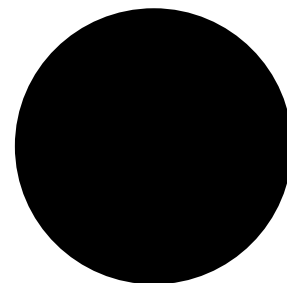
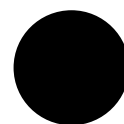


Concepts



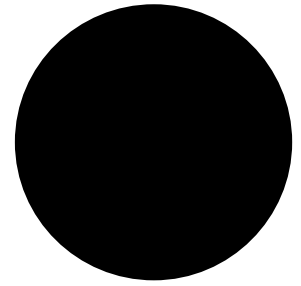
Version 2.0



GEOS Software Development Kit Library

Version 2.0

Concepts



Initial Edition, Unrevised and Unexpanded

Geoworks, Inc.
Alameda, CA



Geoworks provides this publication "as is" without warranty of any kind, either express or implied, including but not limited to the implied warranties of merchantability or fitness for a particular purpose. Geoworks may revise this publication from time to time without notice. Geoworks does not promise support of this documentation. Some states or jurisdictions do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

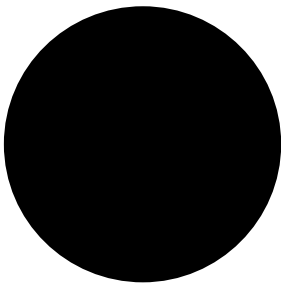
Copyright © 1994 by Geoworks, Incorporated.
All rights reserved. Published 1994
Printed in the United States of America

Geoworks®, Geoworks Ensemble®, Ensemble®, GEOS®, PC/GEOS®, GeoDraw®, GeoManager®, GeoPlanner®, GeoFile®, GeoDex® and GeoComm® are registered trademarks of Geoworks in the United States and Other countries.

Geoworks® Pro, PEN./GEOS, Quick Start, GeoWrite, GeoBanner, GeoCrypt, GeoCalc, GeoDOS, Geoworks® Writer, Geoworks® Desktop, Geoworks® Designer, Geoworks® Font Library, Geoworks® Art Library, Geoworks® Escape, Lights Out, and Simply Better Software are trademarks of Geoworks in the United States and other countries.

Trademarks and service marks not listed here are the property of companies other than Geoworks. Every effort has been made to treat trademarks and service marks in accordance with the United States Trademark Association's guidelines. Any omissions are unintentional and should not be regarded as affecting the validity of any trademark or service mark.

Contents



List of C Examples..... 25
List of Figures..... 29

Volume 1

1 Introduction..... 35

1.1 Overview of The Documentation..... CIntro : 37

 1.1.1 What You Will Learn..... CIntro : 37

 1.1.2 What You Are Expected To Know..... CIntro : 37

 1.1.3 Roadmap to the Development Kit..... CIntro : 38

 1.1.4 Typographical Cues..... CIntro : 39

1.2 Chapters in the Books..... CIntro : 40

 1.2.1 The Concepts Book CIntro : 40

 1.2.2 The Object Reference Book..... CIntro : 43

 1.2.3 The Tools Reference Manual..... CIntro : 47

 1.2.4 The Esp Book..... CIntro : 48

1.3 Suggestions for Study CIntro : 48

2 Building Your Application 51

2.1 What Everyone Should Read..... CBuild : 53

2.2 Topics Listing CBuild : 54

 2.2.1 Defining Your User Interface CBuild : 54

 2.2.2 Providing Other User Interface CBuild : 56

 2.2.3 Documents and Data Structures CBuild : 56

 2.2.4 Accessing Hardware..... CBuild : 57

 2.2.5 Programming Topics..... CBuild : 58

 2.2.6 Other Topics..... CBuild : 58

3	System Architecture	61
3.1	GEOS Overview	CArch : 63
3.2	The System Architecture	CArch : 64
	3.2.1 Applications.....	CArch : 66
	3.2.2 Libraries	CArch : 66
	3.2.3 The Kernel	CArch : 66
	3.2.4 Device Drivers	CArch : 67
	3.2.5 The User Interface.....	CArch : 67
3.3	Object-Oriented Programming	CArch : 68
	3.3.1 Objects, Messages, and Methods	CArch : 69
	3.3.2 Classes and Inheritance	CArch : 73
3.4	Multitasking and Multithreading	CArch : 75
3.5	The GEOS User Interface.....	CArch : 76
	3.5.1 The Generic User Interface	CArch : 77
	3.5.2 The Scalable User Interface.....	CArch : 79
	3.5.3 Windows and Window Management	CArch : 79
	3.5.4 Input	CArch : 80
	3.5.5 Menus and Dialog Boxes	CArch : 80
	3.5.6 Scrolling Views.....	CArch : 84
	3.5.7 Visible Object Classes.....	CArch : 86
	3.5.8 Geometry Manager.....	CArch : 86
	3.5.9 Lists.....	CArch : 86
	3.5.10 Other Gadgets	CArch : 87
	3.5.11 Managing Documents and Files	CArch : 88
	3.5.12 Multiple Document Interface.....	CArch : 88
	3.5.13 Clipboard and Quick-Transfer	CArch : 88
	3.5.14 General Change Notification	CArch : 89
	3.5.15 Help Object.....	CArch : 89

3.6	System Services.....	CArch : 90
	3.6.1 Memory	CArch : 90
	3.6.2 Virtual Memory	CArch : 93
	3.6.3 Local Memory and Object Blocks	CArch : 93
	3.6.4 Item Database Library.....	CArch : 94
	3.6.5 Graphics System	CArch : 95
	3.6.6 Text	CArch : 98
	3.6.7 Print Spooler and Printing.....	CArch : 99
	3.6.8 Timers	CArch : 99
	3.6.9 Streams.....	CArch : 100
	3.6.10 Math support	CArch : 100
	3.6.11 International Support	CArch : 100
3.7	Libraries.....	CArch : 101
3.8	Device Drivers	CArch : 102
4	First Steps: Hello World.....	105
4.1	Application Structure.....	CGetSta : 107
4.2	Hello World	CGetSta : 108
	4.2.1 Features of Hello World.....	CGetSta : 109
	4.2.2 Strategy and Internals.....	CGetSta : 109
	4.2.3 Naming Conventions	CGetSta : 112
4.3	Geode Parameters File	CGetSta : 114
4.4	The Source File and Source Code.....	CGetSta : 116
	4.4.1 Inclusions and Global Variables...	CGetSta : 117
	4.4.2 The Process Object.....	CGetSta : 119
	4.4.3 UI Objects	CGetSta : 120
	4.4.4 Code and Message Handlers.....	CGetSta : 128
4.5	Exercises and Suggestions	CGetSta : 136

