR
     C*
     C                    MOVE 'P'       MEMTYP           Proc
     C                    EXSR GETDIR
     C*
     C                    MOVE 'R'       MEMTYP           Subroutine
     C                    EXSR GETDIR
     C*
     C                    MOVE 'S'       MEMTYP           Source
     C                    EXSR GETDIR
     C*
     C*-- Now get info for entire library
     C*
     C                    MOVE '*LIBR   'MEMNAM           Library request
     C                    EXIT SUBRLD
     C                    RLABL          LIBNAM
     C                    RLABL          MEMNAM
     C                    RLABL          MEMTYP
     C                    RLABL          LIBDS
     C                    RLABL          RCODE
     C*
     C*-- Subroutine to read members until a "2" is returned in RCODE
     C*
     C           GETDIR   BEGSR
     C           RCODE    DOUEQ'2'
     C                    EXIT SUBRLD
     C                    RLABL          LIBNAM
     C                    RLABL          MEMNAM
     C                    RLABL          MEMTYP
     C                    RLABL          DIRDS
     C                    RLABL          RCODE
     C*          (At this point DIRDS contains the directory info
     C*           for the next member. Insert code as needed.)
     C                    END
     C                    ENDSR
```

SUBRLR

Description

Read individual library members in sector mode.

Calling sequence

This subroutine requires two different calling sequences, one to open the library member (the Open call) and another to read the sequential sectors (the Get Next call).

The calling sequence for an Open call is:

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                          EXIT SUBRLR                      Used for
     C                          RLABL         OP      1          Input
     C                          RLABL         LIBNAM  8          Input
     C                          RLABL         MEMNAM  8          Input
     C                          RLABL         MEMTYP  1          Input
     C                          RLABL         RCODE   1          Output
```

Parameter definition

OP — A 1-byte field that contains the operation to perform (O for Open request).

LIBNAM — An 8-byte field that contains the library name where the member resides.

MEMNAM — An 8-byte field that contains the member name.

MEMTYP — A 1-byte field that contains the member type.
- O = Object
- P = Procedure
- R = Subroutine
- S = Source

RCODE — A 1-byte field to contain the return code. For an Open request, return codes are:
- 0 = Member found in library, OK to issue Get Next requests
- 1 = Library not found
- 2 = Member not found

Calling sequence

The calling sequence for Get Next calls is:

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                          EXIT SUBRLR                      Used for
     C                          RLABL         OP      1          Input
     C                          RLABL         BUFF  256          Output
     C                          RLABL         RCODE   1          Output
```

Parameter definition

OP — A 1-byte field that contains the operation to perform (N for Get Next request).

BUFF — A 256-byte field to contain the member sector data.

`RCODE` A 1-byte field to contain the return code. For a Get Next request, return codes are:

0 = Normal return
3 = End of member

Notes

- On Get Next requests when a 3 is returned in RCODE, this indicates that the last sector in the member has just been returned in BUFF. Please be aware that this is different from reading records from a file in an RPG program, where the LR indicator does not come on until a subsequent read is attempted *after* reading the last record.

SUBRSG

Description

Read source or procedure members from a library in record mode.

Calling sequence

This subroutine requires two different calling sequences. One to open the library member (the Open call) and another to read the text records (the Get Next call).

The calling sequence for an Open call is:

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                        EXIT SUBRSG                     Used for
     C                        RLABL          OP       1       Input
     C                        RLABL          LIBNAM   8       Input
     C                        RLABL          MEMTYP   1       Input
     C                        RLABL          MEMNAM   8       Input
     C                        RLABL          PLIST   39       Input/Output
     C                        RLABL          RCODE    1       Output
```

Parameter definition

`OP` A 1-byte field that contains the operation to perform (O for Open request).

`LIBNAM` An 8-byte field that contains the library name where the member resides.

`MEMTYP` A 1-byte field that contains the member type.

P = Procedure
S = Source

`MEMNAM` An 8-byte field that contains the member name.

`PLIST` A 39-byte field to contain the source get parameter list. This is returned by SUBRSG to the user after each call. It is the user's

responsibility to keep track of each PLIST for the associated module. This field is used as input for subsequent Get Next calls and should not be modified by the user.

RCODE A 1-byte field to contain the return code. For an Open request, return codes are:

0 = Member found in library, OK to issue Get Next requests
1 = Library not found
2 = Member not found or bad member

Calling sequence

The calling sequence for Get Next calls is:

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                    EXIT SUBRSG                     Used for
     C                    RLABL        OP       1         Input
     C                    RLABL        PLIST   39         Input/Output
     C                    RLABL        TEXT   120         Output
     C                    RLABL        RCODE    1         Output
```

Parameter definition

OP A 1-byte field that contains the operation to perform (N for Get Next request).

PLIST A 39-byte field that contains the source get parameter list. This field is updated after every call.

TEXT A 120-byte field to contain the text record.

RCODE A 1-byte field to contain the return code. For a Get Next request, return codes are:

0 = Normal return, source line will be in TEXT
3 = End of file or bad member

Notes

- On Get Next requests when a 3 is returned in RCODE, this indicates that the last record in the member has just been returned in TEXT. Please be aware that this is different from reading records from a file in an RPG program, where the LR indicator does not come on until a subsequent read is attempted *after* reading the last record.
- SUBRSG may be used to read an unlimited number of library members simultaneously. After an Open request, save the contents of PLIST. Then when you want to read the text from a certain member, use the saved PLIST that corresponds with that member. Remember to resave the contents of PLIST after every call (see the following example).

Example

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     E*
     E                   PLST        5 39            Save up to 5 plists
     E                   PROC       10120            Proc save array
     E                   SRC        10120            Source save array
     E*
     C*
     C                    Z-ADD1         Z              Init index
     C                    MOVE 'O'       OP             Open request
     C*
     C*-- First open proc CUS001 in ARLIB
     C*
     C                    MOVE 'ARLIB   'LIBNAM         Library name
     C                    MOVE 'P'       MODTYP         Type = Proc
     C                    MOVE 'CUS001  'MODNAM         Proc name
     C                    MOVE *BLANKS   PLIST          Clear parm list
     C                    EXSR OPEN                     Open the member
     C*
     C          RCODE     IFEQ '0'                      If open successful
     C                    MOVE PLIST     PLST,Z          Save parm list
     C                    ADD  1         Z               Bump index
     C                    END
     C*
     C*-- Then open source CUS001 in ARLIB
     C*
     C                    MOVE 'S'       MODTYP         Type = Source
     C                    MOVE 'CUS001  'MODNAM         Source name
     C                    MOVE *BLANKS   PLIST          Clear parm list
     C                    EXSR OPEN                     Open the member
     C*
     C          RCODE     IFEQ '0'                      If open successful
     C                    MOVE PLIST     PLST,Z          Save parm list
     C                    ADD  1         Z               Bump index
     C                    END
     C*
     C*-- Read the first 10 records from each member
     C*
     C                    MOVE 'N'       OP             Get Next request
     C*
     C                    MOVE PLST,1    PLIST          Proc plist
     C                    Z-ADD1         X
     C*
     C                    DO   10        X
     C                    EXSR NEXT                     Get next record
     C                    MOVE TEXT      PROC,X         Save text record
     C                    MOVE PLIST     PLST,1         Save parm list
     C                    END
     C*
     C                    MOVE PLST,2    PLIST          Source plist
     C                    Z-ADD1         X
     C*
     C                    DO   10        X
     C                    EXSR NEXT                     Get next record
     C                    MOVE TEXT      SRC,X          Save text record
```

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                      MOVE PLIST     PLST,2             Save parm list
     C                      END
     C*
     C*-- Call SUBRSG to open a member
     C*
     C           OPEN       BEGSR
     C                      EXIT SUBRSG
     C                      RLABL          OP
     C                      RLABL          LIBNAM
     C                      RLABL          MODTYP
     C                      RLABL          MODNAM
     C                      RLABL          PLIST
     C                      RLABL          RCODE
     C                      ENDSR
     C*
     C*-- Call SUBRSG to get the next text record
     C*
     C           NEXT       BEGSR
     C                      EXIT SUBRSG
     C                      RLABL          OP
     C                      RLABL          PLIST
     C                      RLABL          TEXT
     C                      RLABL          RCODE
     C                      ENDSR
     C*
```

FILE MANIPULATION FUNCTIONS

A significant limitation of S/36 RPG is the inability to dynamically access files. RPG automatically opens files at program initiation and automatically closes them at termination. RPG requires that you know the record length and other file attributes at compile time. RPG makes no provision for retrieving file information dynamically or determining your absolute position within a file. And finally, RPG locks you out of some useful file operations supported by S/36 Disk Data Management.

The three routines in this section address these problems. The first, SUBRCO, lets you close and reopen files dynamically. If you've ever wanted to reread a sequential file after reaching EOF, you've felt the need for dynamic open/close. When applied to printer files, SUBRCO lets you release accumulated spooled output for printing without ending the program.

The second, SUBRFA, is really a "Swiss army knife" of file manipulation tools. It lets you open any file regardless of record length or other file attributes, retrieve those attributes from the S/36 VTOC, and create new files, all without requiring RPG F-specs or S/36 // FILE OCL statements. SUBRFA also lets you circumvent RPG's 15-file limit. These capabilities make SUBRFA ideal for custom file-manipulation tools, such as file editors, that must support late binding and dynamic file access. You also can use SUBRFA to perform a number of record-level I/O operations that RPG doesn't support, such as reading deleted records or reading the last record of a file.

Finally, SUBRRN gives you the ability to retrieve the physical relative record number (RRN) for the current record in any file, regardless of access method and whether or not the file is accessed via an alternate index. Combined with SUBRFA, SUBRRN lets you perform efficient file I/O by bypassing indexes when necessary.

You can put your future-compatibility fears aside when using these routines — all of them have easily reproduced counterparts on both the AS/400 and RS/6000 follow-on platforms. RPG/400 OPEN and CLOSE opcodes provide the same capability as SUBRCO, and execution-time file support provides the dynamic file access you get with SUBRFA. RPG/400 also supports the file feedback information provided by SUBRFA and SUBRRN. When you migrate to a new platform, you can encapsulate these functions in HLL programs having the same names — SUBRCO, SUBRFA and SUBRRN — as these assembler routines, and possibly avoid even the need to make source code changes to your application programs.

SUBRCO

Description

Close or reopen a file.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                             EXIT SUBRCO                        Used for
     C                             RLABL           OP      1          Input
     C                             RLABL           FILNAM  8          Input
     C                             RLABL           RCODE   1          Output
```

Parameter definition

`OP` A 1-byte field that contains the operation to perform.
- C = Close the file
- O = Open the file
- B = Close then reopen the file

`FILNAM` An 8-byte field that contains the file name.

`RCODE` A 1-byte field to contain the return code.
- 0 = Normal return
- 1 = File not found

Notes

- This subroutine is especially useful for closing and reopening PRINTER files. When a PRINTER file is closed, its spool file entry is released and another created upon reopening it.

- This subroutine cannot be called from COBOL.
- If you close a file, be sure to reopen it before end of job or a system error message will result.
- The name specified in FILNAM should be the file's name in the RPG F-spec.

SUBRFA

Description

Full procedural access to any disk file.

Calling sequence

This subroutine requires three different calling sequences: one to open/create the file, one to access the file, and one to close the file.

The calling sequence for an Open/Create call is:

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                    EXIT SUBRFA                       Used for
     C                    RLABL          OP      6          Input/Output
     C                    RLABL          DTF   128          Output
     C                    RLABL          NAME    8          Input
     C                    RLABL          PARMS   4          Input
     C                    RLABL          FEEDBK 20          Output
     C                    RLABL          DATE    6          Input
     C                    RLABL          BLDDS  20          Input
```

Parameter definition

`OP` A 6-byte field that contains *OPEN, left-justified. This field may also contain a return code if errors during open were detected.
#DTFE = DTF field is not 128 bytes
#NOTC = File open attempted but file in DTF not closed
##nnnn = If the return code contains ##nnnn (where nnnn is four digits), an error occurred during Special Allocate. In this case, the four digits are a MIC, which can be found by accessing the system message member ##MSG1 or by looking it up in the *System Messages* manual (SC21-7938).

`DTF` A 128-byte field to contain the file's DTF. This field is used on subsequent Access/Close calls and should not be modified.

`NAME` An 8-byte field that contains the file to be opened.

`PARMS` A 4-byte field that contains the open parameters: xyyz
x = Type of processing (I=Input, O=Ouput, U=Update)
yy = Share level (RR, RM, MM, MR, NO=No sharing, NW=New file)
z = Miscellaneous option (K=Keyed access, D=Return deleted records)

FEEDBK The file information feedback data structure. This must be the name of a data structure to receive information about the file after it is opened. The format of the data structure, and the field definitions, are as follows:

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     IFEEDBK     DS
     I                                              1   8 FFRUSD
     I                                              9  12 FFRECL
     I                                             13  20 FFCAPY
```

FFRUSD Number of records used
FFRECL Record length
FFCAPY File capacity in records

DATE A 6-byte field that contains the file date in YYMMDD format. If no date is specified, the file with the latest date will be opened.

BLDDS The new file build data structure. This must be the name of a data structure that contains information about the new file to be built if "NW" was specified in the PARMS parameter. *Note*: The DATE RLABL, if non-zero and non-blank, is used to set the date of the resulting file. The format of the data structure and the field definitions is as follows:

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     IBLDDS      DS
     I                                              1   1 BLTYPE
     I                                              2   2 BLRCBL
     I                                              3  10 BLCAPY
     I                                             11  14 BLRECL
     I                                             15  18 BLKEYS
     I                                             19  21 BLKEYL
     I                                             22  22 BLDELC
     I                                             23  23 BLDUPA
     I                                             24  31 BLXTND
```

BLTYPE Type of file (D,I,S)
BLRCBL Size is in records or blocks (R,B)
BLCAPY Size of file to be created
BLRECL Record length
BLKEYS Starting position of key if indexed
BLKEYL Length of key if indexed
BLDELC Delete capable? (Y,N)
BLDUPA Dup keys allowed? (Y,N)
BLXTND Default extend value

Calling sequence

The calling sequence for an Access call is:

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                         EXIT SUBRFA                         Used for
     C                         RLABL         OP       6            Input/Output
     C                         RLABL         DTF    128            Input/Output
     C                         RLABL         RECORD  ?             Input/Output
     C                         RLABL         KEY      ?            Input
                               or
     C                         RLABL         RRN     80            Input
```

Parameter definition

OP A 6-byte field that contains the operation code, left-justified. This field may also contain a return code if an error occurred. The valid operation codes and possible return codes are as follows:

Operation codes for indexed files.

*GETA	Get a record by key above the key specified (SETGT and READ)
*GETD	Get next duplicate key (READE)
*GETE	Get a record by key equal or above
*GETK	Get a record by key (CHAIN by key)

Operation codes for non-indexed files.

*ADDR	Add by RRN
*DELR	Delete by RRN
*FEOD	Force end of data — chops off file at current record
*GETR	Get a record by RRN (CHAIN by RRN)
*UPDR	Update by RRN

Operation codes for any file type.

*ADD	Add a record to the end of a file
*DEL	Delete a record
*GETC	Get current record
*GETF	Get the first record
*GETN	Get the next record
*GETL	Get the last record
*GETP	Get the previous record
*REL	Release a record
*SBOF	Set beginning of file — positions file pointer to first record
*SEOF	Set end of file — positions file pointer past last record
*UPD	Update the last record read

Possible return codes.

#41	Permanent I/O error
#42	End or beginning of file

#43	Invalid operation code
#44	Record not found
#45	Record update attempted before input
#48	Invalid relative record number
#49	Invalid data record
#50	Update key error
#53	Duplicate relative record number
#60	Duplicate key
#61	Duplicate key in another index
#62	Key out of sequence
#63	Invalid key length
#70	File is full
#75	Undefined access type
#99	File not opened
#BADOP	Bad operation code
#ADDRS	DTF address is greater than key field address
#RLERR	Record length of file opened exceeds RPG record buffer

DTF A 128-byte field that contains the DTF of the file to be accessed.

RECORD A field or data structure to contain the record. This field should be equal to or greater than the record length.

KEY A field containing the key for indexed processing,

or,

RRN An 8-byte numeric field that contains the relative record number for random processing.

Important note: For indexed file operations, the DTF field must be *physically* defined *before* the key field. If not, a return code of #ADDRS will result.

Calling sequence

The calling sequence for a Close call is:

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70.. .
     C                         EXIT SUBRFA                     Used for
     C                         RLABL          OP      6        Input
     C                         RLABL          DTF   128        Input/Output
```

Parameter definition

OP A 6-byte field that contains *CLOSE.

DTF A 128-byte field that contains the DTF of the file to be closed.

Example

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     I*
     I*-- File feedback data structure
     I*
     IFEEDBK     DS
     I                                        1   8 FFRUSD
     I                                        9  12 FFRECL
     I                                       13  20 ⁻FCAPY
     I*
     I*-- New file build data structure
     I*
     IBLDDS      DS
     I                                        1   1 BLTYPE
     I                                        2   2 BLRCBL
     I                                        3  10 BLCAPY
     I                                       11  14 BLRECL
     I                                       15  18 BLKEYS
     I                                       19  21 BLKEYL
     I                                       22  22 BLDELC
     I                                       23  23 BLDUPA
     I                                       24  31 BLXTND
     C*
     C*-- Open file TESTFL for input, DISP-SHRRR, keyed access
     C*
     C                     MOVE '*OPEN '  OP                Set op code
     C                     MOVE 'TESTFL ' NAME              File name
     C                     MOVE 'IRRK'    PARMS             Open parms
     C                     EXSR OPEN                        Open the file
     C           OP        IFNE '*OPEN '                    If error,
     C                     EXSR ERROR                       Do error stuff
     C                     END
     C*
     C*-- Read sequentially by key through the whole file
     C*
     C                     MOVE '*GETA '  OP                Set op code
     C           OP        DOUEQ'#42    '                   Until EOF
     C                     EXSR READ                        Read next record
     C           OP        COMP '*GETA '          50        If other
     C  N50      OP        COMP '#42    '         50        than EOF,
     C  N50                EXSR ERROR                       Do error stuff
     C                     .
     C*          (At this point RECBUF contains the next
     C*           record. Insert code as needed.)
     C                     .
     C                     END
     C                     END
     C*
     C*-- Close file TESTFL
     C*
     C                     MOVE '*CLOSE'  OP                 Set op code
     C                     EXSR CLOSE                        Close the file
     C*
     C*-- Open calling sequence
     C*
```

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C           OPEN      BEGSR
     C                     EXIT SUBRFA
     C                     RLABL          OP      6
     C                     RLABL          DTF   128
     C                     RLABL          NAME    8
     C                     RLABL          PARMS   4
     C                     RLABL          FEEDBK
     C                     RLABL          DATE    60
     C                     RLABL          BLDDS
     C                     ENDSR
     C*
     C*-- Access calling sequence
     C
     C           READ      BEGSR
     C                     EXIT SUBRFA
     C                     RLABL          OP
     C                     RLABL          DTF
     C                     RLABL          RECBUF 64
     C                     RLABL          KEY    10
     C*
     C*-- Close calling sequence
     C*
     C           CLOSE     BEGSR
     C                     EXIT SUBRFA
     C                     RLABL          OP
     C                     RLABL          DTF
     C*
```

SUBRRN

Description

Return the relative record number of the last record read or added to a file.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70...
     C                    EXIT  SUBRRN                     Used for
     C                    RLABL           FILNAM           Input
     C                    RLABL           RRN      80      Output
     C                    RLABL           RCODE    1       Output
```

Parameter definition

FILNAM An 8-byte field that contains the file name.

RRN An 8-byte numeric field that will contain the RRN.

RCODE A 1-byte field to contain the return code.
  0 = Normal return
  1 = File not found

Notes

- This subroutine must be called once for each requested file before attempting to return any RRNs. This is because a bit must be set on in the file's DTF (Define-the-file) by this routine to alert Data Management that RRNs are to be returned to the user (see the following example).
- The name specified in FILNAM should be the file's name in the RPG F-spec.

Example

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     FJUNKO    IF  F  10  10 10AI     1 DISK                    A
     IJUNKO      NS
     I                                        1  10 DATA
     C                    MOVE 'JUNKO   'FILE    8       Set file name
     C                    EXIT  SUBRRN                   Enable RRN returns
     C                    RLABL          FILE
     C                    RLABL          RRN     8
     C                    RLABL          RCODE   1
     C*
     C           KEY      CHAIN JUNKO               50   Chain to file
     C                    EXIT  SUBRRN                   Retrieve RRN read
     C                    RLABL          FILE
     C                    RLABL          RRN
     C                    RLABL          RCODE
```

SYSTEM MANAGEMEMENT FUNCTIONS

Keeping daily S/36 operations running smoothly often requires automating various system management functions that otherwise would consume a great deal of manual labor. IBM hasn't been very helpful, however, in giving you program-level access to the information you need to write your own system management tools. The half-dozen routines in this section give you that access.

SUBRSY returns the S/36 release and modification level, PTF level, and serial number — all vital statistics if you must maintain the system software for several S/36 installations. Getting this information manually requires running several programs and making a slew of navigations through complicated prompt screens. With SUBRSY, you can easily track the state of system software on each S/36 you manage.

A trio of routines — SUBRUF, SUBRUL, and SUBRUR — give you real-time status on how files and libraries are used on your system. SUBRUF returns a list of users for a particular file, SUBRUL the users for a library, and SUBRUR the users of records within a file. All of this information is indispensible when tracking down file, library, or record interlock problems.

The last two functions deal with system time and date. SUBRCT lets you change the system time without IPLing your system — a capability you'll dearly love twice a year if your time zone goes on daylight savings time.

SUBRDT returns the date formats currently set for the system and session dates — essential to know if you want your software to run correctly in the global marketplace.

Because these routines depend upon SSP internal data structures, you most likely won't be able to port them to the AS/400 or RS/6000. However, equivalent capabilities often are available through native application program interfaces (APIs) documented by IBM or the platform vendor.

SUBRSY

Description

Return the S/36 release and modification level, PTF level, and serial number.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                    EXIT SUBRSY                  Used for
     C                    RLABL          RELMOD 40     Output
     C                    RLABL          PTFLVL 40     Output
     C                    RLABL          SER#   60     Output
```

Parameter definition

RELMOD A 4-byte numeric field that will contain the release and modification level.

PTFLVL A 4-byte numeric field that will contain the current PTF level.

SER# A 6-byte numeric field that will contain the serial number.

SUBRUF

Description

Find jobs using a specified file and return attribute information for each job.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                    EXIT SUBRUF                  Used for
     C                    RLABL         FILNAM  8      Input
     C                    RLABL         JOB#    30     Input
     C                    RLABL         JOBDS          Output
```

Parameter definition

FILNAM An 8-byte field that contains the file name to check.

JOB# A 3-byte numeric field that contains the relative job number in case there are several. To inquire on all jobs using the file, call SUBRUF

continuously, incrementing JOB# until no more jobs are found (see SUBRUL example).

JOBDS A 47-byte data structure that will contain information about the job using the file. If this data structure is not at least 47 bytes long, no operation is performed. When the end of the job chain is reached, all fields in the data structure will contain blanks. The format of the data structure and the field definitions are as follows:

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     IJOBDS       DS
     I                                         1   8 USERID
     I                                         9  16 JOBNAM
     I                                        17  24 FSTPRC
     I                                        25  32 CURPRC
     I                                        33  40 PRGNAM
     I                                        41  46 JSTIME
     I                                        47  47 SHRLVL
```

USERID	User id
JOBNAM	Job name
FSTPRC	First level procedure name
CURPRC	Current level procedure name
PRGNAM	Program name
JSTIME	Job start time if running from the JOBQ
SHRLVL	Share level (DISP-SHRxx value, 0=RM, 1=RR, 2=MR, 3=N0, 9=MM)

SUBRUL

Description

Find jobs using a specified library and return attribute information for each job.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                    EXIT SUBRUL                    Used for
     C                    RLABL          LIBNAM  8       Input
     C                    RLABL          JOB#    30      Input
     C                    RLABL          JOBDS           Output
```

Parameter definition

LIBNAM An 8-byte field that contains the library name to check.

JOB# A 3-byte numeric field that contains the relative job number in case there are several. To inquire on all jobs using the library, call SUBRUL continuously, incrementing JOB# until no more jobs are found.

JOBDS A 46-byte data structure that will contain information about the job

using the library. If this data structure is not at least 46 bytes long, no operation is performed. When the end of the job chain is reached, all fields in the data structure will contain blanks. The format of the data structure and the field definitions are as follows:

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70.. .
    IJOBDS       DS
    I                                        1   8 USERID
    I                                        9  16 JOBNAM
    I                                       17  24 FSTPRC
    I                                       25  32 CURPRC
    I                                       33  40 PRGNAM
    I                                       41  46 JSTIME
```

USERID	User id
JOBNAM	Job name
FSTPRC	First level procedure name
CURPRC	Current level procedure name
PRGNAM	Program name
JSTIME	Job start time if running from the JOBQ

Example

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
    C                    MOVE 'TESTLIB 'LIBNAM          Set library name
    C      USERID        Z-ADD1         JOB#            Init index
    C                    DOUEQ          *BLANKS         DOU no more jobs
    C                    EXIT SUBRUL
    C                    RLABL          LIBNAM   8
    C                    RLABL          JOB#     30
    C                    RLABL          JOBDS
    C*         (At this point JOBDS contains the job info
    C*          for the next job. Insert code as needed.)
    C                    ADD  1         JOB#            Increment JOB#
    C                    END
```

SUBRUR

Description

Find which records are being used for a specified file and return the relative record number and attribute information for each job.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
    C                    EXIT SUBRUR                    Used for
    C                    RLABL          FILNAM  8       Input
    C                    RLABL          JOB#    30      Input
    C                    RLABL          JOBDS           Output
```

Parameter definition

FILNAM An 8-byte field that contains the file name to check.

JOB# A 3-byte numeric field that contains the relative job number in case there are several. To inquire on all jobs using the file, call SUBRUR continuously, incrementing JOB# until no more jobs are found (see SUBRUL example).

JOBDS A 49-byte data structure that will contain information about the job using the file. If this data structure is not at least 49 bytes long, no operation is performed. When the end of the job chain is reached, all fields in the data structure will contain blanks. The format of the data structure and the field definitions are as follows:

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     IJOBDS       DS
     I                                            1   8 USERID
     I                                            9  16 JOBNAM
     I                                           17  24 FSTPRC
     I                                           25  32 CURPRC
     I                                           33  40 PRGNAM
     I                                           41  48 RRN
     I                                           49  49 FLAGS
```

USERID User id
JOBNAM Job name
FSTPRC First level procedure name
CURPRC Current level procedure name
PRGNAM Program name
RRN Relative record number
FLAGS Flags byte
Bit 3 = Disallow locking same record under this task
Bit 4 = File is being closed
Bit 5 = Wait if locked
Bit 6 = Record owned
Bit 7 = Waiting for record

SUBRCT

Description

Change the system time or date without having to perform an IPL.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                    EXIT SUBRCT                      Used for
     C                    RLABL          TIME    60        Input
     C                    RLABL          DATE    60        Input
```

Parameter definition

TIME A 6-byte numeric field that contains the new system time in the format HHMMSS. If this field is blank, the time is not changed.

DATE A 6-byte numeric field that contains the new system date in the format YYMMDD. If this field is blank, the date is not changed.

Notes

- This routine will only change the SYSTEM date; all SESSION and PROGRAM dates will remain the same. To update the SESSION and PROGRAM dates, end all programs and sign off all terminals, then sign back on.

SUBRDT

Description

Return the SYSTEM and SESSION date formats.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                    EXIT SUBRDT                        Used for
     C                    RLABL           SYSFMT  3          Output
     C                    RLABL           SESFMT  3          Output
```

Parameter definition

SYSFMT A 3-byte field that will contain the SYSTEM date format.

SESFMT A 3-byte field that will contain the SESSION date format. The data returned in both fields will be in the format:
YMD = Year, month, day
MDY = Month, day, year
DMY = Day, month, year

DEVICE CONTROL FUNCTIONS

IBM built all kinds of nifty features into its workstation and printer devices and data management facilities. Unfortunately, RPG doesn't let you get at many of these features on the S/36 (although the capabilities are available on the AS/400 and RS/6000). The nine routines in this section let you directly access workstation, printer, spool, and timer functions that you can use to make your applications both easier to use and more functional. Most of these capabilities are available on the AS/400 and RS/6000, although you may have to make source code changes to achieve the same results.

SUBRCP

Description

Return the current row and column position of the cursor for a WORKSTN file.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                     EXIT SUBRCP                     Used for
     C                     RLABL          ROW     30       Output
     C                     RLABL          COL     30       Output
```

Parameter definition

ROW A 3-byte numeric field that will contain the current cursor row position.