5	GEOS Programming	139
5.1	Basic Data Types and Structures	CCoding : 141
	5.1.1 Records and Enumerated Types...	CCoding : 142
	5.1.2 Handles and Pointers	CCoding : 144
	5.1.3 Fixed Point Structures	CCoding : 147
5.2	Goc and C.....	CCoding : 149
	5.2.1 Goc File Types	CCoding : 149
	5.2.2 Conditional Code in Goc	CCoding : 151
	5.2.3 Macros in Goc	CCoding : 151
	5.2.4 Using Routine Pointers in Goc	CCoding : 152
5.3	The GEOS Object System.....	CCoding : 154
	5.3.1 GEOS Terminology.....	CCoding : 154
	5.3.2 Object Structures	CCoding : 158
	5.3.3 The GEOS Message System.....	CCoding : 180
5.4	Using Classes and Objects	CCoding : 182
	5.4.1 Defining a New Class or Subclass	CCoding : 184
	5.4.2 Non-relocatable Data.....	CCoding : 203
	5.4.3 Defining Methods	CCoding : 204
	5.4.4 Declaring Objects	CCoding : 209
	5.4.5 Sending Messages.....	CCoding : 219
	5.4.6 Managing Objects	CCoding : 226
6	Applications and Geodes.....	239
6.1	Geodes	CAppl : 242
	6.1.1 Geode Components and Structures	CAppl : 244
	6.1.2 Launching an Application.....	CAppl : 247
	6.1.3 Shutting Down an Application.....	CAppl : 249
	6.1.4 Saving and Restoring State.....	CAppl : 250

6.1.5	Using Other Geodes	CAppl : 253
6.1.6	Writing Your Own Libraries.....	CAppl : 257
6.1.7	Working with Geodes.....	CAppl : 259
6.1.8	Geode Protocols and Release Levels	CAppl : 261
6.1.9	Temporary Geode Memory	CAppl : 264
6.2	Creating Icons	CAppl : 265
6.2.1	The Token Database	CAppl : 265
6.2.2	Managing the Token Database File	CAppl : 267
6.3	Saving User Options.....	CAppl : 269
6.3.1	Saving Generic Object Options	CAppl : 269
6.3.2	The GEOS.INI File.....	CAppl : 271
6.4	General System Utilities	CAppl : 277
6.4.1	Changing the System Clock	CAppl : 277
6.4.2	Using Timers	CAppl : 277
6.4.3	System Statistics and Utilities	CAppl : 278
6.4.4	Shutting the System Down	CAppl : 279
6.5	The Error-Checking Version	CAppl : 280
6.5.1	Adding EC Code to Your Program...	CAppl : 281
6.5.2	Special EC Routines.....	CAppl : 282
6.6	Inter-Application Communication	CAppl : 283
6.6.1	IACP Overview	CAppl : 284
6.6.2	GenApplicationClass Behavior	CAppl : 286
6.6.3	Messages Across an IACP Link	CAppl : 287
6.6.4	Being a Client.....	CAppl : 288
6.6.5	Being a Server	CAppl : 292
7	The Clipboard.....	297
7.1	Overview	CClipb : 299

	7.1.1	Cut, Copy, and Paste	CClipb : 300
	7.1.2	Quick-Transfer	CClipb : 301
7.2		Transfer Data Structures	CClipb : 302
	7.2.1	The Transfer VM File Format	CClipb : 303
	7.2.2	ClipboardItemFormatInfo.....	CClipb : 305
	7.2.3	Transfer Data Structures.....	CClipb : 307
	7.2.4	Clipboard Item Formats	CClipb : 308
7.3		Using The Clipboard	CClipb : 309
	7.3.1	Registering with the Clipboard.....	CClipb : 311
	7.3.2	Managing the Edit Menu	CClipb : 311
	7.3.3	The GenEditControl.....	CClipb : 314
	7.3.4	Handling Cut and Copy	CClipb : 316
	7.3.5	Handling Paste	CClipb : 319
	7.3.6	Unregistering with the Clipboard...	CClipb : 322
	7.3.7	Implementing Undo	CClipb : 322
	7.3.8	Transfer File Information	CClipb : 322
	7.3.9	Undoing a Clipboard Change.....	CClipb : 323
7.4		Using Quick-Transfer	CClipb : 323
	7.4.1	Supporting Quick-Transfer	CClipb : 324
	7.4.2	Quick-Transfer Procedure.....	CClipb : 325
	7.4.3	Quick-Transfer Data Structures	CClipb : 326
	7.4.4	Source Object Responsibility	CClipb : 327
7.5		Shutdown Issues	CClipb : 331
8		Localization	333
8.1		Localization Goals	CLocal : 335
8.2		How To Use Localization	CLocal : 337
8.3		Preparing for ResEdit.....	CLocal : 337

8.4	International Formats	CLocal : 340
8.4.1	Number and Measure.....	CLocal : 341
8.4.2	Currency.....	CLocal : 343
8.4.3	Quotation Marks	CLocal : 343
8.4.4	Dates and Times.....	CLocal : 344
8.4.5	Filters for Formats	CLocal : 346
8.5	Lexical Functions.....	CLocal : 348
8.5.1	Comparing Strings	CLocal : 348
8.5.2	String Length and Size	CLocal : 349
8.5.3	Casing.....	CLocal : 349
8.5.4	Character Categories.....	CLocal : 349
8.5.5	Lexical Values	CLocal : 350
8.5.6	DOS Text & Code Pages.....	CLocal : 350
9	General Change Notification	353
9.1	Design Goals	CGCN : 355
9.2	The Mechanics of GCN	CGCN : 356
9.3	System Notification.....	CGCN : 357
9.3.1	Registering for System Notification.....	CGCN : 357
9.3.2	Handling System Notification.....	CGCN : 360
9.3.3	Removal from a System List.....	CGCN : 364
9.4	Application Local GCN Lists.....	CGCN : 364
9.4.1	Creating Types and Lists.....	CGCN : 366
9.4.2	Registering for Notification.....	CGCN : 366
9.4.3	Handling Application Notification..	CGCN : 368
9.4.4	Removal from Application Lists	CGCN : 371
10	The GEOS User Interface	373
10.1	The GUI	CUIOver : 375

10.2	The GEOS User Interface.....	CUIOver : 376
10.3	Using the Generic Classes	CUIOver : 378
	10.3.1 The Generic Class Tree.....	CUIOver : 380
	10.3.2 Creating a Generic Object Tree ...	CUIOver : 385
10.4	Using the Visible Classes	CUIOver : 385
	10.4.1 Visible Objects and the GenView	CUIOver : 386
	10.4.2 The Visible Object Document	CUIOver : 386
	10.4.3 Visible Object Abilities.....	CUIOver : 387
	10.4.4 The Vis Class Tree.....	CUIOver : 388
	10.4.5 Creating a Visible Object Tree	CUIOver : 389
	10.4.6 Working with Visible Object Trees	CUIOver : 390
10.5	A UI Example.....	CUIOver : 391
	10.5.1 What TicTac Illustrates.....	CUIOver : 391
	10.5.2 What TicTac Does	CUIOver : 392
	10.5.3 The Structure of TicTac	CUIOver : 392
	10.5.4 TicTacBoard Specifics.....	CUIOver : 400
	10.5.5 TicTacPiece Specifics	CUIOver : 404
11	Input.....	417
11.1	Input Flow.....	CInput : 419
	11.1.1 Devices and Drivers	CInput : 421
	11.1.2 Input Manager and GenSystem	CInput : 422
	11.1.3 Input Events	CInput : 422
	11.1.4 Input Hierarchies.....	CInput : 423
11.2	Mouse Input	CInput : 424
	11.2.1 Mouse Events	CInput : 425
	11.2.2 Gaining the Mouse Grab	CInput : 431
	11.2.3 Large Mouse Events.....	CInput : 433
	11.2.4 Setting the Pointer Image.....	CInput : 434