COL A 3-byte numeric field that will contain the current cursor column position.

Notes

- The WORKSTN file may be defined as either Combined Demand (CD) or Combined Primary (CP).

SUBRDU

Description

This subroutine is used to replace whatever is in the DUP key save area for the current workstation. The DUP key save area is used to store the last command entered from the keyboard. Normally, the system updates this area as it sees fit; however, you can override its contents using this subroutine.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                     EXIT SUBRDU                     Used for
     C                     RLABL          DUP    120       Input
     C                     RLABL          RCODE   1        Output
```

Parameter definition

DUP A 120-byte field that contains the data to be placed in the DUP key save area.

RCODE A 1-byte field to contain the return code.
0 = Normal return
1 = Job not running from a workstation

Notes

- When this routine completes, pressing the DUP key at the command line will display the updated contents of the save area.

SUBREK

Description

Dynamically enable or disable command keys and enable or disable data to be returned with function keys.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                     EXIT SUBREK                    Used for
     C                     RLABL          MASK    4       Input
```

Parameter definition

MASK A 4-byte field containing a mask for enabling or disabling command keys. If the bit is on, the command key is enabled; if the bit is off, the command key is disabled.

Byte 1
bit 0: enable Cmd1
bit 1: enable Cmd2
bit 2: enable Cmd3
bit 3: enable Cmd4
bit 4: enable Cmd5
bit 5: enable Cmd6
bit 6: enable Cmd7
bit 7: enable Cmd8
Byte 2
bit 0: enable Cmd9
bit 1: enable Cmd10
bit 2: enable Cmd11
bit 3: enable Cmd12
bit 4: enable Cmd13
bit 5: enable Cmd14
bit 6: enable Cmd15
bit 7: enable Cmd16
Byte 3
bit 0: enable Cmd17
bit 1: enable Cmd18
bit 2: enable Cmd19
bit 3: enable Cmd20
bit 4: enable Cmd21
bit 5: enable Cmd22
bit 6: enable Cmd23
bit 7: enable Cmd24

Byte 4
bit 0: pass back print key
bit 1: pass back roll-up
bit 2: pass back roll-down
bit 3: pass back clear
bit 4: pass back help
bit 5: pass back record-backspace
bit 6: return data with function keys
bit 7: unused

Notes

- The WORKSTN file may be defined as either Combined Demand (CD) or Combined Primary (CP).

Example

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     I            DS
     I                                        1   4 MASK
     I                                        1   1 MASK1
     I                                        2   2 MASK2
     I                                        3   3 MASK3
     I                                        3   4 MASK4
     C*
     C                   BITOF'01234567' HEX00  1         Build binary 0
     C                   MOVE HEX00      MASK1            Clear
     C                   MOVE HEX00      MASK2            all
     C                   MOVE HEX00      MASK3            command
     C                   MOVE HEX00      MASK4            keys
     C                   BITON'026'      MASK1            Enable 1,3,7
     C                   BITON'2'        MASK3            and 19
     C                   EXIT            SUBREK
     C                   RLABL           MASK
```

SUBRLN

Description

Return the current line number for a PRINTER file.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                   EXIT SUBRLN                     Used for
     C                   RLABL          FILNAM  8        Input
     C                   RLABL          LINE#   30       Output
     C                   RLABL          RCODE   1        Output
```

Parameter definition

FILNAM An 8-byte field that contains the file name.

LINE# A 3-byte numeric field that will contain the current line number for the specified printer file.

RCODE A 1-byte field to contain the return code.
0 = Normal return
1 = File not found

SUBRMG

Description

Execute the MSG command, just as if it were being entered from the keyboard.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                      EXIT SUBRMG                 Used for
     C                      RLABL         INPUT   117   Input
     C                      RLABL         MIC     40    Output
```

Parameter definition

INPUT A 117-byte field that contains the MSG command as it would be entered from the keyboard. For example, to send a message to workstation W2, INPUT should look like this:

MSG W2,THIS IS THE MESSAGE TO BE SENT

MIC A 4-byte numeric field that will contain the MIC if errors occurred. The MIC will be a SYS message MIC, and can be looked up in the IBM *System Messages* manual (SC21-7938) or accessed using system message member ##MSG1.

Notes

- The command in INPUT should be left justified.

SUBRPC

Description

Enable/disable override cursor positioning, or position the cursor.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
    C                 EXIT SUBRPC                        Used for
    C                 RLABL           OP     1           Input
    C                 RLABL           ROW    20          Input
    C                 RLABL           COL    30          Input
    C                 RLABL           RCODE  1           Output
```

Parameter definition

OP — A 1-byte field that contains the operation to perform.
- 1 = Enable override cursor positioning. When this option is used, the cursor will remain where it last was displayed.
- 2 = Override current cursor row and column positions. The cursor will be positioned according to the values in ROW and/or COL. To leave the row or column unchanged, place zeros in the field.
- 3 = Revert to standard cursor positioning. Standard cursor positioning will be used, according to the screen format member.

ROW — A 2-byte numeric field that contains the desired cursor row position when using OP code 2.

COL — A 3-byte numeric field that contains the desired cursor column position when using OP code 2.

RCODE — A 1-byte field to contain the return code.
- 0 = Normal return
- 1 = Invalid operation

Notes

- Use subroutine SUBRCP to return the current cursor position.

SUBRPS

Description

Print the current screen displayed through a WORKSTN file.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
    C                 EXIT SUBRPS
```

Parameter definition

None.

Notes

- This subroutine provides the functional equivalent of the PRINT key operation. It can be used to print the current screen when the PRINT key is being used for another purpose.

SUBRSX

Description

Return the spool spool ID created for the specified PRINTER file.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                    EXIT SUBRSX                      Used for
     C                    RLABL          FILNAM  8         Input
     C                    RLABL          SPID    6         Output
```

Parameter definition

`FILNAM` An 8-byte field that contains the requested file.

`SPID` A 6-byte field that will contain the spool ID in the format SPnnnn. If no spool ID was returned, SPID will contain NOFIND.

SUBRWT

Description

Place a program in a wait state for the specified time interval.

Calling sequence

```
....+...10....+...20....+...30....+...40....+...50....+...60....+...70....
     C                    EXIT SUBRWT                      Used for
     C                    RLABL          HHMMSS  60        Input
```

Parameter definition

`HHMMSS` A 6-byte numeric field that contains the hours, minutes, and seconds to wait.

Notes

- This subroutine is the functional equivalent of the WAIT OCL statement.
- While the program is suspended, it is not looping, but uses the built-in timer mechanism of the system, and requires no machine resources during this period.

RBRIDG

At last count, more than 230 assembly language subroutines existed in the S/36 marketplace. These routines provide access to machine and operating system capabilities not directly accessible through high-level languages (HLLs), frequently making the impossible possible for many applications.

Alas, if your HLL of choice is COBOL, you're constrained to use a mere two dozen of these technical gems, because most of the routines only interface with RPG. The inscrutable IBM chose long ago to use different assembler subroutine linkage conventions for RPG and COBOL, making each camp's routines inaccessible to the other — until now.

Assembly language subroutine RBRIDG lets you "build a bridge" between your COBOL program and most, if not all, existing RPG assembler routines. Using RBRIDG is simply a matter of defining, in your COBOL program's WORKING STORAGE section, the RLABL parameters expected by any RPG assembler subroutines you want to use. Then, before calling the routine itself, you just make a call to RBRIDG to build a bridge to the desired routine.

Defining RLABLs

Subroutine RBRIDG interfaces with any assembler routine that you can call via the RPG EXIT operation, as long as the routine doesn't require indicator or array parameters. (COBOL has no RPG-like indicator area or array definitions.) To use subroutine RBRIDG in a COBOL program, you must first build, in the WORKING STORAGE section, an RLABL definition list for each subroutine you plan to call (see Figure 17.1 for an example of coding an RLABL definition list). In the 01-level data description entry, code a name for the definition list; later, you'll pass this name to RBRIDG. Each RLABL the target subroutine uses has a corresponding RLABL definition within this 01-level item. Each definition consists of three data items: type, length, and the data field itself. The type item is a one-character variable containing 'F' for RPG field RLABLs and 'D' for RPG data structure RLABLs.

The length item is a 2-byte COMP-4 (binary) variable containing the length of the RLABL field. The data field item represents the RPG field or data structure — it contains data being exchanged with the target subroutine — and is the only data description item you must name uniquely. All other items can have the name FILLER.

You can code as many RLABL definition entries as you like. After the last entry, code a 1-byte FILLER with a value of 'E' to mark the end of the definition list.

Making Your Call

With an RLABL definition list, using RBRIDG to call an RPG assembler subroutine is simple (see Figure 17.2 for an example). Just code a CALL to subroutine

Figure 17.1
Example of Coding an RLABL Definition List

```
 WORKING-STORAGE SECTION.
*
* RLABL definition list for three RLABLs:
*        RLABL          FIELD1        A 10-byte field
*        RLABL          FIELD2        A 1-byte field
*        RLABL          DSTRUC        A 300-byte data structure
*
 01  SUBRXX-RLABLS.
     05  FILLER.
         10  FILLER                PIC A          VALUE 'F'.
         10  FILLER                PIC 9999 COMP-4 VALUE 10.
         10  SUBRXX-FIELD1         PIC A(10).
     05  FILLER.
         10  FILLER                PIC A          VALUE 'F'.
         10  FILLER                PIC 9999 COMP-4 VALUE 1.
         10  SUBRXX-FIELD2         PIC A(1).
     05  FILLER.
         10  FILLER                PIC A          VALUE 'D'.
         10  FILLER                PIC 9999 COMP-4 VALUE 300.
         10  SUBRXX-DSTRUC         PIC A(300).
     05  FILLER                    PIC A          VALUE 'E'.
```

Figure 17.2
Example of Coding CALL Statements for RBRIDG

```
     CALL 'RBRIDG' USING SUBRXX-RLABLS.
     CALL 'SUBRXX'.
```

RBRIDG, specifying the name of the RLABL definition list in the USING clause. Immediately follow this CALL with a CALL to the target subroutine, without a USING clause. Note that you can't code any statements between the two CALLs. If subroutine RBRIDG detects a statement between the two CALL statements or an error in the RLABL definition list (e.g., the length item doesn't match the actual data field length), it halts with an error message. Figure 17.3 gives a sample COBOL program that calls the RPG assembler subroutine SUBRLD to read a library directory and print it.

Figure 17.3

Sample COBOL Program Using RBRIDG

```
*************************************************************
*                                                           *
*          This is a sample COBOL program that tests the    *
*          RBRIDG (RPG Assembler Subroutine Bridge).        *
*                                                           *
*************************************************************
 PROCESS MAP,OFFSET
 IDENTIFICATION DIVISION.
 PROGRAM-ID.      TBRIDG.
 AUTHOR.          MEL BECKMAN.
 INSTALLATION.    BECKMAN SOFTWARE ENGINEERING.
 DATE-WRITTEN.    22 FEBRUARY 1990.
 SECURITY.        NONE.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  IBM-S36.
 OBJECT-COMPUTER.  IBM-S36.
 SPECIAL-NAMES.
     SYSTEM-CONSOLE IS CONSOLE.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
* SUBRLD RLABL parameters:
*
*       RLABL        LIBNAM    8       Input
*       RLABL        MEMNAM    8       Input
*       RLABL        MEMTYP    1       Input
*       RLABL        DIRDS    80       Output
*       RLABL        RCODE     1       Output
*
 01  SUBRLD-RLABLS.
     05  FILLER.
         10  FILLER                PIC A          VALUE 'F'.
         10  FILLER                PIC 9999 COMP-4 VALUE 8.
         10  SUBRLD-LIBNAM         PIC A(8).
     05  FILLER.
         10  FILLER                PIC A          VALUE 'F'.
         10  FILLER                PIC 9999 COMP-4 VALUE 8.
         10  SUBRLD-MEMNAM         PIC A(8).
     05  FILLER.
         10  FILLER                PIC A          VALUE 'F'.
         10  FILLER                PIC 9999 COMP-4 VALUE 1.
         10  SUBRLD-MEMTYP         PIC A(1).
     05  FILLER.
         10  FILLER                PIC A          VALUE 'D'.
         10  FILLER                PIC 9999 COMP-4 VALUE 80.
         10  SUBRLD-DIRDS          PIC A(80).
     05  FILLER.
         10  FILLER                PIC A          VALUE 'F'.
```

Continued

Figure 17.3 Continued

```
          10  FILLER                PIC 9999 COMP-4 VALUE 1.
          10  SUBRLD-RCODE          PIC A(1).
      05  FILLER                    PIC A           VALUE 'E'.
 PROCEDURE DIVISION.
*
* Print all source member directory entries
*
 MAINLINE.
     MOVE 'NEWS3X   ' TO SUBRLD-LIBNAM.
     MOVE '         ' TO SUBRLD-MEMNAM.
     MOVE 'S'         TO SUBRLD-MEMTYP.
     MOVE '0' TO SUBRLD-RCODE.
*
     PERFORM PRINT-DIR-ENTRY
         UNTIL SUBRLD-RCODE IS NOT EQUAL TO '0'.
*
* Get out of Dodge.
*
 EXIT-PROGRAM.
     DISPLAY '*** Test of TBRIDG completed ***'.
     STOP RUN.
*
* Print a directory entry
*
 PRINT-DIR-ENTRY.
     CALL 'RBRIDG' USING SUBRLD-RLABLS.
     CALL 'SUBRLD'.
     DISPLAY SUBRLD-DIRDS.
```

Chapter 18

Profiling and Advanced Debugging

Where software is concerned, different people have different definitions for quality. To a programmer, quality means code that doesn't break after 5:00 PM or on weekends. To a manager, it means programs written so clearly that a new programming staff can be brought up to speed on a moment's notice. Users insist that quality is achieved only when programs run both quickly and correctly. And to the company controller, quality means software delivered on time without emergency loans or federal bailouts.

No matter whose definition you use, the path to quality is a narrow one. This chapter provides directions to that path, and advice on how to stay on it. First, you'll see how a simple development tool, called a profiler, lets you analyze source programs for both efficiency and reliability. The diskette accompanying this book includes a fully functional profiler for RPG. Next, we present a powerful debugging aid that helps you troubleshoot the worst sort of programming error: the intermittent bug. A single annoying bug can make even the best software look junky to users, but catching such bugs red-handed often seems impossible. This tool, included on diskette, lets you capture evidence at the scene of the crime. Finally, the collected wisdom of several experienced software sages reminds you of the fundamental processes behind achieving quality software. It's easy to forget the basics; these gurus capture the essence of quality in a series of memorable quotations.

Profile of a Profiler

When tuning your machine at the system level, you treat programs as black boxes, generalizing their behavior in terms of transactions per hour or megabytes per minute. Eventually, though, you must look at programs on a smaller scale. One tool that makes this task easier, called a *program profiler*, reveals where your programs spend their time, where you should optimize those programs, and how thoroughly they've been tested. Profiling will save you hundreds of programming hours you might otherwise spend tuning code or chasing bugs that should have been caught during testing.

The profiler described here is strictly for RPG, but the concepts apply to any language. Because the profiler is written in RPG, with only a few minor modifications you can migrate it to the AS/400. And even if you're a non-RPG shop, you can use this profiler as a model for building a profiler for your language of choice.

Profilers are a stock-in-trade programming tool; several different profiling techniques have evolved over time. Some profilers require special hardware

and super-accurate timers. Some use interrupts to inspect a program at random intervals and build a statistical map showing where the interrupts occurred most frequently. Each profiling technique has strengths and weaknesses Hardware profilers yield high accuracy but have a high cost. Statistical profilers give accurate results over long runs, but are inaccurate for the short runs or interactive transactions.

The profiler described here uses a technique called *statement counting*, which is the easiest technique to implement and provides accurate reporting regardless of execution time. The profiler works by counting the number of times each program source statement is executed. An example of a source listing produced by the profiler appears in Figure 18.1. Only C-specifications are executed in RPG, so the profiler prints only the calculation part of the code. The leftmost number on each statement shows the number of times the statement is executed during a test run. Statements that are executed most frequently are probably the ones consuming the most time, while statements that aren't executed at all during a test program run aren't tested and therefore flag inadequacies in your test data.

Inspect Figure 18.1, and you can see that the parts of the program executed most frequently are the "interpreter loop" (lines 47 through 54), the GETSYM routine (lines 62 through 70), and the DO routine (lines 76 through 83). The next busiest part of the program is the SC routine (lines 87 through 92). Clearly, any optimizing that reduces the number of times these statements are executed has the best chance of speeding up the program.

Some RPG statements, such as comments and END statements, are not actually executed and thus have no statement counts. However, one statement in the example program that should be counted, but isn't, is the MOVE instruction on line 68, which, according to the comments, is executed when the program senses an "end-of-line" condition. That this line is never executed indicates that the test data is incomplete — it never includes an "end-of-line" case. This example illustrates well the profiler's value in measuring test coverage. Without the profiler, this test data flaw might not be discovered until the program fails in a production environment, which is the very disaster you try to avoid by testing in the first place!

The RPG profiler consists of two procedures, PROFRPG and PROFPRT. You use the profiler in a three-step process. In the first step, you run the PROFRPG procedure to insert extra RPG statements, called *instrumentation code*, into the RPG program under test, creating a new version of the source program:

PROFRPG *program,library*

Where *program* is the name of the source program to be profiled, and *library* is the name of the library containing the program. PROFRPG stores the instru-

Figure 18.1

Sample Profiled Source Program Listing

```
10/02/92                                    RPG Execution Profile                                   Page    1
11:17:33          Program: FSMOCL                                          Library: NEWS3438
      0027 C*
      0028 C* Define local variables
      0029 C*
  365 0030 C                  MOVE *ZEROS    N       30        Next char index
  365 0031 C                  MOVE *ZEROS    S       30        Saved char index
  365 0032 C                  MOVE *ZEROS    STATE   30        Machine state
  365 0033 C                  MOVE *ZEROS    X       30        Column index
  365 0034 C                  MOVE *BLANKS   SYM     6         Input symbol
  365 0035 C                  MOVE *BLANKS   ACTION  2         Action code
  365 0036 C                  MOVE *BLANKS   KIND    8         Kind of name
      0037 C*
      0038 C* Initialization
      0039 C*
  365 0040 C                  Z-ADD1         STATE             Set initial state
  365 0041 C                  EXSR RD                          Read a line
    1 0042 C   LR             Z-ADD0         STATE             Quit if no input
      0043 C*
      0044 C* Interpreter loop
      0045 C*
  365 0046 C         STATE    DOWNE0                           While STATE-0
27755 0047 C                  EXSR GETSYM                       Get next symbol
27755 0048 C                  Z-ADD1         X                  Initialize lookup
27755 0049 C         SYM      LOKUPCOL,X                    11  Lookup column
15470 0050 C  N11             Z-ADD7         X                  (Default is other)
27755 0051 C                  MOVEASTT,STATE ROW,1              Extract table row
27755 0052 C                  MOVELROW,X     STATE              Set new state
27755 0053 C                  MOVE ROW,X     ACTION             Save action code
27755 0054 C                  EXSR DO                           Perform the action
      0055 C                  END                              End DO
      0056 C*
      0057 C* Routine to get the next symbol
      0058 C*
      0059 C*  Returns the next one-character symbol in the input line,
      0060 C*  or 'eol' if the end of the line is encountered
      0061 C*
27755 0062 C         GETSYM   BEGSR
27755 0063 C                  MOVE *BLANKS   SYM               Clear symbol field
27755 0064 C                  ADD  1         N                 Bump to next chr
27755 0065 C         N        IFLE 120                         If not end-of-line
27755 0066 C                  MOVE INP,N     SYM                Save the symbol
      0067 C                  ELSE                             Else it's e.o.l.
      0068 C                  MOVE 'eol'     SYM                So return 'eol'
      0069 C                  END                              End IF
27755 0070 C                  ENDSR
      0071 C*
      0072 C* Routine to do the specified action
      0073 C*
      0074 C*  Input: ACTION contains the action code to execut
      0075 C*
27755 0076 C         DO       BEGSR                            Depending on ACTION
      0077 C         ACTION   CASEQ'sc'      SC                 Save character
      0078 C         ACTION   CASEQ'pi'      PI                 Print identifier
      0079 C         ACTION   CASEQ'pk'      PK                 Print keyword
      0080 C         ACTION   CASEQ'pp'      PP                 Print parameter
      0081 C         ACTION   CASEQ'rd'      RD                 Read next line
      0082 C                  END                              End CAS
27755 0083 C                  ENDSR
      0084 C*
      0085 C* Routine to save symbol
      0086 C*
16016 0087 C         SC       BEGSR
```

Continued

Figure 18.1 *Continued*

```
16016 0088 C          S         IFLT 120                          If STR not full
16016 0089 C                    ADD  1       S                     Bump to next chr
16016 0090 C                    MOVE SYM     STR,S                 Save symbol
      0091 C                    END                               End IF
16016 0092 C                    ENDSR
      0093 C*
      0094 C* Routine to print identifier
      0095 C*
  364 0096 C          PI        BEGSR
  364 0097 C                    MOVE '  Ident:'KIND               Set kind
  364 0098 C                    EXCPTOLINE                        Print kind and name
  364 0099 C                    MOVE *ZEROS  S                    Reset STR index
  364 0100 C                    MOVE *BLANKS STR                  Clear STR array
  364 0101 C                    ENDSR
      0102 C*
      0103 C* Routine to print keyword
      0104 C*
 1183 0105 C          PK        BEGSR
 1183 0106 C                    MOVE 'Keyword:'KIND               Set kind
 1183 0107 C                    EXCPTOLINE                        Print kind and name
 1183 0108 C                    MOVE *ZEROS  S                    Reset STR index
 1183 0109 C                    MOVE *BLANKS STR                  Clear STR array
 1183 0110 C                    ENDSR
      0111 C*
      0112 C* Routine to print parameter
      0113 C*
 1456 0114 C          PP        BEGSR
 1456 0115 C                    MOVE '  Param:'KIND               Set kind
 1456 0116 C                    EXCPTOLINE                        Print kind and name
 1456 0117 C                    MOVE *ZEROS  S                    Reset STR index
 1456 0118 C                    MOVE *BLANKS STR                  Clear STR array
 1456 0119 C                    ENDSR
      0120 C*
      0121 C* Routine to read next line
      0122 C*
  729 0123 C          RD        BEGSR
  729 0124 C                    MOVE *ZEROS  N                    Reset INP index
  729 0125 C                    MOVE *BLANKS INP                  Clear INP array
  729 0126 C                    READ INPUT                      LR Read next record
  728 0127 C  NLR               EXCPTILINE                        Print input line
  729 0128 C                    ENDSR
```

mented version of your source program in a source library member named "P$*program*" in the same library as your original source (lines of instrumentation code are marked with "#+" in statement positions four and five). In step two, you compile the newly "instrumented" source program, and then run the resulting object program in its normal environment. During execution the instrumentation code collects statement execution counts; when the program ends, these counts are written into a data file. The third, and final, step is to run the PROFPRT procedure to print the profiled source listing:

PROFPRT *program,library*

Where once again *program* is the name of the profiled program, and *library* is the name of the library containing the program.

You need not worry about the "overhead," or extra time and memory, used by the instrumentation code except in programs approaching a 64 K

compiled size (obtained from the end of the compilation listing). Execution time overhead isn't important, because the profile results are unaffected by the time consumed by instrumentation code (and you won't leave instrumentation code in production programs). The memory overhead is 12 bytes per counted instruction — six for the counter, and six for the single machine instruction that increments the counter. In addition, your program size will increase about 1 K to accommodate the profile data file. A program containing 1,000 executable C-specs will only increase in size by 13 K.

You must carry out the second profiling step — compiling and running the profiled program — manually. The profiler can't automatically compile the target program because the program may require special values (such as MRTMAX) on the RPGC compile procedure. And only you know the OCL and execution environment required for your program to run. When you compile the instrumented source program, remember that although the source member name (P$*program*) differs from your original program name, the object program name is the same, and the instrumented object program will replace any existing version in the target library. In order to run your instrumented program, you must insert a // FILE statement for a file named P#*program*, which will contain statement counts. Figure 18.2 shows the // FILE statement added to the OCL for the sample program. You can leave this statement in your OCL even after removing instrumentation because SSP will ignore it if your program doesn't actually open the file.

Using a statement counting profiler for performance tuning requires that you keep in mind that all RPG statements don't execute in the same amount of time. In particular, I/O operations take much longer than arithmetic operations. As a rule of thumb, you can estimate the "cost" in execution time of various RPG statements using the table in Figure 18.3. Arithmetic operations such as ADD and SUB, and structural operations such as COMP, DO, IF and GOTO, run directly on the hardware in the S/36, and are thus fastest. Other arithmetic operations, such as MULT, DIV, and SQRT, are carried out more slowly, by compiler-provided runtime routines, because the S/36 lacks multiply and divide hardware instructions. I/O operations are the slowest because they must wait for the mechanical motion of devices such as disk arms and operator fingers. By using the factors in Figure 18.3, you can weight your profile statistics to give you a true measure of the time consumed by each statement.

You won't reap the benefits from useful tools such as this profiler unless you use them. So decide to make this tool work for you by requiring its use in your shop. All of your regression (i.e., stored data) tests should be profiled to ensure that the test data adequately exercises the code. You might even add a feature to the profile print program to flag executable lines that don't get executed (the diskette includes complete source code for the profiler). And before spending money on more memory or a faster CPU, profile your slowest

Figure 18.2

OCL Showing // FILE Statement for P#SAMP Counter File

```
// LOAD SAMP
// FILE NAME-P#SAMP,RECORDS-1000,EXTEND-1000
// RUN
```

Figure 18.3

Execution-Time Cost Multipliers for Various RPG Operations

RPG Operation	Cost milliseconds	Cost multiplier
Indexed disk I/O	100.000	25,000
Nonindexed disk I/O	35.000	8,750
Divide	10.000	2,500
Multiply	5.000	1,250
External program call	3.000	750
Array variable index	1.000	250
Array/table lookup	0.500	125
Other operations	0.004	1

applications to see if some simple coding changes won't ease your processing bottleneck.

RPG X-Ray Machine

Compared to the AS/400, IBM has largely shortchanged S/36 RPG programmers when it comes to debugging tools. The AS/400 has programmable breakpoints, dynamic variable display, the DUMP operation, formatted dump printouts, and the RPG DEBUG statement. The S/36 has the RPG DEBUG statement.

To be sure, some third-party debuggers for the S/36 offer capabilities as good as, and sometimes better than, those of the AS/400. But such third-party tools share a common limitation with the DEBUG statement: You first must compile the program to run specifically in "debug" mode before doing any debugging, and then you must recompile it without "debug" mode when you are through. Unfortunately, bugs don't always give you the kind of advance warning you need to isolate a failing program, recompile it with debugging statements, and try to re-create the original problem. And some program failures don't lend themselves to interactive debugging, because you don't know which variables to inspect — you need to see all the program's indicators and variables.

AS/400 programmers use a *formatted dump* to track down the cause of such intermittent bugs right where they happen. The formatted dump lists every RPG variable by name, along with its value at the time of the dump — a kind of "X-ray" picture of the entire program's internal state. The AS/400 programmer can request a dump in response to any program error message, or generate periodic "snapshot" dumps from within a program via the RPG DUMP statement. And because the formatted dump is actually a file, it can be sent from a remote site to a central programming site, giving AS/400 programmers an important long-distance problem-solving tool.

The S/36 also offers a built-in dump mechanism. But S/36 dumps are unformatted; to use them for debugging, you must manually look up the value of each variable by referring to hexadecimal addresses in the RPG compile listing — a tedious and error-prone proposition. Further, dumping a S/36 RPG program is a one-shot operation that terminates the program; S/36 RPG has no DUMP statement for making snapshot dumps.

To help overcome some of these S/36 debugging limitations, we've produced a utility called RPGDUMP to give you a formatted listing of an RPG dump file, showing the value of every indicator, field, array, and data structure. An assembly language subroutine, SUBRTD, provides the equivalent of RPG/400's DUMP statement, letting you make any number of snapshot dumps during a program's execution. Learning how these tools work and how to use them will put you on an equal footing with AS/400 RPGers.

Figure 18.4 illustrates the report output by utility RPGDUMP. The heading identifies the program dumped and the dump file used, including the program compile date and time — important for verifying which version of your RPG program was dumped. The heading also names a source member (the 'SymTable:' tag) that contains the symbol table, or list of variable names and addresses, for the RPG program. The RPGDUMP utility automatically creates the symbol table source member, as you'll see shortly.