11.3	Keyboard Input.....	CInput : 437
	11.3.1 Keyboard Input Flow	CInput : 437
	11.3.2 Keyboard Events	CInput : 438
11.4	Pen Input and Ink	CInput : 442
	11.4.1 Ink Data Structures	CInput : 443
	11.4.2 Ink Input Flow	CInput : 444
11.5	Input Hierarchies.....	CInput : 448
	11.5.1 The Three Hierarchies	CInput : 449
	11.5.2 Common Hierarchy Basics	CInput : 449
	11.5.3 Using Focus	CInput : 453
	11.5.4 Using Target	CInput : 456
	11.5.5 Using Model	CInput : 460
	11.5.6 Extending the Hierarchies	CInput : 462
12	Managing UI Geometry	465
12.1	Geometry Manager Overview.....	CGeom : 467
	12.1.1 Geometry Manager Features.....	CGeom : 468
	12.1.2 How Geometry Is Managed	CGeom : 469
12.2	Arranging Your Generic Objects	CGeom : 472
	12.2.1 General Geometry Rules	CGeom : 472
	12.2.2 Orienting Children	CGeom : 474
	12.2.3 Justifying and Centering Children	CGeom : 478
	12.2.4 Sizing Objects	CGeom : 480
	12.2.5 Outlining the Composite	CGeom : 487
	12.2.6 Using Monikers.....	CGeom : 487
	12.2.7 Using Custom Child Spacing.....	CGeom : 490
	12.2.8 Allowing Children to Wrap.....	CGeom : 492
	12.2.9 Object Placement	CGeom : 493
12.3	Positioning and Sizing Windows.....	CGeom : 497

	12.3.1 Window Positioning.....	CGeom : 499
	12.3.2 Determining Initial Size	CGeom : 500
	12.3.3 On-Screen Behavior	CGeom : 501
	12.3.4 Window Look and Feel.....	CGeom : 502
13	Sound Library	503
13.1	Goals and Motives	CSound : 505
13.2	Playing UI Sounds	CSound : 505
13.3	Representing Tones	CSound : 506
13.4	Single Notes	CSound : 510
13.5	Declaring Music Buffers	CSound : 513
13.6	Playing Music Buffers	CSound : 516
13.7	Playing Very Large Music Buffers	CSound : 518
13.8	Playing Sampled Sounds.....	CSound : 518
13.9	Grabbing the Sound Exclusive	CSound : 520
13.10	Simulating Musical Instruments	CSound : 521
	13.10.1Acoustics In Brief.....	CSound : 521
	13.10.2Simple Instrument Description....	CSound : 522
	13.10.3Advanced Description	CSound : 523

Volume 2

14	Handles	535
14.1	Design Philosophy	CHandle : 537
14.2	The Global Handle Table.....	CHandle : 539
14.3	Local Handles	CHandle : 539

15	Memory Management	541
15.1	Design Philosophy	CMemory : 543
15.2	The Structure of Memory	CMemory : 544
	15.2.1 Expanded/Extended Memory	CMemory : 544
	15.2.2 Main Memory	CMemory : 545
15.3	Using Global Memory	CMemory : 552
	15.3.1 Memory Etiquette	CMemory : 552
	15.3.2 Requesting Memory	CMemory : 554
	15.3.3 Freeing Memory	CMemory : 555
	15.3.4 Accessing Data in a Block	CMemory : 555
	15.3.5 Accessing Data: An Example	CMemory : 556
	15.3.6 Data-Access Synchronization	CMemory : 558
	15.3.7 Retrieving Block Information	CMemory : 561
	15.3.8 The Reference Count	CMemory : 563
15.4	malloc()	CMemory : 564
16	Local Memory	567
16.1	Design Philosophy	CLMem : 569
16.2	Structure of a Local Memory Heap	CLMem : 570
	16.2.1 The Local Heap	CLMem : 571
	16.2.2 Chunks and Chunk Handles	CLMem : 572
	16.2.3 Types of LMem Heaps	CLMem : 573
16.3	Using Local Memory Heaps	CLMem : 577
	16.3.1 Creating a Local Heap	CLMem : 577
	16.3.2 Using Chunks	CLMem : 579
	16.3.3 Contracting the LMem Heap	CLMem : 581
	16.3.4 Example of LMem Usage	CLMem : 581
16.4	Special LMem Uses	CLMem : 583

	16.4.1	Chunk Arrays	CLMem : 584
	16.4.2	Element Arrays.....	CLMem : 595
	16.4.3	Name Arrays.....	CLMem : 602
17		File System	609
17.1		Design Philosophy	CFile : 611
17.2		File System Overview.....	CFile : 613
17.3		Disks and Drives.....	CFile : 616
	17.3.1	Accessing Drives.....	CFile : 616
	17.3.2	Accessing Disks.....	CFile : 619
17.4		Directories and Paths.....	CFile : 631
	17.4.1	Standard Paths	CFile : 632
	17.4.2	Current Path and Directory Stack....	CFile : 636
	17.4.3	Creating and Deleting Directories....	CFile : 639
17.5		Files	CFile : 641
	17.5.1	DOS Files and GEOS Files.....	CFile : 641
	17.5.2	Files and File Handles	CFile : 643
	17.5.3	GEOS Extended Attributes	CFile : 643
	17.5.4	File Utilities	CFile : 653
	17.5.5	FileEnum()	CFile : 655
	17.5.6	Bytewise File Operations	CFile : 661
18		Virtual Memory	671
18.1		Design Philosophy	CVM : 673
18.2		VM Structure	CVM : 674
	18.2.1	The VM Manager	CVM : 675
	18.2.2	VM Handles.....	CVM : 676
	18.2.3	Virtual Memory Blocks	CVM : 677

	18.2.4 VM File Attributes	CVM : 681
18.3	Using Virtual Memory	CVM : 683
	18.3.1 How to Use VM	CVM : 683
	18.3.2 Opening or Creating a VM File.....	CVM : 684
	18.3.3 Changing VM File Attributes.....	CVM : 687
	18.3.4 Creating and Freeing Blocks	CVM : 687
	18.3.5 Attaching Memory Blocks.....	CVM : 689
	18.3.6 Accessing and Altering VM Blocks....	CVM : 690
	18.3.7 VM Block Information.....	CVM : 692
	18.3.8 Updating and Saving Files.....	CVM : 693
	18.3.9 Closing Files.....	CVM : 695
	18.3.10The VM File's Map Block.....	CVM : 696
	18.3.11File-Access Synchronization.....	CVM : 697
	18.3.12Other VM Utilities	CVM : 699
18.4	VM Chains.....	CVM : 700
	18.4.1 Structure of a VM Chain	CVM : 701
	18.4.2 VM Chain Utilities	CVM : 703
18.5	Huge Arrays.....	CVM : 705
	18.5.1 Structure of a Huge Array	CVM : 706
	18.5.2 Basic Huge Array Routines	CVM : 708
	18.5.3 Huge Array Utilities	CVM : 713
19	Database Library	717
19.1	Design Philosophy.....	CDB : 719
19.2	Database Structure	CDB : 720
	19.2.1 DB Items	CDB : 720
	19.2.2 DB Groups	CDB : 721
	19.2.3 Allocating Groups and Items	CDB : 722
	19.2.4 Ungrouped DB Items.....	CDB : 723