After the headings, the report lists each RPG indicator that was on when the dump occurred, followed by the name, attributes (type and length) and value of each variable (field, data structure, or array), up to 100 bytes per line. Variables longer than 100 bytes have continuation lines (preceded by a colon). For arrays, the attributes column shows the number of elements and the element length. By default, array data appears as a contiguous string; you can optionally print the elements one per line, in which case the array element number precedes the colon on each line.

The first step to using utility RPGDUMP is to create a *symbol table source member* containing the names and memory addresses of all the variables in your RPG program. You need do this only once after you've compiled your program; thereafter you can use the same source member to format any number of dumps for that particular program. Running procedure RPGSYM brings up

Performance Tip

The S/36 doesn't natively provide source-level debugging capabilities, but here are two vendors whose products provide such capabilities for S/36 RPG:

RPG AID, CYBRA Corporation, One Riverdale Avenue, Riverdale, NY 10463, (212) 601-7100 or (800) 292-7288.

RPGD Interactive Source Debugger for S/36 RPG, BPS Information Services, Inc., PO Box 9, Marion, IA 52302, (319) 377-7599.

Figure 18.4
Sample Output from Utility RPGDUMP

```
11/20/90            Main Proc: QVUM01              RPG Formatted Dump            Dump File: #DUMP.08         Page  1
MIC-0016            Main Prog: QVUMEL              SymTable: S$QVUMEL            Dump Date: 90/11/15
                     Compiled: 90/11/15 13:00       Library: QVUDEV              Inv Level: 02 of 03

 Name   Attributes   Data    10         20         30         40         50         60         70         80         90        100
-----  ------------  -------------------------------------------------------------------------------------------- ------------
        Indicators:  L0 U8 01 02
CLINE# F        3.0  006
COL#   F        3.0  001
COL#R  F        3.0  131
COLON  F        1    :
CPAGE# F        6.0  000001

PFDATA F      128    0022    1   1   CMPLR   MEL      $PRINTDMW21258520001P1                                d
                    :
Y      F        8.0  00000025
Z      F        8.0  00000000
INFDS  D       32    00000                 35640000000000
JOBDS  D       50    MEL      W2131226QVU      QVU#     0000000020001066
MSG    A    5@ 36   1:Invalid command
                    2:* * * End of Spool File * * *
                    3:No room to keep additional lines
                    4:V 4.90 Copyright 1990 by Mel Beckman
                    5:Spool ID not available
QUE    A  10@ 80    1:  1    SP0016 RPGC     W2113444 MEL      $PRINTDM P1  1  0001  1   -    12
                    2:  2    SP0020 CMPLR    W1120556 MEL      $PRINTDM P1  1  0001  1   -    13
                    3:  3    SP0021 DMPFMT   W2120601 MEL      PRINT    P1  1  0001  1   -     1
                    4:  4    SP0022 CMPLR    W2125852 MEL      $PRINTDM P1  1  0001  1   -   105
                    5:* * * End of Spool Queue * * *
                     6:
                    7:
                    8:
                    9:
                   10:
SCL    A 256@  1    1:—+--10—+--20—+--30—+--40—+--50—+--60—+--70—+--80-- -+--90—+—100
                     :—+—110—+—120—+—130—+—140—+—150—+—160—+—170—+—180— +—190—+—200
                     :—+—210—+—220—+—230—+—240—+—250—+-
```

the symbol table creation prompt screen (Figure 18.5). Here you enter the name or forms number of the spool file containing the RPG compile listing for the program in question, followed by the name of the RPG program and the library where you want the symbol table stored. The name of the symbol table source member is always the name of your program preceded by 'S$', so you can store the symbol table in the same library containing your RPG source member. Symbol table source members do not occupy very much library space, so you may want to incorporate the RPGSYM procedure into your RPG compile procedure to automatically generate a new symbol table whenever an RPG program is compiled. Figure 18.6 shows an example symbol table source member.

Step two in using RPGDUMP is getting a task dump of your program. There are four ways to do this:

1. Respond with option "D" to any system message that allows option "3".
2. Use the "D" option on a CANCEL command (e.g., CANCEL W1082345,D).

Figure 18.5
Symbol Table Creation Prompt Screen

```
                              RPGSYM PROCEDURE

               Extracts the symbol table from an RPG compile listing
               and stores it in a source member for use by DMPFMT.

Spool ID (SPxxxx) or forms number (Fxxxx) . . . . . . . . . . .
  of RPG compile listing to process

Name of RPG source program  . . . . . . . . . . . . . . . . . .
  (RPGSYM stores the symbol table in a member named S$xxxxxx)

Name of library to contain symbol table source member . . . . .



Cmd7-Cancel            Cmd4-Put on job queue
```

Figure 18.6
Sample Symbol Table Source Member

```
1...:...10....:...20....:...30
F ATIME  0002 0 0000 00971
F BTIME  0006 0 0000 00871
F CTIME  0009 0 0000 00571
F WORD   0008   0000 00990
D TASKDS 0090   0000 01520
D TIMEDS 4060   0000 03001
A TBLA   0015   0008 02522
A TBLB   0015   0008 02630
A TBLC   0015   0008 02880
```

3. Run the IBM SETDUMP procedure.
4. Use assembly language subroutine SUBRTD in your RPG program.

The first two methods terminate your program after the dump is taken. The third method, the SETDUMP procedure, lets your RPG program continue execution after the dump, but using that method requires knowledge of RPG internals that lies outside the scope of this chapter. The fourth method, using subroutine SUBRTD (included on the "Desktop Guide" diskette), lets you take "snapshot" dumps without terminating your RPG program.

Figure 18.7 shows how to use subroutine SUBRTD. Following the EXIT SUBRTD statement, a single RLABL statement identifies a 1-byte flag variable. If the flag variable contains '0' your program keeps running after the dump; if it

Figure 18.7

Example of RPG Code Using SUBRTD

```
C                  MOVE '0'       DUFLAG           Don't terminate task
C                  EXIT SUBRTD                     Take a dump
C                  RLABL          DUFLAG
```

Figure 18.8

RPG Dump Formatter Prompt Screen

```
                          RPGDMP PROCEDURE

          Prints a formatted list of indicator, field, DS and
          array values from a task dump of an RPG program.

Dump file name (#DUMP.xx) . . . . . . . . . . . . . . . . . . .

Name of RPG source program  . . . . . . . . . . . . . . . . . .
  (to locate symbol table source member created by RPGSYM)

Name of library containing the symbol table source member . . .

Print array elements on individual lines? . . . . . . . . . . .  N

Invocation level to dump (01-main program)  . . . . . . . . . .  01



Cmd7-Cancel            Cmd4-Put on job queue
```

contains '1' your program terminates. Be careful when using SUBRTD; placing it in an infinite loop will generate a continuous series of dumps! The result of any of these four techniques is a dump file on disk. The name of the dump file is #DUMP.nn, where nn is automatically incremented from 00 to 99. If you already have dump files on disk, the system automatically skips the sequence numbers already in use. If you want to have more than 100 dump files on disk at one time, you can rename some files with names that don't end in .nn.

The third, and final, step to using RPGDUMP is generating the formatted dump listing. Running RPGDMP displays a prompt screen (Figure 18.8) that asks for the dump file name, the program name, the library containing the symbol table source member, and whether or not you want array elements printed on separate lines. Formatted dumps for small programs require only a few seconds; larger programs may require a minute or more.

With RPGDUMP, you have a utility comparable to the dump debugging

tools AS/400 programmers use. Combine that with a good third-party interactive debugger and you're on your way to achieving the same level of debugging productivity IBM provides on the AS/400.

Going for the Gold

Unfortunately, most of us programmers learned what we know about software testing at the school of hard knocks — at the cost of more than a few "flaky" programs haunting our professional past. And although we now acknowledge the need to test our software, we suffer from the mistaken notion that testing and debugging are the same thing. Although often performed together, testing and debugging are two quite different processes. While they share a common goal — to produce programs that work — the tasks involved differ in purpose and method. The purpose of testing is to reveal bugs; the purpose of debugging is to locate their causes and repair the damage. Testing is planned before it is performed; debugging is an impromptu activity. Testing usually does not require much detailed knowledge about a program's design; debugging is impossible without such knowledge. Testing takes a programmer methodically along a preordained path; debugging takes a programmer through the outer limits of intuition, experimentation, and blind luck. Finally, you test *before* you debug, for the results of testing drive the debugging process.

It's never too late to start testing the right way, by learning from those who have already gained experience. If you plan for thorough, intelligent testing, you can reduce greatly the time spent in wild and woolly debugging. In that spirit, we present a collection of "Golden Rules" — wise tips and techniques about software testing culled from recognized experts in computer science.

Testing can show the presence of bugs, but not their absence.

— *Edsger W. Dijkstra*
University of Texas

This is not an easy truth for most programmers to accept, but embracing it is essential to enlightened testing. How many times have you heard someone say "That's the last bug"? Of course, what they meant was "That's the last *known* bug." The program almost certainly contains other bugs waiting to be discovered. Even if there are no more bugs, no amount of testing can prove this fact positively because testing a program is not like testing a mechanical device for which all the parameters to be measured are known and limited in number. The number of permutations of input and output for even simple programs quickly becomes too large to evaluate systematically. The goal of testing is simply to reveal bugs so that they may be reduced to an acceptable

level. There is no such thing as "exhaustive" software testing — except for the most trivial problems.

Once you get over the hope for vindication through testing, it is a simple leap of faith to a slightly more cynical corollary to this rule: All non-trivial programs have bugs.

There can be no testing without specifications of intentions.

— Boris Beizer
Data Systems Analysts, Inc.

What is a bug? Simply put, a bug is program behavior that does not meet design specifications. This definition presumes that design specifications exist and that they were written before the program. With no specifications, any kind of program behavior could be correct. You may suspect that a report showing a 10-million-dollar net loss for your employer is wrong, but without specifications, who can know for sure?

Worse than no specification is the *de facto* specification cooked up after the program is running. "The program does *this*, so let's put *that* in the specification" is heard all too frequently in programming circles. Most programmers have a strong impulse to start coding immediately, as soon as they see partial solutions to a problem. But this kind of enthusiasm usually overlooks design considerations that may indicate a completely different approach. Often, after a flurry of coding activity and impressive results, you must throw out the entire coding effort because it doesn't solve the whole problem. When the desire to write wild code begins to overtake you, remember the words of Roy Carlston of the University of Wisconsin: "The sooner you start to code, the longer the program will take to write."

Good testing works best on good code and good design.

— Boris Beizer
Data Systems Analysts, Inc.

Good design and good coding are defensive activities, nipping bugs in the bud before they have a chance to develop complicated symptoms. On the other hand, bad design and coding are often so riddled with errors that even the best techniques may not find most of them. In such cases, rewriting the offensive code may be cheaper than trying to make it work. A combination of good design, good coding, and good testing yields a synergistic result: The sum is greater than the parts. It's obviously easier to plan tests when the design objectives are clearly spelled out. And tracking down the inevitable bugs revealed through testing is much easier if the code is well-structured and

readable. The effort put into laying a solid foundation pays off in painless, rapid testing and debugging.

It takes three times the effort to find and fix bugs in a system test. It takes 10 times the effort to find and fix bugs in the field than when done in a system test. Therefore, insist on unit tests by the developer.

— Larry Bernstein
Bell Communications Research

"Units" are the individual programs and routines that are later combined into a complete system. The kinds of tests performed on units, *structural tests*, are tied closely to internal details such as programming style, source languages, error-handling, and coding. For example, a test that follows a certain path through the code is structural, and a test that verifies the correct input, processing, and output of the unit is also structural.

The person in the best position to design structural tests for your programs is a programmer — not you, but someone you work with. Having a colleague design structural tests has two advantages: Someone with a fresh viewpoint is more likely to see bugs in your code, and the second walkthrough familiarizes another team member with code he or she may have to support later. And, of course, eventually you will find yourself wearing the structural testing hat for someone else's unit because part of your job as software engineer entails the ability to design and execute proper structural tests.

While designing structural tests for a unit, you should make note of input data that exercised each logic path. Then use this collection of inputs to develop a test strategy. Don't forget to profile the source code, using a statement-counting profiler such as the RPG profiler described earlier in this chapter. Profiling lets you easily spot program code not exercised by your test. If it isn't exercised, it isn't tested!

It behooves you to be diligent in unit testing because later, during the system test when all units are exercised as an integrated whole, your "public" bugs will surface for all to see.

Insist on a system test by an antagonistic third party.

— Boris Beizer
Data Systems Analysts, Inc.

System testing is a specialized kind of testing applied to the interfaces between units. The system test assumes that each unit is performing close to design specifications. Beginning the system test too early — when units contain many structural errors — is a waste of time. However, the system test does not have to wait until all units are ready; it can begin as soon as two units that communicate

with each other have passed unit tests. Performing this kind of "incremental" system testing effectively overlaps development and testing and can reveal design flaws before they become too firmly entrenched in code.

You should avoid wearing both programmer and system-tester hats on the same project. You have such a detailed knowledge of system internals that you almost certainly can't design unbiased tests. "Oh, yeah, I remember *that* routine. That's so simple it has to work; no need to exercise it unduly." An objective third party with no detailed knowledge of the underlying code will be suspicious of everything and much more likely to produce competent tests. If the third party has it in for you, the tests will be exceptionally thorough, resulting in a very clean system that increases your renown as a first-class programmer.

Of course, not every programmer has the luxury of an independent testing laboratory. If you must do your own system testing, get away from the project for a while to reduce the likelihood of your making unwarranted assumptions. And it can't hurt to get *any* kind of second opinion, even from a user. The best system testers are often not programmers.

Regression testing cuts test intervals in half.

— Larry Bernstein
Bell Communications Research

Regression testing is the practice of saving test input data and resulting output for reuse during future test sessions. After the first wave of testing is finished and bug reports are shipped off to the programmers, a second wave is necessary to verify that the programmers actually fixed the bugs and did not introduce others. In fact, it's likely you'll repeat the cycle of testing and debugging many times before the system is accepted. Saving the input makes subsequent test runs easier to reproduce, and saving the expected output greatly simplifies the post-test analysis. Next year, when you make program changes to implement user-requested enhancements, much of your test data will still be useful for validating the new version.

Regression testing of batch programs is straightforward: Save the input and output files and simply rerun the programs against the saved files to repeat a test run. Interactive programs, however, are more difficult to test consistently. Some operating systems provide scripting tools that let you save a "transcript" of an interactive session that you can play back during subsequent test runs to simulate the actions of a human operator. Unfortunately, S/36 SSP doesn't have such a tool, forcing you to maintain manual scripts from which to produce repeat performances. You might consider recording such scripts using PC-based session recording with a 5250 terminal emulator, which at least lets you automate playback for a single user. Even though they're tedious to prepare,

don't underestimate the importance of such scripts. Unless the interactive dialogues are simple, you will not be able to reproduce the same input and output consistently for each session without scripts, resulting in inconsistent testing.

Test the documentation while you test the system.

— Donald Knuth
Stanford University

Of course, you wrote the documentation as part of the design process, didn't you? The time to make sure that the documentation and system behavior agree is during the system test. Even a perfectly functioning system loses credibility with users if the manual says one thing and the programs do another. A third party performing system tests will find it convenient to pass judgment on the documentation at the same time to ensure a harmonious package is delivered to the end users.

Another point about documentation: Be sure the person testing it is a potential user. A programmer reviewing another programmer's operating instructions written for a non-technical user is likely to pass over documentation omissions the intended user would spot right away. Conversely, a non-programmer reviewing a programmer's technical manual might return a literary work of art that is a technical nightmare. And despite the best intentions, experience shows that documentation is seldom revisited once a system is put into successful operation. The system test is probably the last opportunity you'll have to debug the manuals thoroughly.

A program should ***WORK***.

— Brian Kernighan and P.J. Plauger
Bell Labs

Testing is important because programs should work — not just in the obvious cases, or in those cases you expect to encounter, but all the time. A program should survive assaults from bad data, frenzied operators, and malicious hackers. No programmer claiming to be a pro should invoke the magic chant "garbage in, garbage out"; this is just an excuse for a program that does not perform adequate checks on its input. Programs should be tested especially well near the edges, or boundary conditions, for reasonable behavior. Does your sort work with two elements? How about none? Does your program with a capacity to process 999 items behave sanely when presented with the thousandth item? Such boundary tests are among those most neglected by programmers yet the first to be breached by innocent users.

During the design and coding processes, you as a programmer introduce error-checks to provide early warning of data problems. If the error-check

was important enough to code, it's important enough to test. An essential part of the unit test is an exercise of every error trap in your program, especially the ones that say "This should never happen" in the comments. If an error trap is not tested, how can you rely on it to work in the heat of battle? Making a program work is obviously more difficult than making it run, but working programs handle all the little details that affect overall system quality.

Make it work first before you make it work fast.

— Brian Kernighan and P.J. Plauger
Bell Labs

Resist the urge to be clever, especially at the expense of clarity and reliability. A program that produces the right answer slowly is much more useful than one that produces the wrong answer quickly. Once everything is in good working order, you can take the time for intelligent performance tuning, which first requires a measurement of the entire application system to see where optimization will pay off. Shortening a loop from 10 seconds to 10 milliseconds enhances nothing but your ego if the loop is encountered only once a day. Note that there is nothing wrong with using efficient techniques if you don't have to give up readability, maintainability, or reliability.

There is nothing magical about testing and test design that immunizes testers against bugs.

— Boris Beizer
Data Systems Analysts, Inc.

Tests are rigorous procedures with prepared inputs, expected outputs, and formal procedures. Often you must write special programs to generate test data and validate results. It's not unusual for the number of lines of test code to exceed the number of lines of production code! Code is code, and code used in diagnostic tools is as likely to contain bugs as any other. At least in the early stages of testing, an incorrect test result may be due to test-program bugs as often as it is to bugs in the programs being tested. Even test data used in regression testing may contain latent bugs that jump out at unexpected times, forcing you to revise your previous opinions of the tests. Therefore, everything that applies to testing in general applies in a recursive sort of way to programs that aid the testing process.

Reasonable Goals

After 30 years of software development, nobody can yet say how much tuning, testing, and debugging is "enough." Because no amount of testing can prove the absence of bugs, and no amount of tuning can assure optimal performance,

you can only hope to make a convincing demonstration. What constitutes a convincing demonstration depends on the intended application of the system. A reliability level of 95 percent is perhaps adequate for software running on a home computer. But even 99.999 percent is not high enough for software controlling Star Wars systems if the one failure in 1,000 trials results in the end of life as we know it.

You must select goals reasonable for your programs. Once you've established those goals, you can use the wisdom contained in this chapter to measure program performance, locate and fix bugs, and systematically test against known standards — and you will have a better chance of producing quality software.

Chapter 19

Achieving Upward Compatibility

Nothing lasts forever, not even the S/36. Sooner or later most of us will migrate to a machine that is source code-compatible with the S/36. That machine might be the AS/400, it might be the RS/6000 running sophisticated S/36 emulation software, or it might be a machine we don't yet know about. As S/36 users and application developers, we face the challenge of writing new S/36 applications that will port easily to our next platform. This chapter is about creating those S/36 applications with an assumption that the software will probably spend its golden years on another machine.

With that in mind, and if you have ongoing application development, we strongly advocate using either ASNA's 400RPG or BPS's RPG II½ to add RPG/400-like capabilities to your RPG II code. By providing such features as externally described files, enhanced external program calls, additional RPG operation codes, and other RPG/400 language enhancements, ASNA's 400RPG and BPS's RPG II½ provide the needed leverage to write better and more maintainable RPG applications.

In this chapter, we will discuss these two packages and explain why you should use one of them, and we will suggest coding practices you should follow — especially if some day you plan to migrate your S/36 applications to an AS/400.

FIRST, A WORD OF CAUTION

Before we discuss the bells and whistles offered by ASNA's 400RPG or BPS's RPG II½, note that we do not encourage willy-nilly use of either of these products without regard for the future of your code. Such an approach could cause you enormous headaches because, with either of these products, it's possible to write "hybrid" RPG programs on your S/36 that won't compile in either the S/36 environment (S/36E) or the AS/400 native environment without considerable modification.

Another potential problem is that S/36 RPG II programs that use RPG/400 features will not be compatible with the AS/400's RPG II-compatible compiler, which is used to compile S/36 RPG II programs for use in the S/36E. If you use 400RPG or RPG II½ to develop S/36 applications, this incompatibility commits you to using the native RPG/400 compiler when you move to the AS/400. Fortunately, the AS/400 supports "mixed mode" applications. That is, programs compiled with RPG/400, a substantially enhanced version of RPG, can be called from the S/36 environment. For example, if FRED is a program compiled with RPG/400 and stored in library BARNEY,

you could use the following OCL from the S/36E to load and run that native program from the S/36E:

```
// LOAD FRED,BARNEY
// RUN
```

Technical Note

The AS/400 comes with two RPG compilers: RPG/400, the native RPG compiler, and the S/36 RPG II-compatible compiler, used to compile migrated stock S/36 RPG II programs to run in the S/36E. Programs compiled with RPG/400, a substantially enhanced version of RPG, also can run in the S/36E.

Although using 400RPG or RPG II½ features in your S/36 RPG code will require you to use the native RPG/400 compiler, you would not be forced to migrate immediately to the native environment. You can take your time learning about CL, the AS/400's messaging system, or any of the other features offered by the native environment.

If you know the rules for AS/400 native programming and understand the required file design, the use of 400RPG or RPG II½ will give you more programming features on your S/36 and a leg up on native AS/400 features when you move to the AS/400. Our advice: Use 400RPG or RPG II½ to develop new S/36 applications, but do so with at least a passing knowledge of what the RPG and database requirements are in the AS/400's native environment. Get copies of the *AS/400 RPG User Guide* (SC09-1348), the *RPG Reference* (SC09-1349), and the *Database Guide* (SC41-9659) for your S/36 shop so you'll know what's portable and what isn't. You can buy these manuals, or if you're lucky and know a shop with an AS/400, you can talk them out of their old manuals. The basic programming guidelines you're interested in won't change much from release to release and the older manuals will probably do the job for you. It also would be helpful for you to have a copy of IBM's Redbook, *S/36 to AS/400 Application Migration* (GG24-3250).

Technical Note

When you move to an AS/400, the choice of S/36E or native environment does not have to be an either/or decision. With attention to detail, programs developed using 400RPG or RPG II½ features can be migrated straight to the AS/400 native environment. They will compile with few changes and you will still be able to work with S/36 RPG II-only programs that you migrate to the S/36E. To make migration easier, consider keeping RPG II-only

code and code using 400RPG or RPG II1/2 features in separate libraries. The RPG II-only libraries will migrate to the S/36E and the libraries with enhanced code will migrate to the native environment.

Technical Note

The S/36 Unix/AIX environments, most notably Open RS/36 by Universal and Unibol by ICS Computing Group, don't impose the same restrictions on their S/36 environments that the AS/400 does. Open RS/36 and Unibol's S/36 environments support most of the RPG/400 enhancements offered by ASNA's 400RPG and many by BPS's RPG II1/2.

The Two Products

Section III stressed the importance of external program calls (EPCs), offered detailed comparisons of the EPC features offered by 400RPG and RPG II1/2, and noted that EPCs also are available in IBM's Value-Added Software Package (VASP) for the S/36. VASP, which is available free to all registered S/36 SSP users, is certainly worthwhile — you should get it as soon as possible — but its EPC implementation does not render 400RPG and RPG II1/2 obsolete. Not only do they both provide a better EPC implementation than VASP does, but they both are packed with many other valuable features.

While 400RPG and RPG II1/2 both offer many of the same features, 400RPG and RPG II1/2 look and feel more like distant cousins than siblings. By dovetailing itself into your system's RPGC procedure, ASNA's 400RPG integrates well with your S/36 to the point of seeming nearly invisible. Originally named RPG/III, 400RPG was initially a relatively straightforward implementation of RPG III from the S/38 for the S/36, with its primary features being EPCs and externally described files. With the advent of the AS/400, 400RPG has enriched its feature set and, in addition to providing an enhanced RPG language for the S/36, has positioned itself as a S/36-to-AS/400 conversion aid.

RPG II1/2 is a worthy competitor to 400RPG, but it has an entirely different personality. RPG II1/2 has the look and feel of having been developed late at night by an obsessed programmer. Its manual almost comes with its own Dorito fingerprints and Jolt Cola rings. Eschewing tight integration and user transparency, RPG II1/2 goes its own way, implementing many of its features through a main menu and throwing in a handful of interesting programmer goodies (e.g., code-generating macros and cross-referencing utilities.)

Those of you who swear by true-blue products and shiver at the notion of installing a third-party compiler on your S/36 can relax. Neither 400RPG nor RPG II1/2 are actually compilers; instead, they are source code preprocessors. You write your code with RPG/400 syntax and these products

massage that code, sometimes merging it with an assembler subroutine or two, and then submit that source code to your true-blue RPG II compiler. The translated code is discarded after being used by your RPG II compiler. By default, the compiler listing prints the source code you wrote, not the translated code the preprocessors created. 400RPG and RPG II½ both include methods of printing generated code. Seeing the preprocessed code is a handy way to see how the products do their magic; but beyond this point of interest, you'll generally want your original code printed on your compiler listings.

The implementations of 400RPG and RPG II½ are very similar. Because of this, we chose one of the products — 400RPG — to serve as our primary example of how the two products work. For a pre- and post-processed example, compare Figures 19.1a and 19.1b. Figure 19.1a is a complete RPG program that uses externally described files and some other RPG/400 features. Figure 19.1b is the code generated by 400RPG that will be submitted to the S/36 RPG II compiler. As mentioned earlier, unless you specifically ask to see it, the post-processed code is generally hidden from you — you'll never modify it with SEU or save it in a library. The original code as shown in Figure 19.1a is what you work with. We'll take a closer look at the code in these figures later.

Because both of these products add a preprocessing step, they add a slight amount of time to the compiling process. However, the time added is minimal and barely noticeable. As mentioned, 400RPG is invoked automatically by your RPGC procedure. RPG II½ can be called either from its main menu or invoked with a procedure call.

The Implementation of Externally Described Files

Both 400RPG and RPG II½ address a major limitation of S/36 RPG II by providing externally described files for the S/36. With RPG II on the S/36, all data files are program-described (i.e., all data file descriptions are internally bound to programs). As many different descriptions of your data files exist as you have programs using them. Any changes you make to data file layouts require finding all programs that reference that file and changing those programs — *one at a time* — a slow, laborious, error-prone process.

Very few programs use *every* field in a file, so it probably takes several programs just to form a complete description of a data file. With a little luck, all the programs you've found that describe a file describe that file consistently — with a little less luck you'll spend a couple of days with POP scanning through source members trying to determine what field really ends in which position.

In the AS/400 native environment, as well as on the S/38, data files are externally described (for the record, AS/400 data files can also be program-

Figure 19.1a

RPG Program Using Externally Described Files

```
 1 H      Ø64                                                 EXTFIL
 2 FJRL      IP  E                       DISK
 3 FBANK     UF  E              K        DISK                        A
 4 I#JRL
 5 I                                                        JJACCTL3
 6  **
 7 C            #BAKEY    KLIST
 8 C                      KFLD           JJSTOR
 9 C                      KFLD           JJACCT
1Ø C                      KFLD           JJDEPT
11 C                      KFLD           JJLOC
12 C            *LIKE     DEFN BABAL     VTOTAL
13 C    L3                Z-ADDØ         VTOTAL
14  **
15 C                      ADD  JJAMT     VTOTAL
16  **
17 CL3                    EXSRWRTBNK
18  **
19 CSR          WRTBNK    BEGSR
2Ø C                      Z-ADDJJSTOR    BASTOR
21 C                      Z-ADDVTOTAL    BABAL
22 C           .
23 C           . Fill other BANK fields as needed
24 C           .
25 C            #BAKEY    CHAINBANK                   55
26 C      N55             UPDAT#BANK
27 C       55             WRITE#BANK
28 C                      ENDSR
```

described, but that is very rarely done in the native environment). With externally described files, data file descriptions are stored in only one place and never in any programs. All programs simply reference the external file description at compilation time.

Externally described files ensure that your data is program-independent (i.e., your data files don't depend on programs to describe them). File modifications such as changing the key position, field lengths or record lengths do not require modifying the programs that use that data. After changing the external file description, the dependent programs need only be recompiled. Externally described files also ensure that files are referenced the same way in every program, that field names are the same in every program and that you have, in one place, a comprehensive description of the file.

Both 400RPG and RPG II½ offer a couple of ways to create external file descriptions and both products vary a bit in their implementation, but the results are the same: program-independent external file descriptions for your S/36. RPG II½ uses the most proprietary method of describing files externally. It uses a source member that is a variation of RPG's F- and I-specs. These members can be created manually with SEU or they can be created from

Figure 19.1b

Code Generated by 400RPG

```
H     064                                                     EXTFIL
FJRL      IP  F       72            DISK
FBANK     UF  F      128  9AI     01 DISK                          A
IJRL      NS  01
I                                        1   20JJYEAR
I                                        3   70JJJENO
I                                        8   90JJSTOR
I                                       10  130JJACCTL3
I                                       14  140JJDEPT
I                                       15  160JJLOC
I                                       17  21 JJOB
I                                       22  230JJFOLI
I                                       24  24 JJTYP
I                                       25  27 JJSRC
I                                       28  290JJPER
I                                       30  350JJDATE
I                                       36  400JJCHEK
I                                       41  41 JJCODE
I                                       42  66 JJDESC
I                                   P   67  712JJAMT
I                                       72  72 JJPOST
IBANK     NS  01
I                                        1   20BASTOR
I                                        3   60BAACCT
I                                        7   70BADEPT
I                                        8   90BALOC
I                                       10  33 BANAME
I                                       34  57 BADDR
I                                       58  672BABAL
C                     EXIT SUBR3V
C                     EXIT SUBR3M
C           *LIKE     DEFN BABAL     VTOTAL
C   L3                Z-ADD0         VTOTAL
 **
C                     ADD  JJAMT     VTOTAL
CL3                   EXSR WRTBNK
 **
CSR         WRTBNK    BEGSR
C                     Z-ADDJJSTOR    BASTOR
C                     Z-ADDVTOTAL    BABAL
C          .
C          . Fill other BANK fields as needed
C          .
C           #BAKEY    CHAINBANK                 55
C                     EXIT SUBR3A
C                     RLABL          #BAKEY  9
C                     RLABL          JJSTOR
C                     RLABL          JJACCT
C                     RLABL          JJDEPT
C                     RLABL          JJLOC
C                     RLABL          IN00
C      N55            EXCPT#$E001
C       55            EXCPT#$E002
```

Continued

Figure 19.1b Continued

```
C                        ENDSR
OBANK        ADD           #$E002
O                          BASTOR     2
O                          BAACCT     6
O                          BADEPT     7
O                          BALOC      9
O                          BANAME    33
O                          BADDR     57
O                          BABAL     67
OBANK                      #$E001
O                          BASTOR     2
O                          BAACCT     6
O                          BADEPT     7
O                          BALOC      9
O                          BANAME    33
O                          BADDR     57
O                          BABAL     67
```

IDDU or an existing RPG program's F- and I-specs. RPG II½ includes utilities to translate RPG programs written on the S/36 using its externally described files to RPG/400-compatible native RPG programs.

400RPG uses a less proprietary method of providing externally described files on the S/36. It uses an almost completely RPG II-compatible F- and I-spec source member to describe a file. The only variation from RPG's F- and I-spec syntax is that on the record-level I-spec, a record-format name must be coded in columns 53-58. This record name will be used by RPG I/O operations. Multiple record-format files require a different record name for each format.

Figure 19.2 shows the 400RPG external file description member for the BANK file. It is from this member that file attributes such as record length, key length and position, as well as field names and positions, are known by the RPG program. BANK has a single record format named #BANK. Note that for program clarity (this is not a rule imposed by 400RPG), all of BANK's fields start with "BA," and that no other fields will ever start with "BA." This field name scheme ensures that when you see a field starting with "BA," you'll know it belongs to the BANK file.

Any 400RPG program can reference the BANK external file description by using a BANK F-spec as shown in line 3 of Figure 19.1a. The "E" in the F-spec's column 19 tells 400RPG that the BANK file is externally described and the optional "K" in column 31 tells the program the BANK file is being read by key. Record lengths and key positions are *not* entered in a program's F-spec for externally described files. These values will be read from the external description. The file type and designation (columns 15 and 16), the mode of processing (column 28), and the file addition value (column 66) are coded as required in each program.

Figure 19.2

400RPG External File Description Member for BANK File

```
FBANK    IP    128 128  9AI    01 DISK
IBANK    NS  01                              #BANK
I                                      1   20BASTOR
I                                      3   60BAACCT
I                                      7   70BADEPT
I                                      8   90BALOC
I                                     10  33 BANAME
I                                     34  57 BADDR
I                                     58  672BABAL
```

Field overrides, to specify such things as control breaks and changing externally described field names, are achieved easily with a couple of extra I-specs in the program. Lines 4 and 5 of Figure 19.1a specify an overriding control break for the externally described file JRL by referencing a record format name being overridden and a field name. Except for field overrides like this, no other input specs are required in the RPG program. Note that there are no other I-specs in Figure 19.1a. You'll also notice that there are no output specs coded. 400RPG puts the input specs from the external description (after recording any field overrides) into the RPG program and also uses them to create output specs.

Technical Note

On your S/36, programs using externally described files as provided by 400RPG or RPG II½ will coexist nicely with RPG II programs that don't use externally described files. You can start using externally described files on your S/36 gradually, describing a couple of your most-used files and then working your way slowly into externally describing all your files.

RPG/400 FILE OPERATION CODES

To perform file I/O for externally described files, both 400RPG and RPG II½ use three RPG/400 operation codes: WRITE, which adds a new record to a file; UPDAT, which updates an existing record in a file; and DELET, which deletes a record from a delete-capable file. The operation codes are implemented in the same fashion by 400RPG and RPG II½.

Figure 19.1a shows the WRITE and UPDAT operation codes in action as records are added or updated to the BANK file, depending on the results of

Figure 19.3a

Using RPG II to Describe a Multipart Key

```
IJRL      NS  01
I                                     1   20JJSTOR
I                                     3   60JJACCT
I                                     7   70JJDEPT
I                                     8   90JJLOC
I                                     1   9 #BAKEY
I                                    10  10 JJTYPE
```

Figure 19.3b

Using a Data Structure to Describe a Multipart Key

```
I           DS
I                                     1   20JJSTOR
I                                     3   60JJACCT
I                                     7   70JJDEPT
I                                     8   90JJLOC
I                                     1   9 #BAKEY
```

Figure 19.3c

Using KLIST to Assemble Fields of a Multipart Key

```
C          #BAKEY    KLIST
C                    KFLD           JJSTOR
C                    KFLD           JJACCT
C                    KFLD           JJDEPT
C                    KFLD           JJLOC
```

the CHAIN operation code. One side benefit of the WRITE operation code is that it initializes any numeric fields (including packed and binary fields) to zero that have not already been initialized in your program. (Your CFO will appreciate the fact that no more pay checks will be written for $404040.40!)

To describe multipart keys, RPG/400 uses the KLIST operation code, which is supported by 400RPG and RPG II1/2. Methods used by RPG II to describe a multipart key, such as redefining fields on the input specs (Figure 19.3a) or using a data structure (Figure 19.3b), are not used by RPG/400. Figure 19.3c shows KLIST and how it is used to "assemble" the fields of a multipart key. Note that neither 400RPG nor RPG/400 allow using the KLIST value

for anything other than file operations. If you need to assemble its value for other reasons, you'll need to use a data structure or MOVE/MOVEL.

Technical Note

DDS, which stands for Data Description Specifications, performs a combination of F-, I-, S-, and D-spec responsibilities for the AS/400. The AS/400 uses DDS to describe data files, printer files, and workstation files. By providing DDS support, 400RPG offers RPG/400 compatibility with externally described files that RPG II½ does not. 400RPG's DDS support allows you to use AS/400-compatible DDS to externally describe your data files, and also includes support for alternate indexes and for field reference files (which provide a method of further reducing the redundancy required to define each field in a file).

Line 12 in Figure 19.1a reveals another subtle advantage of using externally described files. Here, RPG II's DEFN operation is used to declare the size of field VTOTAL to be the same as field BABAL in the BANK file. By using the DEFN operation, the size of field VTOTAL is *not* hardwired in the program. With each compilation, it will adjust itself to the size of BANK's BABAL field. If the size of field BABAL were ever changed in BANK's external description, using DEFN ensures that field VTOTAL will always be large enough. In addition to data files, 400RPG and RPG II½ also allow externally described workstation files and externally described data structures. RPG II½ also supports externally described printer files.

Using externally described files with S/36 RPG II is perhaps the most significant step you can take to improve the quality of your S/36 source code. Externally described files, make your programs shorter, more concise and more readable. This is *the* way programs are written on the AS/400. By learning how to use externally described files now on the S/36, you will find that you have shortened the learning curve when you migrate to a future platform.

Other RPG/400 Features

Although gaining the capability of externally described files and enhanced EPCs are the primary reasons to use 400RPG or RPG II½, both products also offer other valuable features as well. For example, both support RPG/400's AND/OR operation codes. Take a look at the indicator-laden RPG II code in Figure 19.4a. Here, an action is to be performed if a customer's balance is equal to or greater than her credit limit or if her balance is equal to or greater than her high balance and if her balance is greater than $500. Figure 19.4b shows how much cleaner the credit limit test is using AND/OR operations.

Figure 19.4a

Indicator-Laden RPG II Code

```
C                 SETOF                56
C          BAL    COMP LIMIT         55  55
C  N55     BAL    COMP HIBAL         56  56
C   55     BAL    COMP 500           55
C   55            ...do something
C   55            ...do something
C   55            ...do something
```

Figure 19.4b

400RPG Code Using AND/OR Operations

```
C          BAL    IFGE CLIMI
C          BAL    ORGE HIBAL
C          BAL    ANDGT500
C                 ...do something
C                 ...do something
C                 ...do something
C                 ENDIF
```

Both 400RPG and RPG II½ also support the RPG/400-compatible *IN,xx array and *INxx fields to reference indicator values as variables. The *IN,xx array is a 99-character array of 1-byte alphanumeric characters. The value of each element corresponds to the corresponding indicator. If the array element has a value of '1', that element's corresponding indicator is set on; if its value is '0', the indicator is set off. For example, the following line of code:

```
C                          MOVE '1'        *IN,55
```

causes indicator 55 to be set on. The *INxx field works in much the same fashion as the *IN,xx array except, as you would probably expect, you can't use the *INxx field with array operations.

Referencing indicators as variables can result in some abysmal RPG. Quick now, what does this line of code do:

```
C                          MOVEA'010011'   *IN,50
```

This code sets off indicators 50, 52, and 53, and sets on 51, 54, and 55. Is that clear or what? For some reason, some RPG/400 programmers think this is better coding practice than using a few SETON and SETOF statements. However, used sparingly and commented clearly, you will find good uses for

referencing indicators as fields, especially when you need to toggle indicators that control workstation display attributes.

RPG400 and RPG II½ have many more tricks up their sleeve than we've discussed. Check them out. As we said earlier, *if* you have ongoing program development on your S/36 and *if* you're willing to roll up your sleeves and learn a little about the AS/400, you *need* one of these products.

ADDITIONAL TIPS FOR UPWARD COMPATIBILITY

Regardless of which language strategy you're using on the S/36 — straight RPG II or RPG II with 400RPG or RPG II½ — you should consider several other things as you develop new applications. The following tips apply primarily if your migration platform will be the AS/400, but most have some merit if you plan a move to a Unix S/36 environment.

Data File Usage on the S/36

When you venture outside the S/36E on the AS/400, you'll find that the rules of file usage change considerably from the old RPG II-S/36 days. RPG/400 doesn't support direct or chained access methods (a "D" or "C" in column 16). In both cases, the AS/400 native environment expects full procedural files instead (an "F" in column 16). For any new application development on the S/36, use full procedural files wherever you can. You won't miss any features by using full procedural files on your S/36, and programs with full procedural files will migrate more easily to the native AS/400 environment.

It's common for file layouts on the S/36 to include many record types per file. For example, when designing an order-entry system for the S/36, it is common S/36 practice to put both order header and order detail records in the same file. Using a record ID code somewhere in the file, the RPG program knows which record it is working with. Multiple record-type files are not directly supported on the AS/400. Design all your new S/36 applications to use only one record type per file.

Code to Avoid Decimal Data Errors

The S/36 uses a zoned-decimal format for internal representation of numeric data. When reading in zoned decimal format, the S/36 uses only the digit portion of each byte to determine its numeric value. Therefore, any alphanumeric character can be read as a numeric value — for example, x40 (a blank) has the numeric value of zero (xF0), xC1 ("A") has the numeric value of one (xF1), and so on. (This is why, on the S/36, after moving a 5-byte blank character field into a 5-byte numeric field, the numeric field has a zero value.) However, the AS/400 native environment is different. It uses a packed format for internal representation of numeric data. With this packed format, both the zone and

digit values are used to determine the value of a byte. Therefore, in the AS/400 native environment, the numeric value of a blank (x40), is 40, not zero.

The difference in data representation often manifests itself in decimal data errors. One place decimal data errors commonly occur is when a S/36E program calls an AS/400 native program. For example, take a look at the small S/36E program in Figure 19.5a and the native program it calls in Figure 19.5b. This should work: The number of parameters being passed match and their attributes match. But it won't. The native program expects numeric data in packed format and when the S/36E program sends it zoned decimal numeric values, a run-time error occurs. You can avoid this problem by mapping the parameters that the native program receives to alpha characters with a data structure, as shown in Figure 19.5c. Here, the native program accepts the alphanumeric values passed to it without error, and the data structures perform the necessary implicit data conversion.

If you use external program calls on the S/36, and you plan to port that code to the AS/400 native environment, consider always passing parameters as character values and having your programs convert the values to numeric when necessary. This practice will avoid lots of annoying run-time decimal data errors on the AS/400.

Assembler Subroutines

Whatever your language strategy, using assembler routines in your code *can* bring migration to another platform to a grinding halt. However, the AS/400 and the Unix S/36 environments — which are probably your primary migration targets — have provided work-arounds for many assembler routines.

In fact, in many cases the problem solved by the assembler subroutine on the S/36 is solved natively and all you'll need to do with your code is remove or change the reference to the subroutine. For example, RPG/400 natively provides the ability to open and close printer files, to determine the cursor position, to manipulate strings, and to directly read and write source code members.

In other cases, alternatives to assembler routines can be coded in other high-level languages. For example, in 400RPG, ASNA provides a CKDT operation code that determines whether a date value is valid. In the 400RPG manual, ASNA provides the same routine ported to CL (the AS/400's Command Language) for use on the AS/400. This CL routine can be called just like the CKDT assembler routine and returns the same result. The Unix alternatives also provide many canned subroutine replacements and document the interface between their RPG compilers and native high-level languages such as C.

Don't shy away from assembler subroutines for your S/36 RPG if they provide a needed solution for you. But document their use well so that when

Figure 19.5a

Sample S/36E Program

```
H* AS/400 S/36 EE RPG II program
H     064
I           UDS
I                                            1   60VAL3
C                     Z-ADD12        VAL1    60
C                     Z-ADD26        VAL2    60
C                     Z-ADDØ         VAL3    60
C                     CALL 'PGM5B'
C                     PARM           VAL1
C                     PARM           VAL2
C                     PARM           VAL3
C* VAL3 should now equal 38
C                     SETON                          LR
```

Figure 19.5b

Native Program Called by Program in Figure 19.5a

```
C* AS/400 native RPG/400 program
C           *ENTRY    PLIST
C                     PARM           NUM1    60
C                     PARM           NUM2    60
C                     PARM           NUM3    60
C           NUM1      ADD  NUM2      NUM3
C                     RETRN
```

Figure 19.5c

Mapping Parameters with a Data Structure

```
C* AS/400 native RPG/400 program
IALPHA1     DS
I                                            1   60NUM1
IALPHA2     DS
I                                            1   60NUM2
IALPHA3     DS
I                                            1   60NUM3
C           *ENTRY    PLIST
C                     PARM           ALPHA1
C                     PARM           ALPHA2
C                     PARM           ALPHA3
C           NUM1      ADD  NUM2      NUM3
C                     RETRN
```

migration time comes, translating the subroutine is a planned activity, not a late-night emergency.

THE FINAL ANALYSIS

Even though the S/36E is improved from its early days on the AS/400, its compiler still offers the limited feature set of the RPG II compiler on the S/36. To take advantage of many of the advanced features and capabilities of the AS/400 and of RPG/400, you'll want to port many of your programs to the native environment quickly. Features such as new string manipulation operation codes, more informative file information data structures, and subfiles (the AS/400 technique of displaying scrolling lists) are only possible with RPG/400 and the native environment on the AS/400.

If you have ongoing program development on the S/36, and if you plan to someday migrate those programs to the AS/400, you have two choices:

1. Continue to code in straight RPG II (including EPC as provided by VASP) on your S/36. When you migrate to the AS/400, move to the S/36E first and then move one program at a time to the native environment to truly take advantage of the AS/400. If you follow this approach, you need to do little defensive programming on the S/36. Perhaps the best advice is to start getting your data file descriptions in order and to standardize file definitions from one program to the other.
2. Use 400RPG or RPG II1/2 on your S/36 now, which will give you many RPG/400-like features today. When you move to the AS/400, compile your programs using the RPG/400 compiler but run in the S/36E. The challenge of moving to the AS/400 will be a little tougher but your programs will be more maintainable and include more modern features and functions during their S/36 lifecycle. This strategy, assuming you do a little AS/400 homework, will also make the richer features of the AS/400's native compiler available to you sooner.

If you contemplate migrating to Unix-based S/36 environments such as Universal's Open RS/36 or ICS's Unibol, you will need to take fewer, if any, defensive migration steps on your S/36 if you stick with stock RPG II or with ASNA's 400RPG. Open RS/36 and Unibol both provide good compatibility with most of ASNA's extensions and their S/36 environment. At the time of this writing, RPG II1/2's more proprietary method of externally describing files may be a bit of a migration barrier to Open RS/36 or Unibol. However, third-party compatibilities are updated frequently, so check with the respective vendors for up-to-the-minute compatibility claims.

As you know by now, we strongly advocate using the RPG/400 features offered S/36 users by ASNA's 400RPG or BPS's RPG II1/2. If you're

willing to apply a little elbow grease, the functions and features they offer are well worthwhile. If you expect something for nothing, you're in for a migration migraine.

Into the Future

"I canna do anything with the engines, Captain."
"Well do something Scotty— we can't stay here forever!"

—*Star Trek*

It's true, as much as S/36 managers would like to think otherwise: We *can't* stay on the S/36 forever. Eventually continued operation and maintenance will just become too expensive to justify. But should you be making your migration decision today? Do the available S/36 replacements really have enough to offer you to pay the Pied Piper of migration or conversion?

While we can't foresee the future, we can give you advice about the present. This section gives you unbiased information about the two main migration options viable for most S/36 sites: AS/400 and RS/6000. Both arenas change monthly, though, so its virtually certain that some of the facts presented here will be out of date by the time you read them. You should read these two chapters with an eye toward understanding the fundamental differences between these wildly different operating systems. Chapter 20 is a kind of travelogue for S/36 users contemplating a move to the AS/400. You'll learn nitty-gritty details about life on the '400 in the S/36 environment and get an overview of AS/400 features. In Chapter 21 you'll see how Unix changes itself, chameleon-like, to appear more like a S/36 than even the AS/400. You'll also get a good assessment of major offerings for Unix migration to the RS/6000.

Nothing in this section will make the migration decision for you. But combining the facts you collect here with a watchful eye on new developments will at least help make your decision an informed one.

Chapter 20

The AS/400

For S/36 users, the AS/400 was once thought to be everything they had ever wished for. The allure of lickety-split performance, easy migration, and a familiar S/36 environment made the AS/400 seem too good to be true. Alas, as pioneering S/36 users who migrated to the AS/400 learned, early promises about the new system often *were* too good to be true. Many of the first machines were woefully under-configured and the AS/400's S/36 environment simply wasn't as fast or as compatible as it needed to be.

With the advent of the AS/400 models D and E, however, and with more appropriate sizing, the system's capabilities as a S/36 alternative are much improved; and price and performance will only continue to improve. We are still proponents of the S/36 and feel that if you have no compelling reason to migrate to the AS/400, don't; but some of you have — or will have — such reasons. In addition to its increased capacities (more memory, more disk, and more users), other valid reasons to migrate to the AS/400 include:

- Vastly broader and improved cooperative processing options. IBM, as well as the third-party market, have made PC connectivity to the AS/400 much better than it is on the S/36.
- Broader assortment of computer languages, much-improved (over the S/36) application development environment, and database and query support.
- Many more communications options, including non-IBM protocols such as TCP/IP and OSI.

However, if you want your AS/400 to be more than a pricey S/36, you'll do well to find out how the AS/400 carries off its feat of impersonation and what advanced AS/400 features you should consider learning first. And because one important feature of the AS/400 is its ability to coexist with existing S/36 software development, you'll want to understand how this coexistence works.

Three Audiences

IBM designed the AS/400 to serve the needs of three kinds of S/36 users. First are users who want to move to the AS/400 permanently. IBM's comprehensive migration tools simplify this move, and the S/36 environment on the AS/400 lets you run your S/36 applications as they are today so you can take your time learning about native AS/400 capabilities. As an alternative to migrating your

S/36 applications to the AS/400's S/36 environment, you can convert your applications to native AS/400 code using IBM's programming tools or third-party migration packages.

Second are users who must continue to live with an installed base of S/36s after the AS/400 arrives. IBM provides programming tools that let you shift software development to the AS/400 and distribution tools that let you easily maintain software at remote S/36 sites. AS/400 debugging and development tools will boost programmer productivity while still maintaining source code that compiles and runs on a S/36.

Third are users, such as software houses, who want to add the AS/400 as a hardware platform for application packages developed and sold primarily for the S/36. These users can continue to develop software on the S/36 and can use batch migration to maintain the current release of their products on the AS/400. Because IBM has retained the S/36, in the form of the AS/400 Model Y10, as its entry-level machine, many horizontal business applications such as accounting and payroll fall into this category.

All three user audiences depend on a common function within AS/400: the S/36 Environment (S/36E). A look at what the S/36E is and how it does its job can help you decide which user audience you want to join.

S/36E Layer

The S/36E is a superset of AS/400 capabilities. Not only does the S/36E support most S/36 facilities, but all OS/400 commands and objects are accessible at any time. Instead of emulating the S/36 at the cost of excluding AS/400 features, IBM built the S/36E as a layer on top of OS/400. An important benefit of this approach is that programs within the S/36E run nearly as quickly as jobs running in the native environment. The extra processing done by the primary component of the S/36E layer — the S/36 OCL Reader/Interpreter — has only a modest affect on performance.

The S/36 OCL Reader/Interpreter (RI) accepts operator commands, executes OCL statements and procedures, performs error and message handling, and invokes other components of the S/36E. Any OS/400 commands encountered by the RI are passed to OS/400 for processing. Thus, a S/36 programmer can intermix S/36 and OS/400 commands and program invocations within a S/36 procedure. (CL commands that don't call IBM or user-written programs, such as the structured commands IF or ELSE, aren't allowed in S/36 procedures because these commands can run only when they are compiled by the CL compiler.)

The RI carries out S/36 commands in one of three ways. The first approach uses specialized S/36E programs to perform tasks similar to S/36 commands and procedures — for example, the S/36 STATUS SESSION (D) command displays screens nearly identical to those displayed on the S/36. The

second approach simply maps S/36 commands and parameters to corresponding OS/400 commands that perform the same function. For example, the S/36 STATUS PRT (D P) command maps to the OS/400 Work with Spooled Files (WRKSPLF) command, which displays screens similar enough to S/36 screens that an operator can quickly adapt to the differences. The third approach deals with S/36 procedures and commands that the RI doesn't support directly, by redirecting the user to an appropriate OS/400 command. For example, the CHANGE PRT (G P) command displays instructions for moving one printout after another and then automatically invokes the OS/400 WRKSPLF command.

For the few S/36 functions that are meaningless on the AS/400, the RI performs syntax checking on the command or OCL statements (to ensure compatibility with the S/36) but takes no action. For example, the AS/400 architecture doesn't require contiguous disk space for files and libraries, so $FREE utility control statements are simply checked for syntax and ignored.

In the S/36E, the RI displays messages in the same format the S/36 uses, with options 0 through 3 and H for reprompting. Most messages have the same message identification codes (MICs) and auto-response values as their S/36 counterparts. You still can set auto-response values for your own application-driven messages, but not for system messages — a minor restriction easily offset by the additional flexibility OS/400 provides in message handling. For example, the RI improves on S/36 message handling by putting the entire contents of the S/36 message manual at your fingertips whenever an operator message is issued. Impromptu messages sent via the S/36 MSG command and // MSG OCL statement are presented in the same way as on the S/36: A message light cues the user that messages are waiting to be displayed. However, you can optionally take advantage of OS/400's break message delivery, which interrupts the current application to display messages as soon as they are sent. Similarly, you can use the AS/400's decentralized system console facility to view and respond to console messages from any workstation with proper authorization — definitely an improvement over the S/36 system console bottleneck.

Smoke and Mirrors

OS/400 objects support familiar S/36 objects such as #LIBRARY and menus. To appreciate where S/36 files and libraries fit into the AS/400 scheme, you must first understand a few important differences between OS/400 and SSP. The first difference is that libraries contain all objects (e.g., files, programs, folders, documents) on the AS/400. Contrast this with the S/36, where libraries contain only library members, folders contain documents, and files and folders are separate entities from libraries. The difference is important because it is the key to OS/400's object access scheme. The second difference is that database files can have multiple members, just as S/36 libraries have multiple members. AS/400 file members, however, look just like regular data files with fixed-length records.

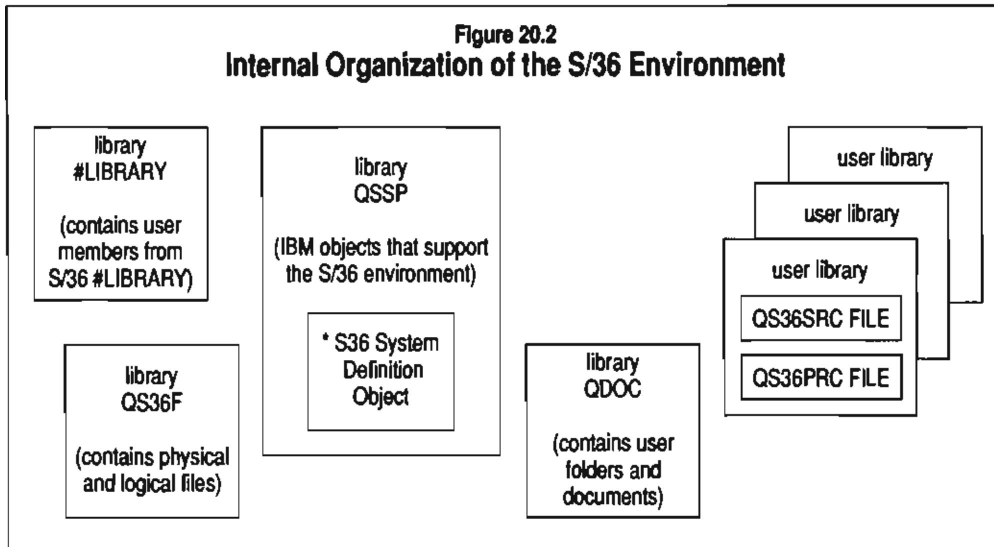
Figure 20.1
How S/36 Objects Relate to Their AS/400 Equivalents

System/36		AS/400	
Object	Inside a...	Object	Inside a...
library #LIBRARY	—	libraries #LIBRARY and QSSP	—
Data file	—	Physical file	library QS36F
Alternate index	—	Logical file	library QS36F
Library	—	Library	—
Load member...			
Compiled program	library	Program	library
Screen format	library	Display file	library
Message member	library	Message file	library
Subroutine member	library	Program	library
Source member	library	Source file member	source file QS36SRC
Procedure member	library	Source file member	source file QS36PRC
Virtual disk	—	Shared folder	library
Folder	—	Folder	library QDOC
Documents	folder	Documents	library QDOC
Data dictionary	folder	Data dictionary and a set of files	library

With those two differences in mind, look at the table of equivalent S/36 and OS/400 objects in Figure 20.1. Notice that source and procedure members are stored as members of the files QS36SRC and QS36PRC, respectively. Source file members are not stored in compressed form, as they are on the S/36. Instead, they are stored as fixed-length records in database file members. While this takes up more storage than the S/36 approach, you can read and write AS/400 source file members directly in any high-level language (HLL) program — a tremendous advantage over the S/36 $MAINT utility support. Also notice that all S/36 data files and alternate indices are kept in library QS36F. This approach ensures that all S/36 programs have global access to all S/36 files — an important feature for maintaining S/36 compatibility.