	19.2.5 The DB Map Item	CDB : 723
19.3	Using Database Routines	CDB : 724
	19.3.1 General Rules to Follow.....	CDB : 724
	19.3.2 Allocating and Freeing Groups	CDB : 725
	19.3.3 Allocating and Freeing Items	CDB : 725
	19.3.4 Accessing DB Items.....	CDB : 726
	19.3.5 Resizing DB Items	CDB : 727
	19.3.6 Setting and Using the Map Item.....	CDB : 728
	19.3.7 Routines for Ungrouped Items.....	CDB : 729
	19.3.8 Other DB Utilities	CDB : 730
19.4	The Cell Library	CDB : 731
	19.4.1 Structure and Design.....	CDB : 732
	19.4.2 Using the Cell Library	CDB : 734
20	Parse Library	741
20.1	Parse Library Behavior	CParse : 743
	20.1.1 The Scanner	CParse : 745
	20.1.2 The Parser	CParse : 753
	20.1.3 Evaluator.....	CParse : 759
	20.1.4 Formatter	CParse : 761
20.2	Parser Functions	CParse : 761
	20.2.1 Internal Functions	CParse : 762
	20.2.2 External Functions.....	CParse : 765
20.3	Coding with the Parse Library.....	CParse : 766
	20.3.1 Parsing a String.....	CParse : 766
	20.3.2 Evaluating a Token Sequence	CParse : 767
	20.3.3 Formatting a Token Sequence.....	CParse : 770
21	Using Streams	771

21.1	Using Streams: The Basics.....	CStream : 773
	21.1.1 Initializing a Stream.....	CStream : 775
	21.1.2 Blocking on Read or Write.....	CStream : 777
	21.1.3 Writing Data to a Stream	CStream : 778
	21.1.4 Reading Data from a Stream.....	CStream : 779
	21.1.5 Shutting Down a Stream.....	CStream : 781
	21.1.6 Miscellaneous Functions	CStream : 782
21.2	Using the Serial Ports.....	CStream : 782
	21.2.1 Initializing a Serial Port.....	CStream : 783
	21.2.2 Communicating.....	CStream : 787
	21.2.3 Closing a Serial Port.....	CStream : 788
21.3	Using the Parallel Ports.....	CStream : 789
	21.3.1 Initializing a Parallel Port.....	CStream : 789
	21.3.2 Communicating.....	CStream : 790
	21.3.3 Closing a Parallel Port	CStream : 790
22	PCCom Library	793
22.1	PCCom Library Abilities	795
22.2	What To Do	795
22.3	Staying Informed.....	796
23	Graphics Environment.....	799
23.1	Graphics Road Map	CGraph : 801
	23.1.1 Chapter Structure.....	CGraph : 801
	23.1.2 Vocabulary	CGraph : 803
23.2	Graphics Goals.....	CGraph : 806
23.3	Graphics Architecture	CGraph : 807
23.4	How To Use Graphics	CGraph : 808

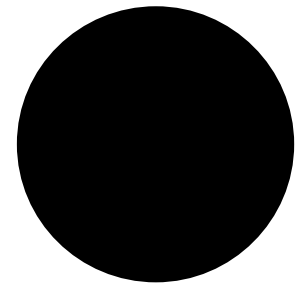
23.5	Coordinate Space	CGraph : 810
	23.5.1 Standard Coordinate Space	CGraph : 811
	23.5.2 Coordinate Transformations	CGraph : 812
	23.5.3 Precise Coordinates	CGraph : 818
	23.5.4 Device Coordinates	CGraph : 819
	23.5.5 Larger Document Spaces	CGraph : 823
	23.5.6 Current Position	CGraph : 824
23.6	Graphics State	CGraph : 825
	23.6.1 GState Contents	CGraph : 826
	23.6.2 Working with GStates	CGraph : 827
23.7	Working With Bitmaps	CGraph : 828
23.8	Graphics Strings	CGraph : 832
	23.8.1 Storage and Loading	CGraph : 832
	23.8.2 Special Drawing Commands	CGraph : 834
	23.8.3 Declaring a GString Statically	CGraph : 836
	23.8.4 Creating GStrings Dynamically	CGraph : 840
	23.8.5 Drawing and Scanning	CGraph : 843
	23.8.6 Editing GStrings Dynamically	CGraph : 846
	23.8.7 Parsing GStrings	CGraph : 847
23.9	Graphics Paths	CGraph : 849
23.10	Working With Video Drivers	CGraph : 852
	23.10.1 Kernel Routines	CGraph : 852
	23.10.2 Direct Calls to the Driver	CGraph : 853
23.11	Windowing and Clipping	CGraph : 854
	23.11.1 Palettes	CGraph : 854
	23.11.2 Clipping	CGraph : 854
	23.11.3 Signalling Updates	CGraph : 855

24	Drawing Graphics	857
24.1	Drawing Goals	CShapes : 859
24.2	Shapes	CShapes : 859
	24.2.1 Dots	CShapes : 860
	24.2.2 Lines	CShapes : 861
	24.2.3 Rectangles.....	CShapes : 862
	24.2.4 Ellipses.....	CShapes : 862
	24.2.5 Elliptical Arcs.....	CShapes : 863
	24.2.6 Three-Point Arcs	CShapes : 864
	24.2.7 Rounded Rectangles.....	CShapes : 865
	24.2.8 Polylines and Polygons	CShapes : 866
	24.2.9 Bézier Curves and Splines.....	CShapes : 867
	24.2.10 Drawing Bitmaps.....	CShapes : 870
	24.2.11 Paths	CShapes : 872
	24.2.12 Regions	CShapes : 873
	24.2.13 Text.....	CShapes : 876
24.3	Shape Attributes	CShapes : 885
	24.3.1 Color.....	CShapes : 886
	24.3.2 Patterns and Hatching.....	CShapes : 891
	24.3.3 Mix Mode.....	CShapes : 895
	24.3.4 Masks	CShapes : 897
	24.3.5 Line-Specific Attributes.....	CShapes : 899
A	Machine Architecture	903
A.1	History of the 80x86	CHardw : 905
A.2	8086 Architecture Overview	CHardw : 906
	A.2.1 Memory	CHardw : 906
	A.2.2 Registers	CHardw : 908
	A.2.3 The Prefetch Queue	CHardw : 909