Figure 20.2 shows the organization of the OS/400 objects that make up the S/36 execution environment. The contents of the S/36's #LIBRARY —- IBM and user programs — are split between two OS/400 libraries. The OS/400 #LIBRARY contains only user programs and procedures from the S/36 #LIBRARY, while a new library, QSSP, contains the IBM-supplied programs that control the S/36E. Many S/36 users provide global access to certain applications by depending on the S/36 to search #LIBRARY after searching the current user library. Unfortunately, installing SSP updates often destroys or interferes with

Figure 20.2
Internal Organization of the S/36 Environment



user members in #LIBRARY unless the user takes care to remove these members first. By separating IBM-supplied objects from user objects, the AS/400 lets you have your cake and eat it too: #LIBRARY is still searched after the current user library, but IBM can update QSSP by replacing it without affecting user members.

Building and Controlling the S/36E

The AS/400 automatically creates the necessary objects to support the S/36E when you start the S/36 subsystem for the first time. One important object that describes the users and configuration of the S/36E is the S/36E definition object, which has an object type of *S36. OS/400 initially creates the *S36 object with default values that satisfy the needs of most S/36 users. However, you can modify the default values at any time to change the behavior of your S/36E. And when you start the S/36 subsystem, you can optionally use one of a number of *S36 environment definition objects you've created to support several different S/36Es.

When the AS/400 automatically creates the S/36E definition object, it adds the set of hardware devices currently configured for the AS/400 native system to the device configuration for the S/36E. When new devices are added to the AS/400 configuration (by simply attaching the devices and turning them on), the *S36 object is updated automatically, letting you begin using new terminals and printers immediately, without a manual reconfigure and IPL.

The *S36 object contains a copy of the AS/400 hardware configuration for two reasons. One is to support S/36 individual session parameters

(e.g., default menu and local printer ID) that don't exist in AS/400 user profiles — these S/36 parameters are stored by workstation ID, not by user ID. The other reason is to automatically translate 10-character AS/400 device names into two-character S/36 device IDs. S/36 applications that use the OCL substitution parameter ?WS? or the RPG F-spec extension keyword KWSID expect a two-character ID, not a 10-character name. For example, an application might create a unique temporary file using the construction ?WS?TEMPFL, which results in an eight-character file name if ?WS? is only two characters long. A 10-character value for ?WS? would yield a 16-character name, and the application would fail when it tried to use the large name in an OCL statement.

The Programming Environment

As a programmer, you'll find quite a few changes between S/36 and AS/400 program development environments. Most of these changes are improvements. For example, SEU on the AS/400 is a program editor nearly identical to DSU on the S/36. SEU, however, adds interactive prompting and diagnosis for CL commands. It also lets you split the screen to edit two source members simultaneously or to edit a source member while viewing a compile listing directly from the spool file (without first performing a COPYPRT). These two features address two common bottlenecks in on-line coding: the need to look up command language formats in manuals and the need to handle paper listings during program development.

The AS/400 equivalent of the S/36 Programmer and Operator Productivity Aid (POP) is the Program Development Manager (PDM). PDM presents on the screen a list of objects from a library. Next to each object is a command field that you use to perform AS/400 operations on the object. PDM also folds the object text feature from the S/38 into the development environment. Object text is a one-line comment attached to every object in the system. However, on the S/38, the object text lines are difficult to maintain and inconvenient to view. PDM eliminates these S/38 troubles by displaying the object text with each object name on the screen, which is one of the reasons PDM displays are single-column. While POP purists may be put off initially by PDM's lack of multicolumn displays, the many improvements on POP's facilities will quickly win purists over.

Users can customize PDM just as they can customize POP. PDM's two-character mnemonic operation codes, in contrast to POP's single-character codes, greatly expand the number of mnemonic operations. And PDM lets each programmer have a private collection of customized PDM operations in addition to the global set available to all users.

Free Usability and Performance Improvements

Applications that run in the S/36E are free from a number of restrictions

imposed by the S/36. For example, the maximum number of open files increases to 50 for RPG programs, and the limit on arrays has been removed. The 64 K program size limitation also disappears — AS/400 programs can contain up to a megabyte in the instruction stream alone, not counting program variables. Programs that run in the S/36E also can take advantage of AS/400 database support such as data integrity checking, logical views over multiple physical files, and recovery functions such as journaling and commitment control.

Compatibility with the S/36

One of IBM's goals with OS/400 is to provide a platform for developing source code that runs on both the AS/400 (in the S/36E) and on the S/36. RPG/400, the RPG compiler for all OS/400 environments, supports a S/36 compatibility option that lets you specify that a particular program will be run on a S/36.

RPG/400 issues warning messages when compiling such programs if they violate S/36 constraints, such as file or array limits. These programs are likely to compile and run successfully on the S/36 if you pay attention to the warning messages. However, the S/36 compatibility option is not an iron-clad guarantee that your programs will run on the S/36, even if no warning messages appear. A few S/36 constraints, such as the 64 K program size limit and alternate index key limitation, can be checked only by compiling the program on a S/36. Nevertheless, the RPG/400 compatibility option, combined with extensive syntax checking provided by the RI for S/36-only OCL and commands, provides enough coexistence support to make the AS/400 a good software development platform for S/36 hardware.

A few significant RPG/400 operation codes that the S/36 compatibility option lets pass as S/36 compatible are CALL, PARM, RETRN, and PLIST — the operations that support external program calls. Because S/36 SSP Release 6.0 and several non-IBM products provide external program call capability on the S/36, IBM apparently thought it prudent not to flag such programs as incompatible.

Sizing Up Your AS/400

Once you've weighed the pros and cons and decided to move up to an AS/400, dozens of questions arise. Which AS/400 model will meet your current needs but still allow growth? How much disk and memory is necessary to fit your existing applications and data files? What resources do such OS/400-unique products as SQL/400 and OfficeVision require?

Welcome to the wonderful world of system sizing. Come up with wrong answers to one or more of the above questions and you buy yourself a pack of trouble. For example, selecting the wrong one of the AS/400's three families might leave your organization with no CPU growth path. Purchasing too little memory could result in a system much slower than the S/36 to which

your users are accustomed. Purchasing too much disk wastes money that might be better spent on additional software or peripherals. Fortunately, others have trod this way before you, resulting in a treasure trove of information about accurately sizing your new AS/400. Follow a few basic guidelines (derived from the experience of hundreds of S/36-to-AS/400 migrations) and you'll stay on the narrow path, out of the thickets of disaster.

Sizing an AS/400 consists of three steps: first, evaluating your information systems' current and future requirements; second, selecting an AS/400 model that meets your current needs but accommodates anticipated growth; and third, configuring appropriate amounts of disk and memory to achieve performance equal to your S/36's performance.

To evaluate your current information system requirements, you'll need to determine the size of your existing data files and libraries, the average number of active users, and the kind of work each user does. You can classify user work as transaction- or word processing-oriented. For example, you would classify an order entry clerk as transaction-oriented, and a DW/36 user as word processing-oriented. Some users may fall into both categories, in which case you should count them as two users — especially if these users work with two interactive jobs running simultaneously, or run batch jobs in the background while doing interactive work.

Evaluating future information system requirements requires that you look at the historical growth of your data files and number of users and determine if that trend will continue, increase, or slow down. With your current and future system requirements in hand, you're ready to select an AS/400 model to accommodate those requirements.

Model Selection

You can make an initial model selection based on the model of your current S/36, and then modify that selection to match other considerations. Figure 20.3 shows corresponding S/36 and AS/400 models based on IBM recommendations and reports from customers. These represent starting points for your selection, based on whether you need a *growth configuration* to accommodate future expansion or a *static configuration* for unchanging "turnkey" operations.

For growth configurations, you must also factor into your model decision the growth capabilities of the three AS/400 hardware families. IBM offers field upgrades within a family, but to cross family boundaries you must purchase an entirely new CPU. The low-end (9402) family consists of the E0x models — E02, E04, and E06; the mid-size (9404) family covers the range E10 through E25; the high-end family (9406) is the E35 and above. If you find your initial selection in the upper end of a family (i.e., E06 or E25), you should consider choosing a model from the next-higher family. Keep in mind that there is some overlap between the mid-size and high-end families: An E25 CPU is

Figure 20.3
Equivalent S/36 and AS/400 Models

S/36 Model	AS/400 Model: Growth	AS/400 Model: Static
5364 single user	9402 E02	9402 E02
5364 multiuser	9402 E04	9402 E02
5363	9404 E10	9402 E04
AS/Entry (Y10)	9404 E20	9404 E10
5362	9404 E20	9404 E10
5360-C	9404 E25	9404 E20
5360-D	9406 E45	9404 E25

Figure 20.4
AS/400 Memory Requirements Worksheet

AS/400 Model	Base Memory
E02 – E20	8 MB
E25 – E45	16 MB
E50	32 MB
E60 – E90	64 MB

Additional Memory

Models 9402/9404

Native Users x .3 MB or S/36E Users x .5 MB
+ Office Users x .8 MB = Total Additional Memory

or

Model 9406

Native Users x .4 MB or S/36EE Users x .7 MB
+ Office Users x 1.0 MB = Total Additional Memory

Total Memory = __________ (Base Memory) + __________ (Total Additional Memory) = __________ **MB**

30 percent faster than an E35. You should probably choose an E45 if future growth will push you out of the mid-size family.

Configuring Memory

While each AS/400 ships with a certain amount of memory — called "base" memory — already installed, chances are that you'll need to add more. Figure

Figure 20.5
AS/400 Model Capacity Chart (MB)

CPU model	Base Configuration Memory	Base Configuration Disk	Maximum Configuration Memory	Maximum Configuration Disk
E02	8	800	16	2,000
E04	8	800	16	4,000
E06	8	800	20	4,000
E10	8	800	32	11,900
E20	8	800	40	11,900
E25	16	800	64	15,800
E35	8	1,280	72	28,700
E45	16	1,280	80	28,700
E50	32	1,280	128	49,200
E60	64	1,280	192	76,700
E70	64	1,280	256	76,700
E80	64	1,280	384	124,700
E90	64	1,280	512	124,700

20.4 depicts a worksheet for estimating AS/400 memory requirements. Use the table at the top of the worksheet to find the base memory for the model you are considering.

Use the second part of the worksheet to determine the per-user memory value for the 9402/9404 family or the 9406 family. Add base memory installed and total additional memory to find your estimated total memory requirements.

The worksheet's "native users" category is useful only if you plan to convert your applications to native mode (using application redesign) rather than run in the S/36E. If you migrate rather than convert, your users probably fall into the "S/36E" category. Existing DW/36 users, or Office/400 users if you're planning to use office automation for the first time, count as "Office" users.

Note that the high-end family has larger per-user memory requirements than the low-end families, indicated by increased per-user values on the chart. Be sure to use values from the column corresponding to your model number.

You should use this worksheet twice: once to compute your immediate memory requirements, and a second time to compute your future memory requirements. Then check the AS/400 capacity chart (Figure 20.5) to see if your chosen model can accommodate your future memory requirements. If it can't, you should step up to a larger model.

Configuring Disk

The AS/400 uses disk storage much more liberally than does the S/36. The operating system and program product software alone occupy from 200 MB to

Figure 20.6

AS/400 Disk Requirements Worksheet

1. Reserve for OS/400 and IBM program products	________	MB
2. Data files (1.2 x S/36 usage)	________	MB
3. Source code (3.6 x S/36 usage)	________	MB
4. Object code (7.0 x S/36 usage)	________	MB
5. Folder contents (2.0 x S/36 usage)	________	MB
6. Mail logs (1.0 x S/36 usage)	________	MB
7. Data dictionaries (1.0 x S/36 usage)	________	MB
Total disk storage required:	________	**MB**
at 60% utilization:	**+ 0.6**	
Total disk storage to configure:	________	**MB**

Figure 20.7

IBM Program Product Space Requirements

Product	Space (MB)	Product	Space (MB)
9402/9404 Base System	186.0	Pascal	4.7
9406 Base System	204.0	C/400	9.5
QGPL & QUSRSYS Libraries	15.0	Application Development Tools	15.0
S/36 Environment	12.0	Performance Tools	8.0
Help Text and Tutorials	10.6	SQL/400	1.0
S/36 Migration Aid	6.5	Communications	3.6
RPG/36 & RPG/400	4.1	OfficeVision/400	17.0
COBOL/36 & COBOL/400	6.4	Query/400	2.5
BASIC	2.5	PC Support	20.0
PL/1	2.1	Online Information	22.0
FORTRAN	2.0	Online Education	5.0

500 MB! You must take care as you determine AS/400 disk requirements to avoid running out of disk space before you even get started.

Figure 20.6 is a worksheet for estimating AS/400 disk requirements. The first item represents disk overhead for OS/400, utilities, compilers and

other IBM software products. Determining the value for this entry is simple: Just add up the allowances given in Figure 20.7 for the IBM software you'll be installing. First select the base system value of 186 MB or 204 MB, depending on the model family, then add in other allowances as required.

The second worksheet item reserves disk space for your data files — which take up 20 percent more space on the AS/400 than they did on the S/36. Similarly, source code and object code on the AS/400 require more space, by factors of 3.6 and 7.0, respectively (items 3 and 4). OS/400 uses ordinary fixed-length data files for source libraries, rather than compressing text as the S/36 does. AS/400 object programs are larger to include debugging information, although you can cut this overhead in half by *removing observability*, a process which deletes debugging tables from program objects. Finally, DW/36 document folders require twice as much room on the AS/400 (item 5), but Office/36 mail logs and data dictionaries retain their original size (items 6 and 7).

Totaling the worksheet yields the amount of disk storage you'll need to get up and running on the AS/400. However, using this figure as-is would leave you with no free disk space. IBM recommends an initial disk utilization of 60 percent, to allow room for OS/400 working areas and short-term growth. Dividing your disk total by .6 meets that goal. As with memory calculations, you should run through the worksheet a second time to determine future disk requirements. Then check the capacity chart (Figure 20.5) to verify that your chosen model is sufficiently upgradable.

Making Sure

The rules of thumb presented in these worksheets will accurately predict model, disk, and memory requirements for typical S/36 migrations. But the operative word here is "typical." If your S/36 installation has any of the following unique requirements, you might need to do further research:

- Minimum acceptable interactive response times
- Heavy batch processing along side interactive use
- High system availability

To ensure a particular response time, you'll need to perform a detailed analysis of user workload, transaction composition, and program organization. Heavy batch processing requires special OS/400 configuration parameters and additional memory. High availability (i.e., minimizing downtime) may require AS/400 reliability features such as journaling, checksum protection, disk mirroring, or RAID (Redundant Array of Inexpensive Disks) technology, which increase disk requirements.

IBM can help you research these advanced requirements using the AS/400 MDLSYS (Model System) tool, a sophisticated performance simulation

model available through your local branch office. To use MDLSYS, you specify your proposed system configuration, transaction volumes and rates, transaction complexity, and performance objectives. MDLSYS uses these specifications to run a simulation predicting response time, disk performance, and transaction throughput, and even makes upgrade recommendations. Once you've installed your AS/400, you can run MDLSYS yourself to project future hardware requirements and performance if you purchase OS/400's performance measurement tools.

Using the guidelines presented here, you can configure an AS/400 with good performance and sufficient expandability without overspending. All that remains is the move itself.

All in All

For S/36 users, these are times that offer an unusual number of choices. Carefully consider the AS/400 as your migration platform of choice, but don't discount that Unix/AIX S/36 work-alike environments. Remember, too that the AS/400 is approaching the downhill side of its lifecycle curve.

For many S/36 users, though, the AS/400 may well be an offer you can't refuse. Relatively easy migration, improved programmer productivity and a more unified development platform, and coexistence with existing S/36 applications all add up to an attractive migration alternative.

Chapter 21

The Unix Alternatives

By reporting hands-on experience with each of the two Unix-based systems (UNIBOL and Open RS/36), this chapter gives you a solid overview of what each product's capabilities and limitations are, and why each might be a viable migration path from the S/36.

UNIBOL ON THE RS/6000: A VIRTUAL S/36

Software Ireland's Unix-based S/36 emulation product, UNIBOL, was first released in 1987. It has matured much since then into a comprehensive clone of the S/36 environment, and is offered for more than 18 Unix platforms, including AT&T, TI Honeywell, HP, Data General, NCR, the RS/6000, and even PCs running SCO Unix. UNIBOL currently has more than 500 U.S. installations, with more than half of them using UNIBOL on the RS/6000. UNIBOL is marketed for the RS/6000 and PC-based SCO Unix platforms by the Atlanta-based UNIBOL, Inc.

Functions and Features

UNIBOL's comprehensive emulation of the S/36 environment includes most major functions and facilities you'd expect. Among the major features of UNIBOL:

Disk Data Management. UNIBOL fully emulates the S/36's DDM facilities. A B-tree facility provides keyed record access, replicating the S/36 indexed sequential access method (ISAM) — but eliminating its needs for the overflow index area and KEYSORTs. Disk-file data retains its EBCDIC format under UNIBOL on the RS/6000, providing full support for packed and binary fields, as well as complete compatibility with the S/36 collating sequence for correct sorting. After migration, files also retain attributes such as share disposition, delete-capable status, and retain status.

Workstation Data Management. UNIBOL's S/36 Workstation Data Management emulation supports all screen attributes, including reverse image, high intensity, and blink. Because of a feature specific to AIX, the RS/6000's implementation of Unix, UNIBOL currently provides S/36-style multiple-session support only on the RS/6000. When running UNIBOL with other Unix implementations, multiple sessions are implemented through a third-party windowing tool. UNIBOL, Inc., is currently evaluating the best way to offer consistent multiple-session support across all Unix platforms.

OCL and Utility-Control Commands. UNIBOL replicates the S/36 OCL command processor, including support for conditional and substitutional expressions. UNIBOL also supports most SSP operator-control commands,

including CANCEL, CHANGE, MENU, MSG, START, and STATUS. However, UNIBOL doesn't support some OCL expressions and statements simply because they're unnecessary. For example, UNIBOL provides dynamic file sizing with UNIBOL, eliminating the need to explicitly size a file using the OCL FILE statement RECORDS parameter. To migrate code developed with UNIBOL on the RS/6000 to the S/36, UNIBOL ignores the RECORDS parameter rather than issuing a message or causing an error.

Additional Features. Additional S/36-supported features include JOBQ and Evoke support, history logging, inquiry mode, console and subconsole emulation, a S/36-compatible print spooler, a fully compatible implementation of #GSORT, a POP replacement, RPG and COBOL languages with SRT/MRT/NEP support, source-level debugging, and support for S/36 utilities such as DFU, COPYPRT, BLDINDEX, and SDA. See Figures 21.1 and 21.2 for more information on UNIBOL's feature and facility support.

Missing Links

Although UNIBOL supports a substantial portion of the S/36 feature set, some users may miss a few important features — such as security and communications. UNIBOL does provide resource and password security, but not as an emulated S/36 service. Rather, security is performed directly under Unix, requiring you to be familiar with Unix to implement and maintain it. As for communications, remote workstation support is a simple matter of using a PC that emulates a 3151 ASCII workstation. But for more advanced communications features (APPC, APPN, SNA, and BSC), UNIBOL leaves you in the lurch. UNIBOL, Inc., plans support for such communications features in future releases, but that support isn't available now. Company officials also say IDDU and other office products will be supported in the future.

UNIBOL emulates the S/36 SSP as an additional operating-system layer running on top of Unix. It's easy to use UNIBOL without knowing any Unix, but system operators or administrators will need at least a passing familiarity with Unix. In addition to resource security, UNIBOL requires direct Unix access for system backup.

UNIBOL doesn't directly support 5250 devices, which means replacing all your 5250 devices with ASCII devices and learning the ergonomics of those devices. For cost and training reasons, some S/36 users would like the option of attaching their 5250 devices directly to the RS/6000. Direct attachment, however, is an option available on UNIBOL only through third-party ASCII-to-EBCDIC converters.

S/36 Look and Feel

UNIBOL provides a very capable implementation of the S/36 operator environment. It closely emulates S/36 menus, operator commands, and utilities. In

Figure 21.1

Comparison of UNIBOL and Open RS/36 Features

Feature	Open RS/36	UNIBOL
Object code compatibility	●	▲
Converts RPG to C	●	●
Converts COBOL to C	■	▲
Converts OCL to C	▲	▲
Runs S/36 native RPG compiler	●	▲
Runs S/36 native COBOL compiler	●	▲
Runs S/36 native utilities	● 1	▲ 13
Diskette data interchange	▲	▲
Tape data interchange	● 2	●
Twinaxial data interchange	●	● 14
Data files retain EBCDIC characteristics	●	●
OCL interpreter	●	●
RPG III language features	● 3, 4, 5	● 17
Full-screen editor	● 6	■
Source-level debugger	●	●
X-Windows support	●	●
Relational database support	■	●
Access to native Unix facilities	● 7	● 7
Unix access to S/36 data files	● 8	● 16
ASCII terminal support	●	●
Twinaxial terminal support	●	● 14, 18
Ethernet connectivity support	●	●
X-Windows terminal support	●	●
Multiple sessions per terminal	● 9	● 14
Common S/36 assembler routines	● 10, 11, 12	● 12, 15

● supported ■ planned ▲ unsupported

Notes:

1. DFU, SEU, SDA.
2. SAVELIBR, SAVE, and FROMLIBR formats.
3. In COBOL and RPG load members (ASNA-, BPS-, and IBM-compatible).
4. RPG-to-C translator supports ASNA, BPS, and IBM EPC syntax.
5. Other RPG III feature support is identical to BPS RPG 2 1/2.
6. FSEDIT-compatible, RPG-sensitive, multilevel undo/redo, cut/paste buffers, multimember editing, user macros.
7. Native Unix utilities are accessible from OCL procedures. RPG and COBOL may CALL native C programs.
8. Shared access through direct calls to S/36 database server; unshared access through C interface routines.
9. Up to 7 S/36 sessions per terminal, with 16 logical screens per session.
10. Many S/36 assembler routines will run "as is." Assembly language macros supported: $ALOC, $CLOS, $EOJ, $GETD, $PUTD, $PUTP, $LOG, $INFO, $OPEN, $RIT, $SIT, $TOD, $WAIT, $WSIO, fixed-disk I/O, and direct #GSORT access.
11. RPG assembly language routines available for library directory get/put, library member get/put, JOBQ and EVOKE, LDA/UPSI access, and program wait state.
12. Standard IBM assembler routines SUBR01, SUBR20, SUBR21, SUBR23, and SUBR95.
13. Services provided by UNIBOL.
14. Supports 15 sessions per terminal on the RS/6000; requires third-party products for multiple session support under other Unix versions.
15. Return cursor position on screen, open and close files.
16. Via supplied C interface.
17. Supports all ASNA and BPS extensions, including ASNA's CALL PROCEDURE.
18. Although UNIBOL doesn't directly provide service, twinaxial support is available through third-party products.

fact, if not for the RS/6000's increased speed over the S/36, UNIBOL could easily fool a casual user into thinking it is a S/36.

Besides the machine's sheer speed, other little clues — such as different message numbers and text — confirm you aren't really on a S/36. Another hint is that some utility prompts don't provide all S/36-supported parameters. But then, not all are necessary. BLDFILE, for example, doesn't prompt for the size of the file to create because UNIBOL manages that dynamically on the RS/6000. For S/36 users — who would certainly be comfortable using UNIBOL almost immediately — UNIBOL well represents the look and feel of the S/36.

Migration Patterns

For S/36 users migrating to the RS/6000, UNIBOL, Inc., offers an optional $800 Migration Toolkit, a menu-driven migration aid running on a S/36 and attached RS/6000. For migration, you generally attach the RS/6000 directly to a S/36 via an Andrew Alliance board in the RS/6000, although you can also attach the systems via an Ethernet or Token-Ring connection. To ease migration headaches, the company also offers its customers migration assistance. In addition, UNIBOL, Inc., is developing a way to use tape drives as the exchange medium and expects to implement a tape-drive migration option soon.

Migrating a test suite of programs (the same ones used to test Open RS/36) was a simple point-and-shoot affair using the Migration Toolkit. Unlike Universal Software's Open RS/36, UNIBOL isn't load-member compatible. Therefore, load members aren't migrated and must be recompiled on the RS/6000 using UNIBOL's compilers. The Migration Toolkit automates the compiling process and re-creates all necessary load members, including RPG and COBOL members, message members, and screen formats. In addition, the migration utility scans all source and procedure members looking for unsupported features, which it describes in a migration report.

Passing Muster

UNIBOL compiled a test suite of RPG programs very quickly — so quickly it was almost unbelievable. A 900-line program compiled in five seconds. That's not a typo — five seconds! Although the suite wasn't exhaustive, it did include batch processing, using #GSORT in several ways, some complex and conditional OCL, interactive RPG programs (some including external program calls and other ASNA-specific features, with which UNIBOL is compatible), and a couple of RPG file-update programs. UNIBOL compiled all the test programs and ran them without a hitch — quickly producing accurate results. Early tests revealed problems with an interactive program that directly manipulated the 5250 data stream, but subsequent tests revealed that glitch has been cleared up.

Software Ireland and UNIBOL, Inc., say UNIBOL has undergone rigorous testing over the years, stressing that what UNIBOL supports, it supports

Figure 21.2

Comparison of UNIBOL and Open RS/36 Facilities

S/36 Facility	Open RS/36	UNIBOL
RPG language	●	●
COBOL language	●	●
BASIC language	▲	▲
External program calls	● 1	● 1
Externally described files	● 2	● 2
OCL, menus	●	●
Evoke, job queue, spooling	●	●
Password, resource security	● 3	■ 3, 15
Base SSP utilities (e.g., $COPY, $LABEL)	● 4	●
Optional SSP utilities (e.g., DFU, GSORT)	●	●
Screen-format generation ($SFGR, SDA)	● 5, 6	●
Low-level workstation support (screen orders)	●	●
FSEDIT equivalent (full-screen editor)	●	■
POP equivalent	●	●
Procedure prompting	●	●
Procedure control expressions	●	●
Diskette OCL notation (S1-S3, M1.xx, M2.xx)	● 7	●
Tape OCL notation (TC, T1, T2)	● 7	●
Operator control commands	● 8	● 13
SSP utility control statements	● 9	● 9
S/36 message handling	● 10	●
System-level help	● 11	●
Application-level help	● 12	● 14
IDDU	■	▲
Query	● 16	● 16
Other office products	▲	■
Communications (e.g., BSC, SNA, ICF, APPC, APPN)	▲	■

● supported ■ planned ▲ unsupported

Notes:

1. Compatible with ASNA, BPS, and IBM SSP Release 6.0 EPC, including ASNA CALL-procedure support.
2. External descriptions for files, workstations, and printers, and associated RPG III opcodes (UPDAT, WRITE, EXFMT).
3. Standard S/36 password security; Unix resource security.
4. Includes diagnostic support for "nonfunctional" utilities such as $DDST and $FREE.
5. Enhanced $SFGR allows cursor movement in D-spec instead of row/column order. This function is portable back to S/36 hardware via screen format load member.
6. IBM's S/36 SDA product runs. SDA/400-compatible screen design aid under development.
7. Supported by $MAINT and $COPY; support by other utilities planned.
8. Except ASSIGN, MODE, POWER, and VARY.
9. For example, // COPY for $MAINT and // COPYFILE for $COPY.
10. Message descriptions and responses correspond to IBM messages.
11. User menus and procedures supported by HELP command and $HELP utility.
12. Application S- and D-spec help supported.
13. Except ASSIGN, CONSOLE, POWER, PRTY, REPLY, and VARY.
14. Limited implementation via H-specs.
15. Security service currently provided only at the Unix level.
16. Both vendors have third-party query products available.

with integrity and reliability. Software Ireland uses a validation suite of hundreds of programs, including testing programs and many production-quality application modules, that takes fives days to run. UNIBOL programmers have completely automated the test suite's interactive portions, which are performed at strategic intervals during program testing to ensure new features haven't introduced new bugs. Beware, though, that while exhaustive testing can prove the existence of bugs, it doesn't prove the absence of bugs. The S/36 has had almost 10 years' experience and countless programs run on it to prove its reliability. If you migrate to a S/36 work-alike environment that claims source-code compatibility, don't assume anything. Migrate defensively and test everything carefully and explicitly.

User Recommended

UNIBOL is a mature product with many users who generally speak highly of its quality and S/36 compatibility. UNIBOL customers — referred by UNIBOL, Inc. — gave the product high marks in interviews about their migration and support experiences. One user said he had migrated more than 2,000 interactive programs without problem, and another reported migrating more than 2,800 sorts and 5,000 programs, trouble free.

When compared to a dedicated model B23 S/36, a dedicated RS/6000 proved to be anywhere from four to six times as fast in the test suite results. Some UNIBOL users even reported speeds of up to 12 times as fast, but performance figures vary widely, depending on the test (see Figure 21.3). For example, the S/36 is notoriously slow at adding records out of key order to a shared, indexed file. Test 2 in Figure 21.3 shows UNIBOL's timing advantage over the S/36 running one copy of this test on the dedicated systems. As proof of the superiority of UNIBOL's B-Tree keyed-access method over the S/36's ISAM, consider the following example. When three copies of Test 2 were evoked, UNIBOL completed the job in two minutes, 16 seconds, while the S/36 plodded along for almost six hours! This test exploits a S/36 handicap and shouldn't be used as a general rule of thumb; but again, it shows UNIBOL's speed advantage over the S/36. Although some evoked jobs were used to test concurrent performance, such simple tests don't really show how UNIBOL and the RS/6000 perform in heavily used, multiuser environments. UNIBOL users, however, have reported satisfaction with the product's multiple-user performance.

Programming with UNIBOL

The UNIBOL programming environment is a bit of a disappointment. To a user, UNIBOL presents itself well as a virtual S/36 and could fool many S/36 operators if they tried to identify which machine was at their fingertips. But UNIBOL couldn't fool a programmer for long. UNIBOL doesn't provide a

Figure 21.3
UNIBOL Versus S/36 Performance Benchmarks

Test		UNIBOL	S/36
1	Create 10,000-record indexed file (Records added in key order)	:31	3:48
2	Add 10,000 records to an existing, empty indexed file (Records added in reverse key sequence)	:57	47:07
3	Read 30,000 records from an indexed file	:37	2:24
4	#GSORT 30,000-record, 256-byte record length file (Tagalong sort)	1:26	4:17

- All tests performed with a file with 256-byte record length and a 16-byte key.
- UNIBOL tests performed on a dedicated model 320 27.5 mips RS/6000 with 16 MB of memory and two 320 MB IBM drives.
- S/36 tests performed on a dedicated B23 S/36 with 768 K of memory.
- Because of the tests' random nature, blocking wasn't used with the S/36 tests.
- All tests were performed with shared files on both platforms.

direct POP replacement but rather a Programmer Interface Environment (PIE) — a cross between the S/36's POP and the AS/400's PDM (leaning heavily on the PDM side), with a pinch of Unix thrown in. Figure 21.4 shows a screen of PIE's POP FILE function equivalent. Note the Unix file structure reference in the upper left corner (/usr/s36/news3x). While it doesn't do everything like POP does it, UNIBOL's PIE offers most library and file features that POP does.

PIE's SEU editor is particularly frustrating. A virtual clone of the old, line-oriented SEU that was state of the art on the S/36 (circa 1983), UNIBOL's SEU doesn't offer full-screen editing in the S/36 environment. But even worse, SEU (tested under UNIBOL's release 4.0) contains a few bugs that can bring it crashing down around unsuspecting users. More than once, an intermittent error suddenly ended an editing session — a couple of times taking code with it. When pressed about editor quality, the UNIBOL, Inc., staff suggested a user could toggle over to the Unix side and use VI, an old-fashioned, Unix full-screen editor, or use FEU, their customized version of VI integrated into PIE. For experienced Unix users, that might be valid advice; but to S/36 users looking for a POP replacement, it's not a reasonable alternative. For a system that so faithfully replicates the S/36 environment, UNIBOL desperately needs a reliable, full-screen editor in its PIE environment.

Despite the lack of FSEDIT and occasional keyboard lapses (PIE sometimes insists on CMD-3 where the S/36 uses CMD-7), you'll quickly — probably within 30 minutes — be competent with PIE's basic functions. Compiler support is integrated into PIE, as it is in POP or PDM, and simply putting a 14 next to the source code name and pressing Enter compiles the program — and compiles it in a hurry. Dialing into an RS/6000 running UNIBOL from a

Figure 21.4

Screen from UNIBOL's Programmer Interface Environment (PIE)

```
  Objects *ALL            Type *MEM        Total 76         Sort Name
Directory  /usr/s36/news3x
  Library TEST3X
  Command
Cmd Keys  3:Exit   5:Refresh  7:Uplevel  10:Log Menu   11:Show Log  12:Select
         13:Sizes 14:Status  15:Basic    16:Sort Size  17:Sort Name 24:More
Options: 2-Seu      3-Copy      4-Delete     5-Display    6-Print      7-Rename
         8-Status  10-FEU/vi   14-Compile   16-Run        17-SDA
 Opt Object        Type     Opt Object        Type       Opt Object        Type
 ___ ALDMSG        MSG      ___ DODATE        OCL        ___ HEX2          RPG
 ___ ALSMSG        MLM      ___ DODATE        RLM        ___ HEXTST1       RLM
 ___ ARR1          RLM      ___ DODATE        RPG        ___ HEXTST1       RPG
 ___ ARR1          RPG      ___ DOW           RLM        ___ HEXTST2       RLM
 ___ CAL1          RLM      ___ DOW           RPG        ___ HEXTST2       RPG
 ___ CAL1          RPG      ___ DOW1          OCL        ___ HEXTST3       RLM
 ___ CAL1FM-OLD    SSM      ___ DOW2          OCL        ___ HEXTST3       RPG
 ___ CAL1FM        SLM      ___ DOW3          OCL        ___ MARMTD        OCL
 ___ CAL1FM        SSM      ___ GETCST        RLM        ___ MARMTD        RLM
 ___ CALSHORT      RPG      ___ GETCST        RPG        ___ MARMTD        RPG
 ___ CUSTB         RPG      ___ GSORTX        OCL        ___ MINRPG        RLM
 ___ DAILY         RPG      ___ HEX2          OCL        ___ MINRPG        RPG
 ___ DELNMX        OCL      ___ HEX2          RLM        ___ MKRP          RLM
                                                                   37 More...
```

PC using a 2400 baud modem, and even with the slowww screen refresh rates at 2400, the speed of UNIBOL on the RS/6000 made us almost as productive on UNIBOL from a remote terminal as we would have been on a local S/36. Not only the compile times, but the speed in general of developing a program on UNIBOL is terrific. From viewing spool files to executing programs, UNIBOL is fast. Despite its lacking modern editing features, PIE's line-oriented SEU provides fast loading and saving of source and procedure members.

Writing and testing programs requires frequent use of the print spooler and COPYPRT utilities, and UNIBOL doesn't let you down here. The UNIBOL print spooler exhibits an impressive level of S/36 emulation. Its faithful implementation works just as you'd expect, with all the features and capabilities of the S/36 print spooler. COPYPRT also makes you feel at home and, except for a few keyboard anomalies, works exactly as it should.

It's also frustrating to program on UNIBOL's virtual S/36 without using what have become integral, third-party S/36 development tools. Without them, you're quickly reminded you're not on a S/36 anymore. Most of these tools probably never will be available under UNIBOL because of licensing, marketing, or technical complications. If the Unix S/36 work-alikes capture a wide market share, someone will make a killing providing third-party, Unix-based RPG and S/36-like development tools for them.

Perhaps the most frustrating thing about UNIBOL's programming environment, though, is not what it offers, but what it doesn't. By delivering

the product with such comprehensive replication of the S/36 environment, the programmers who wrote UNIBOL demonstrated they are a sharp bunch. Surely they could deliver a Borland Integrated Development Environment or a Microsoft Programmer's Workbench type of environment for RPG and the S/36. Given UNIBOL's fast compile times, full-featured debugger, and accessibility to Unix's low-level environment, developing an impressive programming environment should be a snap for the Irish programmers.

More than Token Support

UNIBOL's RPG compiler doesn't create native RS/6000 executable files, but rather compiles a source member to an intermediate tokenized (or compressed) member. At runtime, UNIBOL directly executes this tokenized member in conjunction with a large, re-entrant runtime library of support functions. Part of the reason the compiler runs so fast, then, is that rather than creating a freestanding load member, it actually creates a tokenized member for use with the runtime library. Some people might think this partially interpreted approach would diminish response times, but performance numbers in Figure 21.3 prove otherwise.

Benchmarks aside, the interpreted nature of UNIBOL's RPG compiler isn't native in the truest sense. So in response to users who think they need native applications, Software Ireland currently has an RPG-to-C translator in beta test. The perception that native is better, however, isn't necessarily true. Tests show very little performance difference between executing the binary member created with C and executing the tokenized member created with UNIBOL's RPG compiler. The translator's biggest advantage will be its ability to let you embed C in your RPG.

UNIBOL's RPG compiler supports MRT and NEP programs; CONSOLE, KEYBORD, CRT, and SPECIAL files; RPG/400 features such as external program calls and externally described files, as well as ASNA 400RPG and BPS RPG II½ RPG/400-like extensions; unlimited program size (one test program contained a 1,000,000-character array); and an unlimited number of program files. UNIBOL also provides a full-featured debugger that controls program execution and lets you set breakpoints to examine variables and indicators.

Many S/36 users migrating to the AS/400 know the anguish of trying to implement S/36 assembler routines on the AS/400. UNIBOL, recognizing the problem, supports several popular assembler subroutines available on the S/36. Current subroutine support includes the ability to determine cursor position on the screen, work with printer files, and read and write in ASCII format (remember, S/36 files retain their EBCDIC format under UNIBOL). UNIBOL also provides a subroutine interface — requiring C know-how — with necessary information about implementing your own subroutines. So if your applications use some S/36 subroutines currently unsupported on UNIBOL, this subroutine

interface can ease your assembler-induced migration distress. In addition, UNIBOL supports COBOL with the RS/6000 Microfocus COBOL compiler.

Up and Coming

Software Ireland has big plans for UNIBOL. Besides promising continued enhancements and additions to UNIBOL's S/36 features and facilities support, Software Ireland pledges to add features and functionality to UNIBOL. A graphical user interface (GUI) environment, providing a GUI to existing RPG applications, already is in the testing stages. These GUI features — provided by a PC running Windows 3.0 attached to the RS/6000 via TCP/IP and either an Ethernet or Token-Ring connection — don't require any program changes. This GUI product promises to offer some of the sophisticated cooperative processing power available on several new AS/400 products. Software Ireland also promises embedded C and SQL support for its RPG compiler and transparent access to third-party Unix database products such as Oracle, Informix, Sybase, and Ingres.

If those features aren't enough, Software Ireland also promises a 1993 release of UNIBOL/400, which will support native AS/400 applications on Unix platforms. UNIBOL, Inc., pointing to the number of different UNIBOL-supported platforms, stresses that its product is a multiplatform — not a single-platform — solution. Users can run UNIBOL on high-end RS/6000s as well as on low-priced PCs. The typical UNIBOL customer may be a maxed-out S/36 shop; but UNIBOL, Inc., also notes that many of its customers are successful software houses with good, vertical S/36 packages. These software vendors, witnessing an increasing open-systems awareness in their marketplaces, see UNIBOL as another way to win customers.

Figure 21.5 shows UNIBOL licensing fees, including RPG/400 support (which includes support for ASNA and BPS S/36 preprocessors). If you don't need RPG/400 support, UNIBOL is also available without it, as shown in the right column of Figure 21.5. UNIBOL charges 15 percent of product licensing fees for annual support.

OPEN RS/36: A LOAD 'N' GO ALTERNATIVE

Open RS/36 is an interesting S/36 work-alike, with a special spin on S/36 emulation: It is S/36 load member compatible. That's right — no compiling necessary. You move your S/36 load members to the RS/6000, and they run! Sound too good to be true? It's not. It works. In our tests, it ran previously compiled load members from a S/36 without a hitch. You'll get a more detailed look at Open RS/36's Load Member Processor (LMP) at the end of this chapter.

Open RS/36 also offers a complete S/36 environment, including a runtime system with all the features necessary to emulate the S/36 environment and process S/36 load members. The runtime shell achieves such a high

Figure 21.5
UNIBOL Pricing

No. Users	UNIBOL with RPG/400 Runtime	UNIBOL with RPG/400 Developer	UNIBOL with RPGII Runtime	UNIBOL with RPGII Developer
2*	$1,660	$3,300	$1,550	$3,090
4*	2,160	4,340	1,980	3,980
6*	2,770	5,570	2,510	5,040
8	3,390	6,800	3,040	6,100
12	4,220	8,430	3,740	7,490
16	5,040	10,110	4,450	8,910
20	5,660	11,070	4,980	9,730
24	6,270	11,940	5,500	10,460
28	6,900	12,770	6,030	11,170
32	7,500	13,520	6,560	11,810
40	8,660	15,140	7,550	13,200
48	9,790	16,640	8,530	14,500
56	10,940	18,060	9,520	15,710
64	12,100	18,770	10,510	16,300
96	17,060	24,750	14,770	21,420
128	20,620	27,870	17,820	24,080
129+	29,170	36,460	25,150	31,440

* **Note:** 2-, 4-, and 6-user licenses are available only for 386- and 486-based sysems. The developer package includes all compilers and other resources to create programs, screen formats, etc.

degree of compatibility with the S/36 that you can even use the S/36 Displayed Messages Guide to help diagnose Open RS/36 messages!

The Load Member Processor is a part of Open RS/36's basic shell. For program development, you use the native S/36-licensed RPG compiler. Universal adds a POP clone, including Iris Software's FSEDIT-compatible Blue Iris editor, and other program-development tools such as a screen-format compiler ($SFGR) and #GSORT. An additional development system that includes a native RPG compiler will be made available later. This native compiler would provide the advanced features necessary to exploit the RS/6000 environment. This development system would be a superset of the S/36 programming environment, with RPG/400-like extensions (e.g., externally described files and CALL/PARM). Also included would be a source-level debugger that provides breakpoints, variable inspectors, and other features you'd expect from a high-end debugger. Universal has plans to enhance its development environment even further with a graphical user interface (GUI) for RPG programs, support for Unix XWindows, and an interface to third-party relational databases. Figures 21.1 and 21.2 summarize the features and facilities available in Open RS/36.

Look and Feel

When you sit at the keyboard of an RS/6000 running Open RS/36, you feel you are at the keyboard of a S/36. The level of compatibility is uncanny. Among the key things Open RS/36 offers:

An OCL processor. The OCL processor furnished is completely compatible with the S/36, providing most of the commands you are used to and supporting conditional and substitution expressions. The processor also supports standard S/36 session, job, and job-step contexts.

Most utilities and procedures. Open RS/36 supports $MAINT, $COPY, $SFGR, DELETE, RENAME, COPYPRT, Data File Utility (DFU), and Screen Design Aid (SDA), as well as many other S/36 utilities and procedures. Some S/36 facilities — KEYSORT, for example — become obsolete under Open RS/36 (more on this in a moment). Also implemented are most operator control commands for managing jobs, print spooling, the job queue, and user access. A replacement for CNFIGSSP provides the specific features needed to configure the RS/6000 for its role as a S/36 work-alike.

Disk Data Management. Open RS/36 provides complete support for the S/36 DDM environment, including the VTOC structure. The work-alike uses a B-tree indexing mechanism to emulate the S/36's indexed sequential access method (ISAM). The B-tree file system eliminates the need for keysorts; thus, the KEYSORT procedure is ignored whenever it is encountered in OCL. Note, though, the difference between being ignored and unsupported. The people at Universal Software consider two-way compatibility important. They envision some users or developers mixing RS/6000s and S/36s in an organization. Using the S/36 as the lowest common denominator, you can write code on the RS/6000 with the RS/6000 as the intended execution platform; or you can download that same code to the S/36 and, without changes, execute it there. Any code included for compliance with the S/36 but not needed by the RS/6000 (such as KEYSORT) is ignored when you use the code on the RS/6000. And Open RS/36 stores data files in EBCDIC on the RS/6000 (normally an ASCII-based machine), so data representation — including packed and binary fields — is identical to that on the S/36. Open RS/36 also supports all S/36 file types, including alternate indexes. Source and procedure members are stored in ASCII, letting you use native Unix text-processing tools (e.g., text compare) as development aids.

#GSORT. Universal's product implements a native, syntactically compatible version of #GSORT, achieving functional equivalence with the S/36. All three S/36 sort types (regular, summary, and ADDROUT) are supported. And because Open RS/36 stores all file data in EBCDIC, the data collating sequence is identical to that of the S/36.

Workstation Data Management. Open RS/36 provides support for all 5250 data stream I/O operations, including user-defined data streams written

directly to the workstation file. An impressive claim. To prove that it worked, we used a load member that performed binary-level 5250 workstation I/O. The program ran successfully — manipulating the workstation data stream on the fly. Workstation data management also fully supports all workstation display attributes (e.g., reverse image, high-intensity, blink). In addition, Open RS/36 sports seven sessions per terminal, with hot-key between sessions. Later plans call for a session-to-session messaging system so that RPG programs running in different sessions can talk to each other and can start jobs in other sessions.

RS/6000s generally require ASCII workstations and printers. But Universal has a twinaxial adapter available that lets you attach S/36 5250 workstations and printers to the RS/6000. This accommodation is important not only because of the financial investment these devices represent, but also because of the ergonomic cost of changing user hardware. Did you ever try to find the Print key on a PC emulating a 5250 terminal on the S/36? By supporting twinax 5250 devices, Universal hopes to help you avoid such "learning bumps."

POP. A superset of POP — called Open POP/36 — does everything IBM's POP does and more. Open RS/36's version of POP includes an enhanced version of the aforementioned Blue Iris full-screen editor that follows FSEDIT user interface conventions and adds multiple-level undo and redo, user-defined keystroke macros, multimember editing, clipboard copy and paste, and many other editing goodies long missing from the S/36.

Load Member Processor. Last, and best, is the Load Member Processor mentioned earlier. For a detailed look at this unique Open RS/36 feature, see "Machine Mimicry," page 371.

See How They Run

To test Open RS/36, we migrated several S/36 load members. These load members included interactive workstation applications, batch report-writing jobs (complete with OCL), and three or four #GSORTS. Data files and the test library were loaded onto Universal's S/36 and then migrated electronically to the RS/6000. Migration is achieved via Andrew's RS/6000 Alliance 5250 emulation board and software, which make the RS/6000 look like a cluster of 5250 terminals to the S/36. The Alliance board transfers data at about 60 MB per hour. At $4,000, though, the Andrew board is a pricey migration method for low-end systems. Universal also supports the S/36 tape cartridge as a migration medium, using SAVE, SAVELIBR, and FROMLIBR formats.

When transferred, files retain all their attributes (e.g., delete-capable, extend value). The migration tool first loads libraries and files onto the RS/6000 and then automatically builds alternate indexes. The library came across in the same condition as it had been in on the S/36; Open POP/36 showed the same library list IBM's POP had.

The first test, a simple batch report job, correctly printed a report. The

migration-to-running path was pretty easy — no recompilation or compatibility checking; our load member simply ran and produced correct results. The speed with which the batch job was executed, while not blindingly fast, was certainly acceptable. Although no formal timing benchmarks were performed, the RS/6000 seemed to perform in the same class as the S/36.

The next test, an interactive program, encountered some minor glitches with conditional screen attributes, but the program ran and produced the output as expected. The real problem was with the screen-refresh rate, which at 9,600 bps was still too slow. The S/36 really spoils you with its high-speed screen refreshes. Universal has since corrected the screen refresh times and also noted that users can trade cost for speed by using higher bandwidth terminal connections — up to 38,400 bps.

Next up was a large ADDROUT sort with a few conditional lines of OCL. Original sort test times were disappointing, but Open RS/36 sort throughput is now reported to be approximately twice that of the S/36. The original sort problem certainly wasn't the end of the world, but it had a sobering effect. We wanted Open RS/36 to work — its promise is so great (and, to its credit, even though the sort was slow, it did perform correctly). After the sort test, we became a little less naive, a little harder to impress.

Last, a S/36 load member was tested that used ASNA's external program calls. This load member didn't execute properly at first. It was discovered that the load member had been created three or four years ago, using an older version of ASNA's RPG/III. That old load member didn't have the same characteristics as those compiled under newer versions. It wasn't a critical error, and in short order the engineers had tweaked the LMP to interpret the program. This experience illustrates the level of precision required to successfully emulate a S/36.

Testing, Testing

The LMP accounts for much of Open RS/36's fidelity with the S/36. Consider the RPG trick of accepting input with the function keys (by using the READ opcode without any indicators), or of embedding screen attribute hexadecimal characters in a user-defined workstation output stream. Open RS/36 is so compatible with the S/36 — Universal calls it "bug-level compatible" — that things such as these work implicitly. Although the overall performance of the test programs was a bit disappointing, Universal claimed not to have yet tweaked Open RS/36 to maximize performance. Since our testing, Universal says it has improved response times considerably with just a little tuning.

While speed remains to be proven, Open RS/36's S/36 compatibility seems assured. Universal's engineers displayed the test suite of programs and procedures they used to ensure that Open RS/36 is truly compatible with the S/36. The thorough testing was an impressive achievement. During their testing,

Universal turned up an interesting number of things that don't quite work the way the IBM manuals say. Not big things, mind you, but when you're reverse-engineering load member compatibility, a little thing can make the difference.

Load Member Compatible

Because Open RS/36 is load member compatible, take a second to consider the implications. The S/36 RPG II compiler is a load member, isn't it? Therefore, you can run it under Open RS/36 on the RS/6000. In fact, you can compile a program on the RS/6000 using the IBM compiler, migrate the created load member back to the S/36, and execute the program. Universal Software makes no claims about the legality of running your IBM RPG compiler under Open RS/36, just as VCR salespeople make no claims about the legal implications of using a VCR. You are responsible for ensuring compliance with IBM's licensing terms. However, Universal reports that at least one Open RS/36 customer has received written authorization from IBM to transfer the S/36 RPG compiler to an RS/6000, with the stipulation that the compiler be removed from the S/36 within 90 days. Universal does make the point that you don't need any S/36 SSP products to run its system; the whole thing can run using Open RS/36 code exclusively.

There's a bit of a rub here, though. If you migrate your load members to the RS/6000 and expect never to change them, everything is fine. When you need to modify a program, though, you have two choices: You can compile the source program using Open RS/36's native RPG compiler, or you can compile using your true-Blue copy of the S/36 RPG compiler. By compiling your code with the Open RS/36 native RPG compiler, you risk losing the S/36 fidelity that Universal worked so hard to achieve with its Load Member Processor. That's because now you're executing native RS/6000 code. But by moving your S/36 RPG compiler to the RS/6000 and using it to compile your code — generating a true S/36 load member — you risk losing some performance, and you must live within IBM's RPG feature set.

What Doesn't it Do?

Lest you think Open RS/36 can walk on water, here are some things it does not do. It doesn't offer remote workstation support (RWS), at least as we know it on the S/36. You can achieve simple remote workstation attachment with ASCII workstations and asynchronous modems, but high-level SNA/ICF facilities such as Advanced Program-to-Program Communications (APPC) and Advanced Peer-to-Peer Networking (APPN) aren't supported; you'll have to recode applications that use ICF. Query, office products, and IDDU aren't supported either. (There is, though, talk of providing a DisplayWrite/36-to-Unix WordPerfect document conversion at some time in the future.) Password security is implemented in the emulated S/36 environment, but resource security

for libraries and files follows Unix rules.

Universal makes no apologies for what Open RS/36 doesn't do. The company has a clear vision of the S/36 users it would like as Release 1.0 customers, and it intends this first release of Open RS/36 to appeal to those customers. Universal is initially targeting S/36 users with an immediate need to upgrade resources (perhaps because of reaching S/36 DASD or workstation limits). Universal says that when an unsupported feature (communications or Query, for example) proves to be a marketing roadblock, support for that feature will be re-evaluated.

Scheduled for delivery in the first release of Open RS/36, but not available during this product evaluation, were implementations of the S/36 print spooler, JOBQ, EVOKE, SDA, DFU, and tape and diskette management emulation. Some were nearly ready during testing, but we can't say we saw them and no documentation was available. However, since this evaluation, all of the major pieces are in place in RS/36 version 1.2 and are being shipped to customers.

The price for the basic Open RS/36 operating environment starts at $2,500 for a five-user license for the Model 220 RS/6000. For larger machines, Open RS/36 is $4,000 for a 10-user license. Additional users can be added to either license for $400 per user. The final price for the development environment has yet to be announced.

When pressed about comparative performance figures, Universal equivocated a bit. But assuming everything that needs "optimizing for speed" will receive the appropriate attention, think along the lines of an RS/6000 model 340H providing six to 15 times the performance of a S/36 B24. These figures are based primarily on Universal's disk I/O tests, but Universal also claims that total user throughput will far exceed that of the S/36.

AIX: An Intense Experience?

The RS/6000, being an "open system," represents an important step for the IBM midrange. The naysayers (you know who you are) say it's ludicrous to assume a S/36 user will be willing to spend the time and effort to learn AIX, the RS/6000's version of Unix, to make an environment such as Open RS/36 work. Universal admits that some knowledge of AIX will be required by system operators and programmers, but none by users. And it's Universal's position that learning the bit of Unix Open RS/36 requires is less difficult by an order of magnitude than learning enough OS/400 to use the AS/400's S/36E. Universal believes it has made migration from a S/36 to Open RS/36 easier than going from the S/36 to an AS/400.

AIX/Unix also makes the solution interesting because Open RS/36 is a Unix generic program, not RS/6000-dependent. Therefore, Universal could run Open RS/36 on, say, a MIPS Unix machine or a 386 PC running Santa Cruz Operation Unix. The possibilities are truly amazing, especially when you factor

in the enormous amount of S/36 code currently in the marketplace. Not only does an RS/6000 S/36 work-alike offer S/36 users asylum from the AS/400, but it also offers any sellers of S/36 code a new place to sell it. Regardless of the fact that it works and looks like a S/36, Open RS/36 might just become a computing solution in its own right.

The Magic Answer?

It is just too soon to say how well S/36 users will embrace S/36 work-alike platforms on the RS/6000 generally, and Open RS/36 in particular. Buying what amounts to a complete operating system from a third-party vendor to run on your IBM machine requires an enormous amount of faith in that vendor. For whatever it does or does not offer, IBM is likely to support OS/400 for a long time. Can the same be said for a relatively small newcomer? At what price "openness"?

IBM Rochester has lately been talking a lot about your "total cost of ownership," as well as positioning the AS/400 as a "full-range" machine. As you review your options, Rochester wants you to count education, migration, and support costs, as well as the costs of adding "emerging technology" solutions to your system. If you consider an RS/6000 as a midrange alternative for your business, don't discount these peripheral issues. They are very real and bear serious consideration.

The United States Department of Agriculture (USDA) Kansas City Management Office recently undertook an extensive study to determine the viability of Open RS/36 as a replacement for its aging network of 3,000 S/36s. Reports indicate the USDA is impressed with what it's seen. The Department is one of IBM's largest S/36 accounts, and its opinion is not taken lightly in Rochester. The USDA reportedly looked extensively at moving its S/36 applications to the AS/400 and decided it needed to evaluate less-expensive alternatives. Regardless of how well the AS/400 plays in Peoria, it will be interesting to see how the RS/6000 and Open RS/36 play in Kansas City.

Product Information

UNIBOL
UNIBOL, Inc.
1800 Sandy Plains Parkway
Marietta, GA 30066
(404) 424-5345
Price: See Figure 21.5

Open RS/36
Universal Software
4400 MacArthur Boulevard
Fifth Floor
Newport Beach, CA 92660
(714) 851-8021
Price: $400 per user — minimum license: 10 users

Machine Mimicry

In the world of S/36 work-alike platforms, Open RS/36 has one unique feature that simplifies achieving close compatibility with the S/36: the Load Member Processor (LMP). In contrast to other work-alike approaches, which require recompilation of RPG source code using a foreign platform compiler, the LMP actually simulates the execution of the S/36 CPU. S/36 executable programs exist as library load members, and a load member is nothing more than a string of machine instructions. The LMP interprets the machine instructions contained in the load member, carrying out the same low-level adds, compares, and branches that the S/36 Main Storage Processor executes. If the simulation of S/36 hardware is accurate, emulated programs behave just as they do on the S/36, right down to the smallest nuance.