	A.2.4 Inherent Optimizations.....	CHardw : 910
B	Threads and Semaphores 875.....	911
B.1	Multitasking Goals	CMultit : 913
B.2	Two Models of Multitasking	CMultit : 914
	B.2.1 Cooperative Multitasking	CMultit : 914
	B.2.2 Preemptive Multitasking	CMultit : 915
B.3	GEOS Multitasking.....	CMultit : 916
	B.3.1 GEOS Threads.....	CMultit : 917
	B.3.2 Context Switches	CMultit : 918
	B.3.3 Thread Scheduling	CMultit : 919
	B.3.4 Applications and Threads	CMultit : 920
B.4	Using Multiple Threads	CMultit : 921
	B.4.1 How GEOS Threads Are Created .	CMultit : 921
	B.4.2 Managing Priority Values.....	CMultit : 923
	B.4.3 Handling Errors in a Thread	CMultit : 924
	B.4.4 When a Thread Is Finished.....	CMultit : 925
B.5	Synchronizing Threads	CMultit : 926
	B.5.1 Semaphores: The Concept.....	CMultit : 926
	B.5.2 Semaphores In GEOS.....	CMultit : 928
C	Libraries.....	933
C.1	Design Philosophy	CLibr : 935
C.2	Library Basics.....	CLibr : 936
C.3	The Library Entry Point.....	CLibr : 937
C.4	Exported Routines and Classes	CLibr : 940
C.5	Header Files	CLibr : 941
C.6	Compiler Directives	CLibr : 941

D	The Math Library	943
D.1	Basic Math Functions	CMath : 945
	D.1.1 Algebraic Functions	CMath : 947
	D.1.2 Transcendental Functions	CMath : 949
	D.1.3 Random Number Generation	CMath : 950
D.2	Conversions to Other Types	CMath : 951
D.3	Float Formats	CMath : 960
	D.3.1 System-defined Formats	CMath : 960
	D.3.2 User-defined Formats	CMath : 963
D.4	Direct FP Operations	CMath : 964
	D.4.1 Floating Point Numbers	CMath : 965
	D.4.2 The Floating Point Stack	CMath : 966
	D.4.3 Floating Point Math Routines	CMath : 971
	<i>Credits</i>	979
	<i>Index</i>	IX : 1

List of C Examples



Code Display 4-1	The Hello World Parameters File	CGetSta : 114
Code Display 4-2	Inclusion Files and Global Variables	CGetSta : 118
Code Display 4-3	Hello World's Process Object	CGetSta : 119
Code Display 4-4	Hello World's Application Object	CGetSta : 122
Code Display 4-5	Hello World's Primary and View Objects ...	CGetSta : 123
Code Display 4-6	The Hello World Menu	CGetSta : 125
Code Display 4-7	The Hello World Dialog Box and Its Triggers	CGetSta : 127
Code Display 4-8	Constant and Routine Definition	CGetSta : 129
Code Display 4-9	Messages from the View	CGetSta : 130
Code Display 4-10	MSG_META_EXPOSED Handler	CGetSta : 132
Code Display 4-11	Handlers for MSG_HELLO_... ..	CGetSta : 135
Code Display 5-1	Flag Records and ByteEnums	CCoding : 143
Code Display 5-2	GEOS Data Structures	CCoding : 147
Code Display 5-3	Using ProcCallFixedOrMovable_cdecl()	CCoding : 153
Code Display 5-4	TicTac's New Game Trigger	CCoding : 175
Code Display 5-5	Goc Keywords	CCoding : 183
Code Display 5-6	Defining Classes	CCoding : 186
Code Display 5-7	Defining Messages	CCoding : 188
Code Display 5-8	Aliasing Messages	CCoding : 191
Code Display 5-9	Declaring Instance Data Fields	CCoding : 194
Code Display 5-10	Examples of Instance Data Declarations ..	CCoding : 197
Code Display 5-11	Defining Variable Data	CCoding : 198
Code Display 5-12	Variable Data Handlers	CCoding : 201
Code Display 5-13	Use of the @noreloc Keyword	CCoding : 203
Code Display 5-14	A Class Definition	CCoding : 206
Code Display 5-15	Declaring a Method As a Routine	CCoding : 208
Code Display 5-16	Declaring Data Resources	CCoding : 212
Code Display 5-17	Declaring Objects with @object	CCoding : 218
Code Display 5-18	Instantiating an Object	CCoding : 230
Code Display 6-1	Defining a Library—the sound.gp File	CAPpl : 258
Code Display 6-2	Saving Generic Object Options	CAPpl : 270
Code Display 6-3	Example GEOS.INI File Entries	CAPpl : 272
Code Display 6-4	EC Macros	CAPpl : 282
Code Display 7-1	ClipboardItemHeader	CCLipb : 303

Code Display 7-2	ClipboardItemFormatInfo	CClipb : 306
Code Display 7-3	ClipboardQueryArgs and ClipboardRequestArgs	CClipb:307
Code Display 7-4	A Sample Edit Menu	CClipb : 311
Code Display 7-5	Handling Clipboard Changes	CClipb : 313
Code Display 7-6	GenEditControl Features and Tools	CClipb : 315
Code Display 7-7	MSG_META_CLIPBOARD_CUT	CClipb : 317
Code Display 7-8	MSG_META_CLIPBOARD_COPY	CClipb : 318
Code Display 7-9	MSG_META_CLIPBOARD_PASTE	CClipb : 320
Code Display 8-1	Storing Strings in Localizable Resources	CLocal : 339
Code Display 8-2	Storing Strings in Localizable Resources	CLocal : 340
Code Display 9-1	Adding a Process Object to a GCN List	CGCN : 360
Code Display 9-2	Removing a Process from a GCN list	CGCN : 364
Code Display 9-3	Creating New Notification Types and Lists ..	CGCN : 366
Code Display 9-4	Adding Yourself to a Custom GCN List	CGCN : 367
Code Display 9-5	Using MSG_META_NOTIFY	CGCN : 368
Code Display 9-6	MSG_META_NOTIFY_WITH_DATA_BLOCK	CGCN : 369
Code Display 9-7	Intercepting an Application Notification Change	CGCN:370
Code Display 9-8	Removing from an Application GCN List	CGCN : 371
Code Display 10-1	TicTacApp and TicTacPrimary	CUIOver : 393
Code Display 10-2	The TicTac Game Menu	CUIOver : 395
Code Display 10-3	The TicTacView Object	CUIOver : 396
Code Display 10-4	TicTacBoardClass and TicTacPieceClass	CUIOver : 398
Code Display 10-5	Methods of TicTacBoardClass	CUIOver : 401
Code Display 10-6	Methods for TicTacPieceClass	CUIOver : 406
Code Display 11-1	Sample MSG_META_KBD_CHAR Handler	CInput : 441
Code Display 11-2	Delivering Messages to the Focus	CInput : 455
Code Display 11-3	Delivering Messages to the Target	CInput : 459
Code Display 12-1	Orienting a Composite	CGeom : 475
Code Display 12-2	A Complex Dialog Box	CGeom : 476
Code Display 12-3	Using Full Justification of Children	CGeom : 483
Code Display 12-4	Creating a Reply Bar	CGeom : 494
Code Display 13-1	Allocating Single Music Notes	CSound : 510
Code Display 13-2	Playing Single Music Notes	CSound : 512
Code Display 13-3	Stopping and Freeing a Note	CSound : 512
Code Display 13-4	Simple Sound Buffer Example	CSound : 514

Code Display 13-5	Preparing and Playing Sound Buffers	CSound : 517
Code Display 13-6	Stopping and Freeing a Sound	CSound : 517
Code Display 15-1	Allocating and Using a Block	CMemory : 557
Code Display 16-1	LMem Usage in GOC	CLMem : 581
Code Display 16-2	Example of Chunk Array Usage	CLMem : 592
Code Display 16-3	Structure for Element Array Elements	CLMem : 599
Code Display 16-4	Allocating a Name Array	CLMem : 605
Code Display 17-1	The DiskInfoStruct Structure	CFile : 621
Code Display 17-2	Standard FileEnum() Return Structures	CFile : 660
Code Display 18-1	VMInfoStruct	CVM : 692
Code Display 19-1	CellFunctionParameters	CDB : 735
Code Display 19-2	Rectangle	CDB : 738
Code Display 19-3	The RangeInsertParams and Point structures	CDB : 739
Code Display 23-1	Multi-Page Printing Loop	CGraph : 835
Code Display 23-2	GSCOMMENT Example	CGraph : 836
Code Display 23-3	GString in Visual Monikers	CGraph : 837
Code Display 23-4	GString Declared Without Macros	CGraph : 838
Code Display 23-5	Statically Declared GString	CGraph : 839
Code Display 23-6	Creating a GString Dynamically	CGraph : 840
Code Display 23-7	GrDrawGString() in Action	CGraph : 843
Code Display 23-8	GrSetGStringPos() In Action	CGraph : 846
Code Display 23-9	GrGetGStringElement() In Action	CGraph : 847
Code Display 24-1	Color Data Structures	CShapes : 886
Code Display 24-2	Hatch Pattern Data	CShapes : 894
Code Display C-1	A Library Entry Point	CLibr : 938
Code Display D-1	Extracting an Exponent	CMATH : 946
Code Display D-2	Adding Two FP Numbers	CMATH : 947
Code Display D-3	Creating a Random Number	CMATH : 951
Code Display D-4	FloatFloatToAsciiData Structure	CMATH : 953
Code Display D-5	DateTime Parameters	CMATH : 957
Code Display D-6	System-defined Float Formats	CMATH : 960
Code Display D-7	Float Format IDs	CMATH : 962
Code Display D-8	User-defined Formats	CMATH : 963

List of Figures

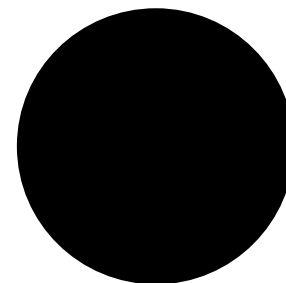


Figure 3-1	The Layers of GEOS	CArch : 65
Figure 3-2	A Sample Counter Object.....	CArch : 71
Figure 3-3	Objects Using Messages	CArch : 72
Figure 3-4	A Sample Subclass.....	CArch : 74
Figure 3-5	Sample <i>OSF/Motif</i> Menus	CArch : 81
Figure 3-6	Sample <i>OSF/Motif</i> Dialog Boxes.....	CArch : 83
Figure 4-1	The Hello World Sample Application	CGetSta : 109
Figure 4-2	Structure of Hello World	CGetSta : 110
Figure 4-3	Generic Tree of Hello World.....	CGetSta : 121
Figure 4-4	Hello Primary and View Windows.....	CGetSta : 123
Figure 4-5	Hello World's Menu and Dialog Box	CGetSta : 126
Figure 5-1	An Object Pointer.....	CCoding : 160
Figure 5-2	Object and Class Interaction	CCoding : 161
Figure 5-3	Structures of an Object	CCoding : 162
Figure 5-4	An Object with Vardata.....	CCoding : 163
Figure 5-5	Master Classes and Master Groups	CCoding : 164
Figure 5-6	A Sample Instance Chunk.....	CCoding : 165
Figure 5-7	A Class Tree.....	CCoding : 166
Figure 5-8	The ClassStruct Structure.....	CCoding : 167
Figure 5-9	Class_masterOffset	CCoding : 168
Figure 5-10	A Variant Class Object.....	CCoding : 173
Figure 5-11	A Resolved Variant Object.....	CCoding : 174
Figure 5-12	TicTac's New Game Trigger	CCoding : 175
Figure 5-13	GenTrigger's Instance Chunk	CCoding : 176
Figure 5-14	TicTacNewTrigger's Class Tree.....	CCoding : 177
Figure 5-15	GenTriggerClass Subclassed	CCoding : 178
Figure 5-16	Conflicting Message Numbers	CCoding : 189
Figure 5-17	Variable Data Storage.....	CCoding : 196
Figure 5-18	Structure of an Object Tree.....	CCoding : 234

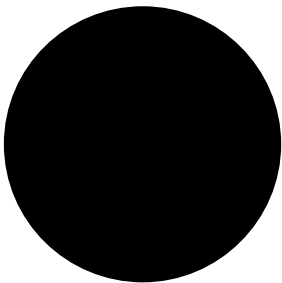
Figure 6-1	IACP Clients and Servers.....	CAppI : 285
Figure 7-1	The Edit Menu	CClipb : 300
Figure 7-2	The Transfer VM File.....	CClipb : 302
Figure 7-3	Transfer VM File Structure.....	CClipb : 305
Figure 8-1	International Formats	CLocal : 336
Figure 8-2	Localizing Strings.....	CLocal : 338
Figure 8-3	Parts of a Number Format	CLocal : 342
Figure 8-4	Parts of the Currency Format.....	CLocal : 344
Figure 9-1	A GCN List of Lists	CGCN : 356
Figure 10-1	Generic Class Hierarchy	CUIOver : 377
Figure 10-2	Visible Class Hierarchy	CUIOver : 378
Figure 10-3	The TicTac Object Trees.....	CUIOver : 393
Figure 10-4	The TicTac Game Board.....	CUIOver : 397
Figure 10-5	Selection of a Game Piece	CUIOver : 406
Figure 11-1	Input Flow	CInput : 420
Figure 11-2	A Pointer Image and Its Mask.....	CInput : 436
Figure 11-3	A Sample Target Hierarchy.....	CInput : 450
Figure 11-4	Hierarchy Terminology.....	CInput : 452
Figure 12-1	Horizontal and Vertical Composites.....	CGeom : 475
Figure 12-2	A Complex Dialog Box.....	CGeom : 476
Figure 12-3	Examples of Edge Justification	CGeom : 478
Figure 12-4	Examples of Centered Children	CGeom : 479
Figure 12-5	Full-Justification of Children.....	CGeom : 481
Figure 12-6	Expanding Children to Fit Parents.....	CGeom : 482
Figure 12-7	Sample Justifications	CGeom : 483
Figure 12-8	Drawing Boxes Around Composites.....	CGeom : 487
Figure 12-9	Placing Monikers	CGeom : 488
Figure 12-10	Centering on Monikers	CGeom : 490
Figure 12-11	Allowing Children to Wrap	CGeom : 493
Figure 12-12	Implementations of a Reply Bar	CGeom : 495