The Payoff

Directly executing compiled RPG programs brings several benefits. First, it eliminates the chore of recompilation, reducing migration to the task of moving data and programs from the S/36 to the RS/6000. Second, problems arising from differences in data representation disappear because there are no such differences. Although the RS/6000 is an ASCII machine, the LMP-interpreted RPG object programs can work directly on S/36 EBCDIC, packed, and binary data without performing conversions. The LMP, then, provides compatibility better even than the AS/400's S/36-compatible RPG compiler, which has problems with S/36-style packed and zoned data. Third, programs having hidden dependencies on S/36 RPG idiosyncrasies — such as the status of indicators at various points in the RPG cycle — don't fail as they would in a translation environment. That's because the internal state of the RPG program — its variables, indicators, and even instruction opcodes — is bit-for-bit identical with the same program running on a real S/36. And once you've accomplished the feat of object-code interpretation for RPG programs, making it work for COBOL, FORTRAN, and even assembly language object programs is trivial.

This ability to execute S/36 object code also eliminates the need to precisely replicate the behavior of the S/36 RPG compiler. Other vendors must write from scratch an RPG compiler functionally equivalent to IBM's — a daunting task given the quirks and complexities buried in S/36 RPG. Open RS/36 can simply run IBM's own RPG compiler, which, after all, is a set of S/36 machine-language load members. This same benefit falls to other IBM software products, such as the COBOL compiler, Data File Utility (DFU), and Screen Design Aid (SDA), and even to third-party products such as ASNA's 400RPG and BPS's RPG II½.

Under the Covers

While machine simulation makes for a purer reproduction of the S/36, actually building a usable machine-instruction interpreter is an inexact and tedious task. The science of software machine simulation is not new — the IBM S/360 sported a software emulator for its predecessor, the IBM 1401. In modern times, software emulators such as SoftPC simulate the IBM PC on Macintosh, Next, and RS/6000 computers. These emulators are so good that even the PC-DOS operating system doesn't know it's running in emulation: Using such a simulator, you can run

Machine Mimicry *(continued)*

Figure A

S/36 Machine Language Instruction Set

Opcode	Description	Opcode	Description
ALC	Add Logical Characters	MVI	Move Immediate
ALI	Add Logical Immediate	SBF	Set Bits Off Masked
A	Add to Register	SBN	Set Bits On Masked
AZ	Add Zoned Decimal	SRC	Shift Right Character
BC	Branch on Condition	ST	Store Register
CLC	Compare Logical Character	S	Subtract from Register
CLI	Compare Logical Immediate	SLC	Subtract Logical Character
ED	Edit	SLI	Subtract Logical Immediate
ITC	Insert and Test Character	SZ	Subtract Zoned Decimal
JC	Jump on Condition	**SVC**	**Supervisor Call**
LA	Load Address	TBF	Test Bits Off Masked
LPMR	Load Program Mode Register	TBN	Test Bits On Masked
L	Load Register	XFER	Transfer
MVC	Move Character	ZAZ	Zero and Add Zoned
MVX	Move Hexadecimal Character		

Microsoft Windows in a window on a Macintosh or Next. However, both the 1401 and the PC are single-processor machines, running single-user operating systems. The S/36, in contrast, is a multiprocessor machine with a multiuser, multitasking operating system. Although it's theoretically possible to build a S/36 emulator so detailed that you could run all of SSP transparently, Universal chose another path — one that achieves acceptable results with much less effort.

Instead of emulating the entire S/36 machine — which consists of at least two processors (the Main Storage Processor and the Control Storage Processor) and IBM-proprietary microcode — the LMP limits itself to simulating just the MSP. The S/36 MSP understands only 29 basic opcodes (Figure A), all but one of which the LMP can interpret with the same precision as true-Blue hardware. But that single exception — the Supervisor Call opcode — is a doozy. Where the other opcodes confine themselves to pushing bits and bytes around in memory, the Supervisor Call (SVC) provides several hundred special-purpose functions, including disk and workstation I/O, exception handling, data communications, and task management. Of these hundreds of functions, compiled RPG and COBOL programs use about 30, the RPG and COBOL compilers use another dozen, and SSP utilities such as DFU and SDA require an additional half dozen. By emulating this subset of about 50 common SVC functions, the LMP is able to execute the vast majority of S/36 object programs.

Many of the remaining SVC functions support SSP services such as print spooling, job queue, task management, OCL interpretation, and command processing. The LMP does not

provide these functions; they're emulated by other components of Open RS/36's Enhanced Compatibility Environment (ECE). But then Open RS/36 doesn't require IBM's SSP, or any other IBM software, to run S/36 applications — eliminating one sticky licensing issue. The IBM program products that the LMP can run are inexpensive, and they're necessary only for continued LMP-compatible software development. Even if users end up having to repurchase these products from IBM (rather than simply transferring the licenses from their old S/36s), the increased cost is negligible.

However, every silver lining has a dark cloud. The down side of Universal's limited SVC support is that some IBM program products you might like to move to the RS/6000 won't fly once they get there. In particular, IDDU, Query, DisplayWrite/36, and the entire suite of S/36 communications facilities (e.g., BSC, SNA, and ICF) won't run on the LMP. Universal may get IDDU and Query running by modifying the LMP to support a few additional SVC functions, but DW/36, as well as ICF and its ilk, are deeply intertwined with CSP services and complex data structures embedded in the SSP itself. DW/36, for example, relies on the CSP to provide word processing functions for 5250 terminals. Similarly, ICF depends on a multitude of SSP internal tables and queues. The complexity of, and scant IBM documentation on, such dependencies means that these products may never make their way to the RS/6000 — or to any other platform.

How Good Is Good Enough?

Of course, the success of the LMP hinges on how closely it mimics the S/36 MSP. Just as machine language emulation leverages compatibility for a whole range of S/36 software, even a tiny flaw in that emulation could cause equally wide-ranging problems. To validate the LMP, Universal's engineers used regression testing. First, they wrote a suite of test programs to exercise each S/36 machine instruction in all its variations. Each test program used its subject instruction to transform a set of test data in various ways. Then the memory image resulting from these transformations was stored as a dump file and moved to the RS/6000, where the engineers ran the test program again. After each test, special debugging components of the LMP compared the simulated S/36 memory image with the image obtained from the real S/36. Any difference meant a deviation from the S/36's operation and required a change to the LMP. After executing hundreds of such tests, Universal claims close to 100 percent compliance with S/36 basic machine instructions and the LMP's SVC subset. Universal's engineers say they also found a few minor differences in the way various S/36 models interpret some instructions, showing that even IBM finds room for interpretation in its hardware specifications.

Universal also reports that after the migration of more than a thousand RPG and COBOL application programs, only two instances of LMP interpretation failure occurred. Both were cases of undocumented machine instruction behavior.

Such attention to detail is laudable, but S/36 applications consist of more than load members alone. Menu, screen format, procedure, and data management behavior also affect appli-

Machine Mimicry *(continued)*

cations. How closely an entire Open RS/36-based application matches the S/36 depends on the fidelity of these emulated facilities as well as the LMP's. As our preliminary experience shows, not everything is perfect in this larger arena. But Universal is using similar regression testing techniques to validate Open RS/36's disk and workstation data management, OCL, menu, and screen format support.

How Fast Is Fast Enough?

Machine simulation takes its toll on performance. For each emulated S/36 machine instruction, the LMP must execute 100 or more RS/6000 machine instructions. The RS/6000 can execute 100 to 200 of these in the time the S/36 executes one instruction, resulting in instruction emulation being performed with speed roughly equivalent to native S/36 execution times.

However, the business-class programs making up the majority of S/36 applications don't use the CPU much, instead spending most of their time waiting for disk I/O. RS/6000 disk data management is about 10 times faster than that on the S/36, due both to faster disk hardware and to the efficient B-tree indexing mechanism Open RS/36 uses. Overall, according to Universal, applications should run several times more quickly than they do on the S/36. Our preliminary benchmarks bear this out in some cases, but not in others. Sorting, for example, was several times slower under Open RS/36. Universal, for its part, contends that the final, fully tweaked versions of the LMP and ECE will meet its performance goals. By recompiling the LMP using IBM's new optimizing C compiler, Universal hopes to improve S/36 instruction execution times by a factor of two or three, while an enhanced ECE data management component should make sorting and other batch processing much faster. Clearly, users want all the speed they can get, and many will look dimly on a "compatible" machine that doesn't offer throughput significantly better than what the S/36 provides. If Universal can achieve the performance levels it claims, users should be happy with Open RS/36 performance.

Afterword

"It is not best to swap horses while crossing the river."

—Abraham Lincoln

A reader having experience with the AS/400 might be tempted to chastise us for recommending the S/36 as a going proposition. "Are they living in the past?" you might ask. "Don't they know that the AS/400 is light years beyond the S/36?" The answer is that we do know about the AS/400's advantages, having used the machine extensively since its inception. But we also have experience on other platforms: Macintosh, Unix, OS/2 and Windows. Yes, the S/36 is old technology: character-based interfaces, batch processing, procedural languages. But so is the AS/400, with its own character-based interface, batch orientation, and limited suite of procedural languages. *New* technology — graphic user interfaces, modeless interaction, and object-oriented programming — are where the real improvements in application design and productivity are to be had. Although some minor additions to the AS/400 bring a taste of these technologies, we see little hope that the AS/400 will ever embrace them in earnest.

With its same-old, same-old approach, the AS/400 offers potential S/36 migratees only a small return on their investment. That's not to say that for a few S/36 users, the AS/400 isn't a good migration box today. For S/36 users needing better cooperative processing, improved database support, a wider range of programming languages, or more communications options, the AS/400 could be the box for you today. However, for the large majority of S/36 users, we think S/36 migration options with the portent of graphical user interfaces, object-oriented programming, vastly increased response times, and less dependence on IBM's proprietary architectures are worth waiting for. Thus, we say, if you don't have to move, don't. Wait for the good stuff. In the meantime, your S/36 is paid for and still running day-to-day business operations software such as order processing, billing, and payroll — in short, most of what you need right now. We feel strongly the S/36 is a reasonable place to stand your ground until the dust settles.

Index

A

#GSORT, presenting, 51
  addrout files, 155
  alternative to, 189
  DBLOCK and, 202
  Open RS/36 and, 364
  region size and, 202
#LIBRARY, 341
  user programs, 342
#SYSTASK, 29, 108
  maximum size of, 29
  See also Task Work Area (TWA)
$COPY program, 190, 218
  region size and, 202
$MAINT, 218
*36 object type, 343
*ENTRY, 115, 119, 130
  PLIST, 116
*IN
  array, 331
  fields, 331
*LIBL, 130
64K region, 128
400RPG, 123, 321
  code generated by, 326-327
  code using AND/OR operations, 331
  coding for subprograms, 126
  compiling programs and, 321
  explained, 323
  external file descriptions and, 325-328
  file names and, 124
  implementation of, 324
  library lists and, 131
  opcodes, 345
  preprocessing step, 324
  program invocation time, 129
  running called modules with, 130
  screen formats, 125
  UNIBOL and, 361
  See also RPG
5250 network, 68
  UNIBOL and, 354
5360, 60-61
  communications configuration for, 71
  disk capacity for, 87
  disk drives, 64-65
  finding stage level of, 91
  memory
    address space of, 22
    configuration, 63
    maximum, 81
  model D upgrade, 91-92
  relative performance factor, 62
  upgrading to, 92-94
  variations, 60-61
5362, 61
  communications configuration for, 71
  disk capacity for, 87
  disk drives, 64-65
  memory
    configuration, 63
    maximum, 81
  relative performance factor, 62
5363, 61
  communications configuration for, 71
  disk capacity for, 87
  disk drives, 64, 65
  memory
    configuration, 63
    maximum, 81
  relative performance factor, 62
5364, 61
  communications configuration for, 71
  disk capacity for, 87
  disk drives, 64-65
  memory
    configuration, 63
    maximum, 81
  as programmer's box, 95
  relative performance factor, 62
9402 Y10, 61
  communications configuration for, 71
  disk capacity for, 87
  disk drives, 64, 65
  memory
    configuration, 63
    maximum, 81
/COPY
  feature, 109, 110
  statement, 124, 125
// FILE statements, 117, 118
  execution-time file names and, 125
ACCELER8, 51
  Index Doctor and, 168
  TWS problem and, 232
Activation, 136
Address Compare Stop feature, 8
Address space. *See* Memory address space
Address translation, 24
  registers (ATRs), 24
ADDROUT sort, 189-190
  time required for, 189
Advanced program-to-program communications (APPC), 99, 367
Advanced program-to-program networking (APPN), 99, 367
AI/GBT, 79
AIX, 323, 353, 368-369
Alternate indexes, 33, 51-52, 189
  changing duplicate key values in, 191
  changing key field in, 224
  disk spindles and, 193
  duplicate keys in, 52
  key update ability, 190
  minimizing use of, 191
  overflow area, 191
  replacing indexed files with, 190
  See also Index
Amalgamated Software of North America (ASNA), 107, 123
  400RPG, 321, 323
  CKDT opcode, 333
Andrew Alliance board, 356, 365
ANSI Intelligent Peripheral Interface (IPI-3), 64
Appended buffers, 161
Appointment scheduling application, 138-142
Architecture
  DDM, 33
  memory, 19-25
  multiprocessor, 11-14
  single-processor, 11, 14
  virtual memory, 25-29
AS/400, 3, 59, 77, 108
  9402, 61
  advantages, 88
  audiences built for, 339-440
  compilers, 321-322

RPG/400, 322
S/36 RPG II-compatible, 322
configuring disk for, 348-350
debugging tools, 308
demand paging, 27
disk requirements, 349
worksheet, 349
environments, 321
native, 321
S/36E, 321
externally described files and, 324-328
file manipulation function, 280
growth configuration, 346
language compatibility, 123-124
library list facility, 130-131
MDLSYS, 350-351
memory requirement, 347-348
worksheet, 347
models
9402, 346
9404, 346
9406, 346
capacity chart (MB), 348
equivalent S/36, 347
selection of, 346-347
packed data storage and, 198
PDM, 359
profiler and, 303
reasons for migrating to, 339
reliability features, 350
S/36
comparison, 6
compatibility, 345
sizing up, 345-346
static configuration, 346
string handling functions, 264
system management functions, 288
upgrading to, 95, 322
AS/400 RPG User Guide, 322
Assembly language subroutines, 108
in COBOL, 360
upward compatibility of, 332
SUBR$C, 266
SUBR$F, 265-266
SUBR$X, 267-268
SUBRAT, 268
SUBRBX, 269
SUBRCO, 280-281
SUBRCP, 293
SUBRCS, 269-270
SUBRCT, 291-292
SUBRDT, 292
SUBRDU, 293
SUBREX, 294-295
SUBRFA, 281-286
SUBRLD, 271-274
SUBRLN, 295-296
SUBRLR, 274-276
SUBRMG, 296
SUBRPC, 296-297
SUBRPS, 297-298
SUBRRN, 286-287
SUBRSG, 276-279
SUBRSX, 298
SUBRSY, 288
SUBRTD, 309
SUBRUF, 288-289
SUBRUL, 289-290
SUBRUP, 270
SUBRUR, 290-291
SUBRWT, 298
See also Subroutines
ATOM, 5
Auto-response values, 341

B

Balun (balanced-to-unbalanced) connectors, 68
BASIC programs, 210
Binary fields, 219
Binary storage, 198
Binary-to-decimal conversion, 219
Binding
early, 137
late, 114, 129, 137
BLDINDEX procedure, 189
time required for, 189
Blocking, 147
built-in, 149
cache vs., 239-240
data records, 147-151
benchmarks, 165
considerations, 154-155
enabling, 151-152
locality of, 154
index, 147, 155-156
considerations, 159-161
entire, 160
file reorganization and, 190
strategy, 165
Blue Iris editor, 363
BPS Information Services, 107, 123
RPG II½, 321
Buffers
allocating sizes for, 162-164
appended, 161
exceeding 64K, 161
index, 147, 155-156
sizing, 156-159
non-appended, 161
TWA, 233
TWA-resident, 161
See also Data buffers
Business Graphics Utility (BGU), 210
BYPASS-YES, 169, 194-195
when to use, 194-195

C

Cabling
twinax, 66
unshielded twisted-pair, 66-68
Cache
allocating memory to, 231
backup operations and, 248
blocking vs., 239-240
changing
configuration, 246
in response to changing environment, 246
counting, 241-243
disk, 237
locality and, 239
disk access with/without, 238
dynamically controlled, 245-246
explained, 237-240
hit, 237
locales, 238
locality, 238-239
miss, 237, 238
pages, 238
locality effect and, 238
page size, 240
quick, 243-244
situations for turning off, 248
size, 242
starting out with, 240-241
use decision, 247-248
value-added, 247
write-through, 239
See also CACHIQ utility; Memory
CACHE Facility, 77-78
CACHE procedure, 216, 240
syntax, 240
VASP modifications, 247
CACHIQ utility, 243
display, 244
log file, 243, 244
logfile record format, 245
resetting counters, 244
stopping, 244
suggested use, 215
syntax for, 243
See also Cache
CALL opcode, 114, 123
speeding up, 119
CANCEL command, 310
CATALOG listing, 168

CHANGE PRT (G P) command, 341
CHKDUP, 54
CKDT opcode, 333
CL
  commands, 340
    SEU and, 344
  compiler, 340
CNFIGSSP, 73
  procedure, 216
    for screen formats, 79
  screen 17.0, 81
COBOL
  programming, 299
  RPG assembler subroutine bridge, 299
  sample program using RBRIDG, 301-302
  WORKING STORAGE section, 299
COBOL fields, 219
  COMPUTATIONAL-2, 219
  COMPUTATIONAL-3, 219
Communications adapters, 10-11, 70-71, 93
  ELCA, 70, 93
  MLCA, 70, 93
  physical interfaces for, 70-71
  prices for, 96
  recommended, 96
  SLCA, 70, 93, 95-96
  upgrading, 96
Communications line usage, 213-215
COMPRESS procedure, 83, 202
  smart, 83, 84, 85
Control blocks, 29-30
  SQS requirements for, 31
  VTOC, 30-31
Controllers, 8-10
Control storage, 7-8
Control Storage Processor (CSP), 5-6, 7-8
  characteristics of, 6
  control storage, 7-8
  disk I/O scheduling, 212
  fixed nucleus and, 19
  functions of, 7
  high usage, 217-218
  machines using, 7
  Model D, 92
  task switching and, 13-14
  usage measurement, 210
  See also Main Storage Processor (MSP)
COPYDATA procedure, 102, 169, 190
COPYPRT, 360
COPYPRT CRT utility, 209
Coupling, 135
CSP/I, 6

D

Data buffers, 147
  44K limit, 153
  appended, 152
  for called programs, 164
  DBLOCK factors for, 153
  default, 149
  dirty flags on, 154
  function of, 147
  optimum usage of, 150
  sizing, 152-154
  See also Index buffers, Buffers
Data Description Specifications (DDS), 330
Data files, 39-43
  index areas, 41-43
  organization/operation of, 39-41
  usage on S/36, 332
Data records, 39-40
  blocking, 147-151
    benchmarks, 165
    considerations, 154-155
    enabling, 151-152
    See also DBLOCK; RPG F-spec
  format name, 327
  ISAM and, 41
  length of, 41
  memory and, 152
  NITU (no intention to update), 180-181
  RRN (relative record number), 280
  sector placement of, 40
  speeding addition of, 194-195
  unlocking, 186
Data Storage Controller (DSC), 93
  processor, 6, 8, 9-10
    operation of, 10
Data-transfer rate, 37
Date-differentiated files, 168
DBLOCK, 102, 177
  in checking blocking performance, 225
  factors for data buffer sizes, 153
  IBLOCK combination with, 160-161
  memory and, 233
  values after file reorganization, 190
  See also IBLOCK
DBLOCK keyword, 151
  enabling record blocking with, 152
  See also Blocking
Deadly embrace, 117, 178
  batch programs and, 180
  coding sequence to avoid, 181
  coding sequence leading to, 179
  diagram of, 179
  "lockstep" phenomenon, 223
  SHOWUR utility and, 181, 186
  See also One-way embrace
Debuggers, 308
  BPS, 320
  Cybra, 320
  UNIBOL, 361
Debugging, 308
  source-level, 309
  testing and, 313
Debug mode, 308
DEBUG statement, 308
Decimal data errors, 332
  code to avoid, 332-333
Decimal-to-binary conversion, 219
Decomposition
  functional, 137-138
  procedural, 137-138
DEFN operation, 330
Degap operation, 49, 224
Delayed index maintenance, 43
DELET opcode, 328
Device control assembler subroutines, 292-298
  SUBRCP, 293
  SUBRDU, 293
  SUBREX, 294-295
  SUBRLN, 295-296
  SUBRMG, 296
  SUBRPC, 296-297
  SUBRPS, 297-298
  SUBRSX, 298
  SUBRWT, 298
Digital Data Service (DDS), 97
  adapter (DDSA), 98
  statements, 132
Digital Service Unit (DSU), 98
Disk accesses, 257
Disk data management (DDM), 33
  architecture, 33
  built-in blocking, 149
  called keysort, 191
  disk lock arm feature and, 212
  index maintenance, 224
  logical-file operations, 37
  minimizing weaknesses of, 167-187
Disk drives, 15-16
  9332, 64
  bytes per cylinder of, 66
  characteristics of, 65
  configurations of, 65
  data-access time of, 38
  disk space, 83-86
  operating parameters of, 38
  performance of, 61, 64

pricing, 87
See also Hard disk; spindle
Disk I/O, 33
buffering, 202
CSP scheduling of, 212
decreasing, 152, 156
by adding memory, 215
disks and, 33-39
disk spindles and, 86
eliminating, 34
logical, 150
performance, 203
physical, 147, 148, 150
See also I/O
Disks
access time of, 64
capacity of, 87
extends, 195
understanding mechanics of, 33-39
usage of, 211
high, 215-216
unbalanced, 216-217
See also Hard disk
Display Station Pass Through (DSPT), 94
DisplayWrite/36, 9, 232
DisplayWrite/36-to-Unix WordPerfect document conversion, 367
Distributed Data Management (DDM), 94, 98-102
functions not supported by, 99
functions supported by, 99
guidelines for best performance, 102
more information on, 101-102
operation overview, 100
setting up, 101-102
Documentation, program, 317
Double-bit errors, 16
DSU, 344
D T command, 210
Dump file, 312
DUMP operation, 308, 309
Dumps
formatted, 309, 312
listing of, 312
"snapshot," 309, 311
DUPKEY processing, 33, 51-52
Duplicate keys, 168
changing values in alternate indexes, 191
in Index Doctor report, 170
ripple-down add and, 192

E

EDITNRD procedure, 101
EIA/CITT (RS-232) interface, 98
ELCA (Eight-Line Communications Adapter), 8, 10-11, 70, 93
aggregate line speed, 71
prices for, 96
as remedy for line sharing, 214
upgrading to, 96
Elms Technical Communications, 59
Error Recovery Analysis Report (ERAP), 12
Ethernet connection, 356
EXTEND keyword, 195
EXTEND value, 85, 195
default, 85, 195
override, 85
Externally described field names, 328
Externally described files, 124, 324
AS/400 data files and, 324-325
implementation of, 324-328
with RPG II, 328
RPG program using, 325
External program calls (EPC), 28, 79, 323
15 disk files limitation and, 107
64K region limitation and, 107
benefits, 119-121
capabilities, 107
coding, 114-117
with disk files, 117-118
disk I/O and, 216
late binding, 114
memory and, 108-109
modularization by, 141
opcodes, 114, 115
products compared
400RPG, 123
IBM, 123
RPG II1/2, 123
program invocation, 111, 258
program linkage using, 259
for program-to-program communication, 107
recursive, 113
response time and, 249
RPG/400 operation codes and, 345
RPG interfaces for, 107
vendor offering comparison, 123-133
virtual memory and, 232
See also Programs
External subprogram deactivation, 136

F

Field overrides, 328
Fields
binary, 219
COBOL, 219
packed, 219
Field types
binary, 197
packed, 197
zoned decimal, 197
File Access Counters, 222
by file, 223
by task, 223
Record waits column, 223
File extends, 84-85
File I/O, 124
File manipulation assembler subroutines, 279-287
SUBRCO, 280-281
SUBRFA, 281-286
SUBRRN, 286-287
Files
addrout, 155
data, 39-43
date-differentiated, 168
dump, 312
extend, 195-196
changing, value, 197
default value, 197
retrieving, value, 196
sequence of events for, 196
externally described, 124, 324
implementation of, 324-328
with S/36 RPG II, 330
full procedural, 332
index, 41
multiple record-type, 332
names of, 124
at compile time, 124
execution-time, 124
operation codes for, 328-330
placeholder, 83, 84
placement of, 86, 193
IBM's recommendations for, 217
on appropriate drive, 193
randomly accessed, 154
reorganizing, 190
spool, 83
unblocked, reading, 150
unshared, 193-194
File sharing
program, OCL for, 118
RPG, between programs, 118
File storage, 39
File Transfer Subroutines (FTS), 94
FILEXTND utility, 196
uses for, 196-197
Fixed nucleus, 19, 232
Formatted dump, 309
FORTRAN programs, 210
Fragmentation, 22
causation, 23

solving, 22-25
FREELOW, 83
FREEHIGH, 83
FREE opcode, 114, 116
Functional decomposition, 137
example, 138

G

Growth configuration, 346

H

Hard disk
anatomy illustration, 34
cylinder, 34
illustrated, 36
data access time, 38
physical units, 35
platters, 33, 34
read/write heads, 34
illustrated, 36
sectors, 34
illustrated, 35, 36
record placement in, 40
spindle, 34
tracks, 34
illustrated, 35
servo, 34
See also Disk drives
High disk usage, 215-216

I

IBLOCK, 156, 177
in checking blocking performance, 225
DBLOCK combination with, 160-161
example, 157-159
factors for index buffer size, 158
keyword, 156
memory and, 233
values, 156
determining, 158, 159
See also DBLOCK
IBM Fax Information Service, 59
IBM maintenance, 82
contracts, 103
recertification, 82
IDDU specs, 327
Index
alternate, 189
changing duplicate key value in, 191
changing key field in, 224
disk spindles and, 193
minimizing use of, 191
overflow area, 191
replacing indexed files with, 190
blocking, 147, 155-156
considerations, 159-161
entire, 160
factor, 152
file reorganization and, 190
function of, 155
for small files, 156
buffer, 147, 155-156
IBLOCK factors for, size, 158
sizing, 156-159
degapping, avoiding, 50-51
delayed, maintenance, 43
entries, 42, 156
gaps, 48-51
defined, 170
degap operation and, 49
in Index Doctor report, 170
keys, 41-42
added to index area, 46
duplicate, 51-52, 168
problem of adding randomly, 160
updating, 52
maintenance, 224
overflow, 46, 167, 170
storage, 43-55
keeping open, 171-177
track, 44, 171
variable, array processing, 265
See also Index Doctor
Index areas, 39
illustrated, 39
Index Doctor and, 167
keys added to, 46
overflow, 41-43, 46-48, 167
added, 47
containing many entries, 47
duplicate keys and, 53
full, 50
with gaps, 48-49
total ripple-down of entries, 51
primary, 41-43, 167
Index Doctor, 167
ACCELER8 users and, 168
actions after report, 170-171
date-differentiated files and, 168
frequency of using, 171
running, 167
sample report, 169
using, 186
See also Index
INDEXDR procedure, 167, 220, 224
Indexed Sequential Access Method (ISAM), 33, 353, 364
retrieving data records and, 41
storage index, 44
Information hiding, 110, 135
reasons for, 135
Instruction set, 3
commercial, 5
memory-to-memory, 6
register-to-register, 7
scientific, 3-4
Instructions, Supervisor Call (SVC), 7
Instrumentation code, 304
overhead, 306-307
Interfaces, 96
DDSA, 98
EIA/CITT (RS-232), 98
module, 124-128
V.35, 98
X.21, 98
X.25, 98
I/O
counters, 222
summary, 221
file, 124
workstation, 125, 129
See also Disk I/O
I/O Channel, 14-15
cycle steal and, 14-15

J

JOB-YES, 102, 175

K

KEEPOPEN procedure, 173-174
defined, 174
performance, 177
benchmarks, 178
Keys. *See* Index, keys
KEYSORT, 49, 51, 53-55
disk space for, 199-201
forcing, 54
real, 170
index blocking and, 159
in-memory, 199
identifying, 224-225
invoking, 53-54
problems with, 54
two kinds of, 199
workfile, 55
Keysorting, 33, 52-55
in-memory, 54
work-file, 54
KFMTS continuation line, 176
KLIST opcode, 329
to assemble fields of multipart key, 329