Figure 13-1	Sheet Music Excerpt for Example	CSound : 515
Figure 13-2	Modeling with the ADSR envelope.	CSound : 525
Figure 13-3	Continuing Instrument Envelopes	CSound : 526
Figure 13-4	Diminishing Instrument Envelopes.....	CSound : 527
Figure 15-1	Heap Compaction.....	CMemory : 549
Figure 16-1	A Local Memory Heap.....	CLMem : 571
Figure 16-2	Referencing a Chunk.....	CLMem : 573
Figure 16-3	Uniform-Size Chunk Array	CLMem : 585
Figure 16-4	Variable-Size Chunk Array.....	CLMem : 586
Figure 17-1	Accessing Files Through the File System..	CFile : 613
Figure 17-2	A Path Through a Directory Tree	CFile : 615
Figure 18-1	Structure of a VM File	CVM : 675
Figure 18-2	A Fragmenting VM File	CVM : 678
Figure 18-3	VM File Compaction.....	CVM : 679
Figure 18-4	Saving a backup-enabled VM file	CVM : 694
Figure 18-5	A VM Chain	CVM : 701
Figure 18-6	Structure of a VM tree block	CVM : 703
Figure 18-7	Huge Array Structure	CVM : 707
Figure 19-1	Dereferencing a DB Item.....	CDB : 722
Figure 19-2	Inserting a New Row	CDB : 733
Figure 21-1	Stream Writer and Reader Interaction .	CStream : 774
Figure 23-1	Clipping	CGraph : 805
Figure 23-2	Graphics System Architecture	CGraph : 808
Figure 23-3	Coordinate Spaces	CGraph : 811
Figure 23-4	Effects of Simple Transformations.....	CGraph : 813
Figure 23-5	Rotating an Object About Its Center.....	CGraph : 814
Figure 23-6	Ordering Transformation Combinations	CGraph : 815
Figure 23-7	Document and Device Coordinates	CGraph : 820
Figure 23-8	Different Device Coordinates	CGraph : 821
Figure 23-9	Drawing Thin Lines.....	CGraph : 822

Figure 23-10 Thirty-two Bit Graphics Spaces	CGraph : 824
Figure 23-11 GStrings and Associated GStates	CGraph : 833
Figure 23-12 Graphics Paths	CGraph : 849
Figure 23-13 Combining Paths	CGraph : 850
Figure 23-14 Geode-Defined Clipping Paths	CGraph : 850
Figure 24-1 Point	CShapes : 860
Figure 24-2 Line	CShapes : 861
Figure 24-3 Rectangle	CShapes : 862
Figure 24-4 Ellipse	CShapes : 863
Figure 24-5 Elliptical Arc	CShapes : 863
Figure 24-6 Three-point Arcs	CShapes : 864
Figure 24-7 Advantages of the Three-point Arc	CShapes : 865
Figure 24-8 Rounded Rectangle	CShapes : 865
Figure 24-9 Polylines, Polygons, and Fill Rules	CShapes : 866
Figure 24-10 Splines and Bézier Curves	CShapes : 868
Figure 24-11 Levels of Smoothness	CShapes : 870
Figure 24-12 Bitmap	CShapes : 871
Figure 24-13 GrDrawImage()	CShapes : 872
Figure 24-14 Path Direction for Winding Fills	CShapes : 872
Figure 24-15 Path Intersection and Union	CShapes : 873
Figure 24-16 Sample Region	CShapes : 875
Figure 24-17 Complicated Region Example	CShapes : 877
Figure 24-18 TextMode and Drawing Position	CShapes : 879
Figure 24-19 Common Font Metrics	CShapes : 882
Figure 24-20 Character Metrics	CShapes : 884
Figure 24-21 Dithering	CShapes : 889
Figure 24-22 Custom Hatch Patterns	CShapes : 893
Figure 24-23 Mix Modes	CShapes : 896
Figure 24-24 Masks as Repeating Patterns	CShapes : 898
Figure 24-25 Drawing with Masks	CShapes : 899

Figure 24-26	LineStyle	CShapes : 900
Figure A-1	Structure of a Word	CHardw : 906
Figure A-2	Accessing a Byte in a Segment	CHardw : 907
Figure A-3	The Four General Registers	CHardw : 909
Figure D-1	Translating FP to decimal	CMath : 965
Figure D-2	Diagram of an 80-bit Floating Point Number	CMath : 966
Figure D-3	The Floating Point Stack	CMath : 968

Introduction



1

- 1.1 Overview of The Documentation..... 37**
 - 1.1.1 What You Will Learn..... 37
 - 1.1.2 What You Are Expected To Know..... 37
 - 1.1.3 Roadmap to the Development Kit..... 38
 - 1.1.4 Typographical Cues..... 39
- 1.2 Chapters in the Books..... 40**
 - 1.2.1 The Concepts Book 40
 - 1.2.2 The Object Reference Book..... 43
 - 1.2.3 The Tools Reference Manual..... 47
 - 1.2.4 The Esp Book..... 48
- 1.3 Suggestions for Study 48**





Congratulations on taking the first step towards programming for GEOS. This system will most likely be unlike anything you have programmed for before; among the main goals of the system design were to simplify development of applications and to incorporate many common application and User Interface features within the system software.

1.1

1.1 Overview of The Documentation

These manuals represent the initial non-Beta release of technical documentation for the GEOS operating system. These manuals should provide you with all the knowledge, both conceptual and reference, to write programs for GEOS.

1.1.1 What You Will Learn

This documentation provides everything you need to write a complete GEOS application. It includes in-depth conceptual and reference material about every exported kernel routine and system object.

These books will teach you about GEOS—how the operating system works, from file management to messaging to object creation and destruction. If you read everything in these books, you should be able to create the source code for not only simple applications but even applications of medium complexity.

1.1.2 What You Are Expected To Know

This documentation relies heavily on the reader's knowledge of the C programming language and of Object-Oriented Programming (OOP) concepts. If you are unfamiliar with either of these topics, you should become familiar with them before continuing.

Concepts book



You are also expected to have a working familiarity with Geoworks® products. Working familiarity with the software is important to understand the features of the system from the user's perspective. In addition, many User Interface concepts are illustrated with examples from the retail products.

1.1.3 Roadmap to the Development Kit

1.1

The developer kit documentation is separated into several books. Each of these books has a primary purpose; together, they should give you all the information you need to know about GEOS and how to program for it. The books are

Tutorial The Tutorial describes how to set up your system, how to begin running the tools, and how to get started programming GEOS applications. It takes you step-by-step through modifying, compiling, and debugging a sample program.

Concepts Book This is the Concepts Book. This book explains the structure and concepts of creating a GEOS application. It will help you plan and create the structure of your applications and libraries, and it details all the functions of the GEOS kernel. It describes which system objects you will want to use for various situations and to get various results.

Objects Book The Objects Book contains C reference information and detailed, in-depth discussion of each of the system-provided objects. This book is a hybrid of the traditional reference and conceptual manuals; each chapter contains both a detailed description of and a detailed reference for each object. In most cases, each object is given its own chapter. In some cases, however, several related objects share a single chapter.

Esp Manual The Esp manual describes Esp, GEOS's OOP assembly language, the language in which most of the GEOS kernel is written. Using Esp, you will be able to write optimized routines and applications to handle your most processing-intensive tasks.

C Reference Book The Routines Book details the data structures, routines, and other typical reference material you'll need. It focuses on the routines and



functions provided by the GEOS kernel including the Graphics System and Memory Management.

Esp Reference Book

The Esp reference book provides Esp (assembly-language) information for the structures and routines used by the GEOS kernel and the system libraries.

Tools Manual

The Tools manual describes all the tools included with the SDK. It includes descriptions of the system setup, the Swat debugger, the icon editor, the localization tools, the GEOS initialization file, and all the other tools in the system. In addition, it also details the Tool Command Language, which allows you to extend Swat's functionality for your own purposes.