L

Late binding, 114, 129, 137
LDA, 107, 109

Libraries
  QS36PRC, 343
  QS36SRC, 342
  QSSP, 343
Library list, 130
  for exercising modules, 131
  facility, 130-131
Library manipulation assembler subroutines, 271-279
  SUBRLD, 271-274
  SUBRLR, 274-276
  SUBRSG, 276-279
Load Member Processor (LMP), 362-363, 365, 371-374
Local Area Network (LAN), 71-73
  configuring, support, 73
  processor, 11
  S/36, Communications Program Product, 73
  S/36, interfaces, 72
  Token-Ring, 60
    configurations, 74
    connectivity options, 73
Local Data Area (LDA), 257
  program linkage using, 258
Locality effect, 237
  disk cache and, 239
  multiple pages and, 238
"lockstep" phenomenon, 223
Logfile, 243
  record format, 245
Logical operations, 36-37
LOKUP operation, 220

M

Main memory, *See* Memory
  variable nucleus, 19-21
  See also Memory
Main programs, 110
  invocation by, 110
Main Storage Processor (MSP), 5, 6-7
  characteristics of, 6
  fixed nucleus and, 19
  high usage, 218-220
  Model D, 92
  privileged mode, 264
  task switching and, 13-14
  usage measurement, 210
  See also Control Storage Processor (CSP)
Map, 28-29
Mapping, 161
  parameters with data structure, 334
Memory
  adding additional, 21, 77-79
    procedure for, 216
  allocating, to Cache, 231
  AS/400, 347-348
  capacity per model, 81
  contiguous, 22
  data record blocking and, 152
  DBLOCK/IBLOCK and, 233
  eighth megabyte of, 79-81
  EPCs and, 108-109
  fragmentation, 22-25
  IBM's Office products and, 232
  main
    fixed nucleus, 19
    organization of, 19-21
    overcommitted, 25
    user area, 21
  management, 19-32
  MMETER (memory meter), 215, 234
  model configurations, 63
  non-blocked utilization of, 162
  non-contiguous, 22
  overhead, 306-307
  pages, 22
  pricing, 81-83
  real, 21-22, 77-78
    EPC and, 108-109
  requirements, 231-236
  screen format access and, 79
  system program residents and, 80
  for system programs, 201-202
  third-party, 79
  translated, 21, 22
  upgrading, 63
    cost of, 77
  user, available without CACHE, 78
  variable nucleus area of, 78
  virtual, 7, 25-26
    ACCELER8 and, 232
    EPCs and, 108-109, 232
    S/36, 26-27, 79
  See also Cache
Memory address space
  of 5360 Model D, 22
  64K limit, 24
  main, 21-22
  MSP, 6, 7
  real, 22
    generation example, 25
  virtual, 26
  See also Memory
Memory Resident Screen Formats (MRSF), 216
Message identification codes (MICs), 341
MICR, 5
Microcode, 5
MIPS, 7
MLCA (Multi-Line Communications Adapter), 8, 10-11, 70, 93
  aggregate line speed, 71
  prices for, 96
  as remedy for line sharing, 214
  upgrading to, 96
MMETER (memory meter), 215, 234
  screen, 235
Model System Tool (MDLSYS), 350-351
Modular applications, 124
  implementing, 135-144
    hiding, 135
Modular design, 135
Modular interface, 124-128
  creating, 138
Modularization, 141
  decomposing and, 144
Modules
  communication between, 135
  implementing, procedure for, 139-142
  interaction of, 135-136
  merging, 128
MOVEA operation, 220
MRT-NEP program, 109, 172
  for keeping storage index open, 172
  KPOPEN, 175-176
MS-DOS, file storage and, 39
MSG OCL statement, 341
Multipart keys, 329
  using data structure to describe, 329
  using KLIST to assemble fields of, 329
  using RPG II to describe, 329
Multiple Requester Terminals (MRTs), 257
  UNIBOL and, 361
Multiprocessor architecture, 11-14
  advantages of, 11-12
  single-processor vs., 11
  task switching in, 13
  timeline for, 14

N

Network Resource Directory (NRD), 101
Networks
  LAN, 11
  nonswitched, 97
  packet-switched, 97-98
  switched, 97
  Token-Ring, 11
  DDM and, 101
Never-Ending Programs (NEPs), 257
  UNIBOL and, 361
No-Intention-To-Update (NITU) strategy,

180-181
  RPG code using, 182-186
Non-switched network, 97
  multipoint (multidrop), 97
  point-to-point, 97
Nucleus pages, 234

O

Objects, IBM-supplied, 343
OCL
  program, 257
  program linkage using, 258
  Reader/Interpreter (RI), 340
One-way embrace, 178
  avoiding, 178-179
  coding sequence leading to, 179
  SHOWUR utility and, 181, 186
  See also Deadly embrace
Open RS/36, 263, 353
  #GSORT and, 364
  APPC and, 367
  APPN and, 367
  disk data management, 364
  environment, 323
  LMP, 362-363, 365, 371-374
  load member compatible, 367
  OCL processor, 364
  POP and, 365
  pricing, 368
  product information, 370
  RPG compiler, 367
  RWS and, 367
  SNA/ICF and, 367
  testing, 365-367
  UNIBOL
    facilities comparison, 357
    features comparison, 355
  utilities and procedures, 364
  workstation data management, 364-365
  See also UNIBOL; Unix
Operations
  logical, 36-37
  physical, 37
Optical storage, 73-75
  configurations, 76
  costs, 74-75
OS/400
  break message delivery, 341
  commands, 340
  in message handling, 341

P

Packed data conversion, 219
Packed fields, 219
Packed storage, 198
Packet assembler/disassembler (PAD), 98
  services, 98
Packet-switched network, 97-98
Pages, 22, 238
Paging, 26-28
  assets of, 79
  demand, 26
    comparison, 27
  in, 26
  out, 26
  segmented, 26
    advantages of, 26-27
    comparison, 27
  TWA buffers and, 233
  VM, 109
Parameter interface, 110
  variables, 115
Parameter passing, 137
PARM opcode, 114, 123
PATCH procedure, 169
PC/AT, 61
  internally integrated, 61
Performance tuning, 249
  statement counting profiler for, 307
Personal Services/36, 232
Physical operations, 37
  read, 37
  scan, 37
  time factors, 37
  write, 37
Placeholder file, 83-84
PLIST opcode, 114, 115
Procedural decompostion, 137
  example, 138
Processors, 5
  classes (stages) of, 91-92
  Control Storage (CSP), 5-6, 7-8
  Data Storage Controller (DSC), 6, 8, 9-10
  ELCA, 8, 10-11
  expansion feature, 61
  Local Area Network (LAN), 11
  Main Storage (MSP), 5, 6
  MLCA, 8, 10-11
  Workstation Controller (WSC), 6, 8-9
Profiled source program listing, sample, 305-306
Profiler, 303
  explained, 304
  procedures, 304
  statement counting, 307
  See also Statement counting
Profiling, 303
PROFPRT procedure, 304
  running, 306
PROFRPG procedure, 304
Program Development Manager (PDM), 344
  customizing, 344
Programmer and Operator Productivity Aid (POP), 344
  customizing, 344
  Open POP/36, 365
  PIE and, 359
Programmer Interface Environment (PIE), 359
  POP FILE function, 359, 360
  SEU editor, 359
Program profiler. *See* Profiler
Programs
  activating, 111, 128
    for first time, 129
  active-but-suspended, 111
  bundled sets of, 120
  currently executing, 110-111
  documentation, 317
  entry point, 115
  initial call, 112
  invocation of, 110, 128
  invocation stack, 113-114
  late binding, 114
  main, 110
  parameters and, 110
  quality assurance, 317-319
  SSP memory management and, 114
  subprograms, 110
  See also External program calls (EPC)
Protocol converters, 70
Public Switched Telephone Network (PSTN), 97

Q

QS36PRC library, 342
QS36RC library, 342
QSSP library, 343
Query/36, 232
QueueView utility, 209

R

RAID (Redundant Array of Inexpensive Disks), 350
RBRIDG routine, 264
  COBOL program using, 301-302
  using, 299
Read-Under-Format (RUF) workstation I/O, 257
Read/write heads, 34
Records. *See* Data records
  contention, 178
  locking, 180

speeding, addition of, 194-195
unlocking, 186
Record Waits, 223
Recursive calls, 113
Re-entrant attribute, 213
Refreshable attribute, 213
Region size, 202
Regression testing, 316
Relative record number (RRN), 155
Remote workstation controller, 68
5250 Model 12, 70
5294, 70
5394, 70
Remote Workstation Support (RWS), 68
Open RS/36 and, 367
using communication lines, 69
REORGX utility, 198
calling from a procedure syntax, 201
features, 198-199
prompt screens, 200
using, 199
REORG-YES, 102
Response time, 249-259
human interface and, 249
starting, collection, 250
subsecond, 253
tuning, 251-253
example, 253-256
user complaints, 250
Response Time Measurement Facility (RMF), 249
RETRN opcode, 114, 116, 119, 130
RGZFILE, 198
Ripple-down adds, 33, 46-48, 191
changed duplicate key value and, 192
RLABL, 264
coding CALL statements for, 300
Data Structure (DS) and, 264
defining, 299
definition list, 299
coding for, 300
field length, 299
naming, 300
using, 299-300
variable parameter length, 264
Rotational delay, 37
RPG/400, 123
AND/OR opcodes, 331
CAT operation, 264
CLOSE operation, 280
file operation codes, 328-332
DELET, 328
KLIST, 329
UPDAT, 328
WRITE, 328
multipart keys and, 329
OPEN operation, 280
SCAN operation, 264
SUBST operation, 264
See also RPG II$^1/_2$
RPG
autoreport facility, 109
CHAIN, 38
coding
for subprograms, 127
using NITU strategy, 82-86
compiler, 132
Open RS/36, 367
UNIBOL, 361
DEBUG statement, 308
development tools, 123
dump formatter, 308
file sharing, 118
internal names, 118
operations
ADD, 307
CALL, 114, 123
COMP, 307
DELET, 328
DIV, 307
DO, 307
execution cost multipliers for, 308
FREE, 114, 116
GOTO, 307
IF, 307
KLIST, 329
LOKUP, 220
MOVEA, 220
MULT, 307
PARM, 114, 123
PLIST, 114, 115
RETRN,114, 116, 119, 130
SORTA, 220
SQRT, 307
SUB, 307
UPDAT, 328
WRITE, 328
programs, 110
See also Programs
See also 400RPG; RPGII$^1/_2$
RPGAID, 309
RPGD Interactive Source Debugger, 309
RPGDUMP utility, 309
getting program task dump with, 310-311
sample output from, 310
RPG F-spec, 151
enabling record blocking with, 152
See also blocking
RPG II$^1/_2$, 123, 321
coding for subprograms, 126
compared to 400RPG, 323
compiling programs and, 321
external file descriptions and, 325-327
file names and, 124
implementation of, 324
preprocessing step, 324
program invocation time, 129-130
running called modules with, 130
screen formats, 125
UNIBOL and, 361
See also RPG
RPG/III. *See* 400RPG
RPG Reference, 322
RPGSYM procedure, 309-310
RPG-to-C translator, 361
RS/6000, 77, 88
device control functions, 292
file manipulation function, 280
language compatibility, 124
migration to, 356
string handling functions, 264
subroutines and, 263
system management functions, 288
UNIBOL on, 353-362
RTIMER utility, 215, 249-250
detail report, 251
summary report, 252
using, 250
Run Length Limited (RLL) encoding, 15
codes for, 16
Runtime library, 361

S

S/36E, 321, 339
internal organization, 343
layer, 340-341
sample, program, 334
S/36 to AS/400 Application Migration, 322
Scan operation, 38-39
Scientific Instruction Set (SIS), 210
SCO Unix, 353
Screen formats, 125-126
accessing, 79
memory-resident, 79
Seek
distance, 37
time, 37
SETCACHE utility, 245
SETCOMM procedure, 214
SETDUMP procedure, 311
SET procedure, 202
SEU, 344
PIE, editor, 359

SHOWUR
  screen, 186, 187
  utility, 181-186, 220
Single-bit errors, 16
Single-processor architecture, 11
  timeline for task-switching, 14
Single-Requester-Terminal (SRT), 172
SLCA (Single-Line Communications Adapter), 10, 70, 93, 95-96
SLOWKS utility, 199, 220
  in-memory keysort and, 224-225
  using, 201
Smart compress, 84
  for A1 and A2 spindles, 85
  for A1 spindle, 85
  disk spindle after, 84
SMF log file, 208
  key measurements, 209-210
    Communications line usage, 210, 213-215
    CSP usage, 210
    Disk seeks > 1/3, 210, 211-212
    Disk usage, 210, 211
    list of, 210
    MSP usage, 210
    Translated calls/loads, 210, 213
    UADA, 210, 212-213
  name, 208
  size, 208-209
  See also System Measurement Facility (SMF)
SMFSTART procedure, 208, 240
SMF Summary Report, 209
  Device Usage Rates page, 222
  I/O counters section, 222
  Part 1: Summary Usage, 226
  Part 2: Summary System Event Counters, 227-229
  RTIMER utility sample, 252
  swapping rates on, 211
  System Event Counters page, 211, 212
    for cache evaluation, 242
  Task Status page, 222
SMPRINT procedure, 210, 241
  summary report, 241
Snapshot dumps, 309
  SUBRTD for, 311
  See also Dumps
SNAP (SMF snapshot delay), 215, 220
  for displaying usage values, 219
  sample, utility display, 219
Software Ireland, 356-358
Software testing, 313
SORTA operation, 220
Sort file types
  ADDROUT, 189
  TAGALONG, 189-190
Source member, 309
Spindle, 83
  additional, 86
  compression, 84
  placement, 191-193
  two-spindle system, 86
  See also Disk drives; Hard disk
Spool file, 83
  extents, 83
Star network topology, 68
  twisted pair, 69
Starting Line Number (SLNO) feature, 214
Statement counting, 304
Static configuration, 346
STATUS PRT (D P) command, 341
STATUS SESSION (D) command, 340
STATUS USERS (D U) command, 162-163
STOP SYSTEM, 53
Storage
  primary, 25
  secondary, 25
Storage index, 33, 43-55
  defined, 43-44
  ISAM, 44
  keeping, open, 171-177
    with KEEPOPEN procedure, 173-174
    with MRT-NEP program, 172
  specifying maximum size of, 176
  table of index values, 171
  track numbers, 171
  See also Index
String handling assembler subroutines, 264-270
  SUBR$C, 266
  SUBR$F, 265-266
  SUBR$X, 267-268
  SUBRAT, 268
  SUBRBX, 269
  SUBRCS, 269-270
  SUBRUP, 270
Structural tests, 315
Subprograms, 110
  activation of, 111, 128, 136
    for first time, 129
  active-but-suspended, 111
  entry point, 115
  explicit deactivation of, 116
  external, deactivation, 136
  initial call, 112
  invocation of, 110, 128, 136
  invocation stack, 113-114
  late binding, 114
  parameters and, 110
  variables and, 115
  See also Programs
SUBR$C, 266
SUBR$F, 265-266
SUBR$X, 267-268
SUBRAT, 268
SUBRBX, 269
SUBRCO, 280-281
SUBRCP, 293
SUBRCS, 269-270
SUBRCT, 291-292
SUBRDT, 292
SUBRDU, 293
SUBREX, 294-295
SUBRFA, 281-286
SUBRLD, 271-274
SUBRLN, 295-296
SUBRLR, 274-276
SUBRMG, 296
Subroutines, 108
  categories of, 263-264
    COBOL, 299
    device control, 292-298
    file manipulation, 279-287
    library manipulation, 271-279
    string handling, 254-270
    system management, 287-292
  privileged mode requirement, 264
  RBRIDG and, 264
  RLABLs and, 264
  UNIBOL and, 361
  using, 333-335
  See also specific subroutines
SUBRPC, 296-297
SUBRPS, 297-298
SUBRRN, 286-287
SUBRSG, 276-279
SUBRSX, 298
SUBRSY, 288
SUBRTD, 309
  RPG code using, 312
  for snapshot dumps, 311
SUBRUF, 288-289
SUBRUL, 289-290
SUBRUP, 270
SUBRUR, 290-291
SUBRWT, 298
Swapping, 211
Swaps In, 212
Swaps Out, 212
Switched network, 97
Symbol table, 309
  creation prompt screen, 310, 311
Symbol table source member, 309

sample, 311
System/3, 3-5
System/32, 4, 5
System/34, 4, 5
memory and, 19
System/36
15 disk-file limit, 107
64K region size, 107
architectural contributions, 4
AS/400 vs., 88
objects, 342
built-in dump mechanism, 309
data file usage on, 331-332
debugging tools, 308
environment, 321, 339
genealogy, 3-6
internal components overview, 12
LAN Communications Program Product, 73
learning more about, 17
machine language instruction set, 372
models
feature comparison, 9
memory configurations, 63
overview, 60-62
See also specific models
OCL Reader/Interpreter (RI), 340
tuning, 215
upgrading, 59, 77, 91-94
costs of, 93-94, 95
Workfolder Application Facility, 75
See also S/36E
System/38, 108
demand paging, 27
externally described files and, 324
System Event Counter (SEC), 208
System management assembler subroutines, 287-292
SUBRCT, 291-292
SUBRDT, 292
SUBRSY, 288
SUBRUF, 288-289
SUBRUL, 289-290
SUBRUR, 290-291
System Measurement Facility (SMF), 92
activating, 240
averages, 207-208
CACHIQ and, 215
capturing useful data, 207-209
evaluating output, 209
expired disk lock counter, 212
File Access Counters, 222
line speed and, 208
MMETER and, 215
printout, 5
references, 226
reports
detail, 209, 251
memory use and, 233
summary, 209, 211, 212, 222, 226-229, 252
RTIMER and, 215
SNAP and, 215
tuning "recipes," 215-225
file/index blocking not helpful, 225
high CSP usage, 217-218
high disk usage, 215-216
high MSP usage, 218-220
response time degradation, 220-225
unbalanced disk usage, 216-217
using, 207-229
values, 233
See also SMF log file
System programs, 234
System queue space (SQS), 21, 29-32, 231
requirements for control blocks, 31
storage indexes and, 172
System sizing, 345-346
System Support Program (SSP), 19-20
extending TWA with, 108
FORMAT procedure, 127
operator control commands, 353-354
System work areas, 83
System workspaces, 234-235

T

TAGALONG sort, 189
disk space and, 190
time required for, 189
Tape drives
6157, 60
8809 (reel-to-reel), 60
Task
size limitation, 28
terminating/initiating, sequence, 258
Task-switching, 12-13
"fast task-switch" hardware, 13
for multiprocessor computer, 13-14
steps for, 13-14
Task Work Area (TWA), 29, 108, 233
buffers, 161, 233
paging and, 233
expansion, 29
extent messages, 233
maximum size of, 108
overcommitted memory and, 77
pre-allocating, 29
See also *SYSTASK
Task Work Spaces (TWS), 234
Testing
debugging and, 313
regression, 316
software, 313
structural, 315
Third-party
EPC products, 108
maintenance, 82, 102-103
contracts, 103
memory, 79
prices, 81-82
pitfalls, 89
upgrades, 82
vendors, 82-83
Token-Ring
connection, 356
LAN, 60
configurations, 74
network, 11
DDM and, 101
Token-Ring Network Adapter (TRNA), 72
TRACE facility, 210
turning off, 211
Transient area, 20
Translated addressing, 22
Translated Transfer Calls, 213
Translated Transfer Loads, 212
comparing to TT Calls, 213
Tree Doctor, 168
Tuning, 215
rules for, 215
Twinax
daisy-chain local workstation network, 67
defined, 66-67
lines, 66

U

Unbalanced disk usage, 216-217
UNIBOL, 263, 353
5250 devices and, 354
assembler subroutines and, 361
B-tree facility, 353
debugger, 361
disk data management, 353
dynamic file sizing, 354
environment, 323
features, 354
Migration Toolkit, 356
OCL and utility-control commands, 353-354
Open RS/36
facilities comparison, 357
feature comparison, 355

operator control commands and, 353-354
pricing, 363
print spooler, 360
product information, 370
programming with, 358-361
RPG compiler, 361
supported files, programs, features, 361
S/36 benchmarks vs., 359
security
password, 354
resource, 354
workstation data management, 353
See also Open RS/36; Unix
UNIBOL, Inc., 353-354
Universal Software. See Open RS/36
Unix, 353
AIX environments, 323, 353
file storage and, 39
Open RS/36, 323, 353, 362-369
SCO, 353
UNIBOL, 323, 353-362
Unshielded twisted-pair (UTP) cabling, 66-68
UPDAT opcode, 328
User Area Disk Access (UADA), 212-213
cache use and, 240, 241-242
high values, 233
overallocating and, 231
response time and, 249
Swaps In, 212
Swaps Out, 212
TT Loads, 212

V

V.35 interface, 98
Value Added Package Software (VASP), 3, 198
CACHE modifications, 247
RPG II½ and 400RPG and, 323
WRKSPF utility, 209
Variable index array processing, 265
Variable nucleus, 19-21, 232
components of, 20
Variables
global, 110
local, 109-110
passed to subprogram, 115
Virtual address space, 26
Virtual memory (VM), 7, 25-26, 79
ACCELER8 and, 232
EPC use and, 108-109, 232
paging, 109
S/36 peculiarities, 28-29
task size and, 28
See also Workspaces
Virtual page table, 20

W

Workspaces, 28
disk file, 28
Workstation controller storage, 66
Workstation Controller (WSC)
code, 8
processor, 6, 8-9
Workstation Data Management, 231
Workstation expansion features, 66
Workstations, 66-70
5250, 70
configurations of, 67
I/O, 125, 129
local, 66-68
remote, 68-70
remote, controller, 68
Write-once-read-many (WORM) optical disks, 74, 75
WRITE opcode, 328
WRKSPLF command, 341

X

X.21 interface, 98
X.25, interface, 98
XCACHE program, 245

Z

Zoned decimal format, 332-333
Zoned storage, 198

Also Published by *NEWS 3X/400*

AS/400 POWER TOOLS

Edited by Dan Riehl, a ***NEWS 3X/400*** *technical editor*

This one easy-to-use source unlocks the secrets of time-saving shortcuts, gives you solutions and tips from the experts. Edited for professionals, all the best AS/400 tips and techniques are conveniently organized, indexed, and cross-referenced, with diskette containing all programming code in your choice of format. Tape or cartridge optional extra.

713 pages, 24 chapters, 7"x9", $129.

CONTENTS: HLL Tips • The OPNQRYF Command • String Manipulation • Display Files • Data Files • Data Areas • Data Queues • Messages • Graphics • Debugging • Documentation • System Configuration • Group Jobs • Managing Output Queues • Working with Objects • Change Management • Managing Resources • Disaster Recovery • Security • Job Accounting • PC Support • OfficeVision/400 • Communications • System Programming • Appendices

C FOR RPG PROGRAMMERS

By Jennifer Hamilton, ***NEWS 3X/400*** *author*

Expand your RPG knowledge to include the currently demanded C programming skills. Because this book is written from the perspective of an RPG programmer, you can easily adapt to C without returning to the basics. *C for RPG Programmers* includes side-by-side coding examples written in both C and RPG to aid comprehension and understanding, clear identification of unique C constructs, and a comparison of RPG opcodes to equivalent C concepts. Also, both the novice and experienced C programmer will benefit from the many tips and examples covering the use of C/400.

250 pages, 23 chapters, 7"x9", $69

CONTENTS: Overview • Data Types • Expressions • Statements • Arrays • Structures & Unions • Functions, Scope, and Storage Class • Separate Compilation and Linkage • Parameter Passing • Pointers • Pointer Arithmetic • Using Pointers • The Preprocessor • Type Conversions and Definitions • Stream I/O • Error Handling • Dynamic Storage Allocation • Recursion • Programming Style • The Extended Program Model • Interlanguage Calls • C/400 File I/O • System C/400

COMMON-SENSE C
Advice and Warnings for C and C++ Programmers

by Paul Conte, a ***NEWS 3X/400*** *technical editor*

C is not a programming language without risks. Even C experts rely on careful programming, lint filters, and good debuggers to handle problems common to C. This book helps prevent such problems by showing how C programmers get themselves into trouble. The author draws on more than 15 years of experience to help you avoid C's pitfalls. And he suggests how to manage C and C++ application development.

96 pages, 9 chapters, 7"x9", $24.95

Contents: Preface • Introduction • Common Mistakes and How to Avoid Them • Foolproof Statement and Comment Syntax • Hassle-Free Arrays and Strings • Simplified Variable Declarations • Practical Pointers • Macros and Miscellaneous Pitfalls • C++ • Managing C and C++ Development • Bibliography • Index • Appendix (C Coding Suggestions)

IMPLEMENTING AS/400 SECURITY

*By Wayne Madden, a **NEWS 3X/400** technical editor*

All of the hard work you put into your MIS operation is endangered if system security is substandard or outdated. But everything you need is here to achieve or upgrade security of your AS/400 without struggling for hours searching through manuals and wondering if you have all the bases covered. Expert Wayne Madden shares his know-how, makes recommendations, and leads you through all the necessary steps in one easy-to-read volume. Security implementation utilities are included on accompanying 3-1/2" PC diskette.

286 pages, 13 chapters, 7"x9", $99.

CONTENTS: Security at the System Level • The Facts About User Profiles • Object Authorization • Database Security • Network Security • Evaluating Your Current Strategy • Establishing and Controlling System Access • Building Object and Role Authorizations • Security Implementation Examples • Is Your Strategy Working? • Status Auditing • Event Auditing • Appendices, References, Figures and Tables

LOCAL AREA NETWORKS: A Guide for Midrange Decision Makers

Edited by Teresa Elms, CDP, president of Elms Information Services Group

This easy-to-read book contains everything you need to know to make the right decision on LANS in the fast-changing connectivity market. Includes product evaluations with competitive strengths and weaknesses and 5-year forecast. Provides decision support for installing a LAN and direction on LAN strategy. Numerous charts and tables show current data on LAN products and market information. Here's vital information not available anywhere else that could save you weeks of research time and thousands of dollars. LAN Glossary included while supply lasts.

169 pages, 10 chapters, 8-1/2"x11", $99.

CONTENTS: Executive Summary • Local Area Networks: An Introduction • LAN Technologies • LAN Implementations: ARCNet, Ethernet and Token Ring • Managing LANS in a Midrange Environment • LANS Versus Multiuser Minicomputers • Five-Year Forecast: Analysis of LAN Trends in the Midrange Industry • The AS/400 Token-Ring LAN • Novell Netware • LANtastic from Artisoft • Further Reading

S/36 POWER TOOLS

*Edited by Chuck Lundgren, a **NEWS 3X/400** technical editor*

Five years' worth of articles, tips, and programs published in *NEWS 3X/400* from 1986 to October 1990, including more than 280 programs and procedures. Extensively cross-

referenced for fast and easy problem-solving, and complete with diskette containing all the programming code. Winner of an Award of Achievement from the Society of Technical Communications, 1992.

747 pages, 20 chapters, 7"x9", $89.

CONTENTS: Backup and Restore • Communications • Data Conversion, Edits, and Validation • DFU, SDA, and SEU • Diskettes • DisplayWrite • Documentation • Files • Folders • IDDU and Query/36 • Libraries • MAPICS • Performance • POP • Printers • Programming • Security • System • Tapes • Workstations • Appendix

THE STARTER KIT FOR THE AS/400: 18 Fundamental Concepts

*by Wayne Madden, a **NEWS 3X/400** technical editor*

Here is an indispensable guide for novice to intermediate programmers and system operators. It takes the intimidation factor out of getting started on the AS/400. Written in friendly and understandable yet concise language, it's great as an easy-to-use training tool. Also a perfect way to fill in the knowledge gap if you've learned on the job without essential background. Response to this long-needed book has been tremendous — don't be without it if you're training or just getting started with the AS/400.

Begins with a step-by-step guide to setting up an AS/400. Introduces AS/400 concepts including file structure, handling output queues, OS/400 commands, using file overrides, and the ins-and-outs of AS/400 work management.

220 pages, 18 chapters, 7x9", $89.

CONTENTS: List of Figures • Introduction • Before the Power is On • That Important First Session • Access Made Easy • Public Authorities • Print Files and Job Logs • Understanding Output Queues • Defining a Subsystem • Where Jobs Come From • Demystifying Routing • File Structures • File Overrides • Logical Files • File Sharing • OS/400 Commands • CL Programming • OPNQRYF Fundamentals • Keeping Up with the Past • OS/400 Data Areas • Bibliography • Index

---

FOR MORE INFORMATION
OR TO PLACE AN ORDER, CONTACT:

NEWS 3X/400
Duke Communications International
221 E. 29th Street
Loveland, CO 60539
(800) 621-1544
(303) 663-4700
Fax: (303) 667-2321