1.1

Objects Quick-Reference Manual

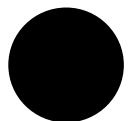
The Quick-Reference manual is a handy booklet that has not text information, but rather all the pass and return and definition information for all the object classes in GEOS. It should be used after you've become familiar with the concepts of the system.

1.1.4 Typographical Cues

Throughout these manuals, you will see several words in bold or italics, and you will read several code examples. For the most part, there are four types of typographical cues that you will encounter:

- ◆ **Book Symbols**
Each book in this developer kit documentation is designated a special shape to help you identify it quickly on the shelf. Wherever possible, this shape is used along with the book title (in cross-references, for example).
- ◆ **Boldface Text**
Boldface text is used to denote GEOS class names, routine and function names, and data structures. It is also used for file names, as-typed commands, and headers of many lists.
- ◆ **Italic Text**
Italic text is used to denote terms that can be found in the glossary, parameters passed to routines and messages, and flags. Variables are often also designated with italics. Note, however, that flags that are all capital letters are not in italics.

Concepts book



- ◆ **Monospace Font**
Monospace font is used for all code samples and illustrations of commands. It is also used as a subheading for sections that describe particular routines, messages, and data structures.

1.2 Chapters in the Books

1.2

This section of this chapter lists all the chapters in each of the main books of this documentation.

1.2.1 The Concepts Book

The Concepts Book describes not only the concepts of the GEOS system but also the steps and components of applications. Typically, a reader will read straight through the first six chapters and then choose whichever order of chapters suits her or him best after that. The book is designed for sequential reading, however.

- 1 Concepts Introduction**
This chapter describes the structure and components of the developer kit documentation and of each book in it. You are reading the Concepts Introduction now.
- 2 Building an Application**
This gives a terse, feature-by-feature listing of chapters you will want to read for various topics. It is much like this section except it is by feature rather than by sequence.
- 3 System Architecture**
This describes the architecture of GEOS and the part that each of its components plays. It describes how applications and libraries fit into the scheme of GEOS as a whole, and it enumerates the mechanisms used throughout the system.
- 4 First Steps: Hello World**
This is an in-depth look at the basics of a GEOS application through a detailed example. It uses a program appropriately titled Hello World; this program draws a primary window, creates a scrolling window, draws



text in the window, and uses a menu and dialog box to allow the user to change the color of the text.

5 GEOS Programming

This describes all the keywords available in the GEOS Goc programming language. It also discusses data types and the GEOS object system. It describes how objects are created and destroyed and how classes are used.

6 Applications and Geodes

This details how an application and other geodes (GEOS executables) are loaded and shut down. This chapter also discusses several things that may be of importance to application programmers such as saving user options and other system utilities.

1.2

7 The Clipboard

This is about the Clipboard and the quick-transfer data transfer mechanism. The Clipboard implements the cut, copy, and paste features usually found in the Edit menu of an application. The quick-transfer mechanism implements the “drag-and-drop” functionality inherent in text objects and available to applications for all data formats.

8 Localization

This discusses how developers can localize their applications for international markets. It discusses not only the Localization Driver but also how to plan ahead when writing your applications.

9 General Change Notification

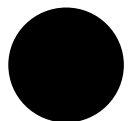
This discusses the General Change Notification (GCN) mechanism of the kernel. This mechanism allows objects to register interest in a particular event so that whenever that event occurs, the object will be notified of the change.

10 The GEOS User Interface

This describes in detail how the GEOS User Interface works. It gives a sample application using visible objects (objects that you use to draw graphics and to interact with the user). This chapter is the “visible world” counterpart to the Hello World application of chapter three.

11 The Input Manager

This describes how GEOS accepts, manages, and passes on input events. An input event can be motion of the mouse, use of a stylus, a press on the keyboard, a click on a mouse button, or some other event.



12 The Geometry Manager

This describes various ways you can manage the size, position, and geometry behavior of generic User Interface objects (e.g. windows, dialog boxes, and scrolling view windows).

13 Sound Library

This details how to use the GEOS sound library with different sound generators.

14 Handles

This discusses in detail the concept, implementation, and use of GEOS handles. Handles are an integral part of the GEOS system for memory management, file management, graphics drawing, and message passing.

15 Memory Management

This details the GEOS memory model and how to allocate, use, and free up memory for your application.

16 Local Memory

This describes how to use local memory. Local memory is a mechanism used to store the instance data of objects as well as small chunks of data such as strings, small graphics, and database items.

17 File Systems

This describes how GEOS interacts with the DOS or other file-access system in use. This chapter details how GEOS applications can open, edit, close, and manipulate all kinds of files. It also discusses how to access disks and drives directly.

18 Virtual Memory

This discusses the GEOS Virtual Memory model. Virtual Memory is used not only to manage memory swapping but also for storing GEOS data and object files. The Virtual Memory mechanism is an integral part of GEOS and will be used by many applications and libraries.

19 Database

This details the Database Library provided with GEOS. The Database Library provides the low-level routines to create, edit, free up, and organize individual database items.

20 Parse Library

This details the Parse Library. This library implements a special mathematical description language.

21 Streams

This discusses the concept and use of streams, a data-transfer mechanism to pass data either through hardware ports or across threads in GEOS. Included in this chapter are the Parallel Driver and Serial Driver, integral in accessing the PC's communication ports.

22 Graphics Environment

This describes in detail the GEOS coordinate and graphics system. It covers the coordinate space, how graphics are drawn, the drawing algorithms, and many other topics. Described herein are GEOS "graphic states" and "graphic strings," both integral parts of GEOS.

1.2

23 PCCom Library

This explains the use of the PC communications library. It explains how a geode may use the library to monitor a serial port, or to transfer files and information to or from a remote machine.

24 Graphics

This is an extension of the discussion of chapter twenty-one and explains how to draw various shapes. It enumerates your options for drawing graphics in GEOS, from simple graphics routines to complex graphical objects provided in the Graphic Object Library.

The Concepts Book also has four appendixes: The first gives a background of the PC architecture. The second gives an in-depth discussion of threads and thread management. The third describes how to create GEOS libraries. The fourth describes the GEOS floating-point math library.

1.2.2 The Object Reference Book

The Object Reference Book is a hybrid of traditional conceptual and traditional reference books. Each chapter gives both in-depth conceptual and usage information as well as specific reference material for the subject objects. It is assumed that you will read through a major portion of the Concepts Book before embarking on the Object Reference chapters; the Concepts Book will direct you to the individual objects you will need to read about in the Object Reference.

1 System Classes

This details the three system classes that handle many of the messages and provide much of the built-in functionality for most objects. These

Concepts book



classes are **MetaClass**, the root of the GEOS class tree; **ProcessClass**, the main class of an application's process thread; and **GenProcessClass**, the superclass of **ProcessClass**.

2 GenClass

This details **GenClass** and much of the common functionality of all generic objects. It describes generic object trees, messaging between generic objects, and using monikers with generic objects. It also discusses input issues with respect to the generic hierarchies.

3 GenApplication

This covers **GenApplicationClass**, the class of all application objects. Every application will have an application object as the root of its generic object tree. This object has no visual representation but handles loading and shutting down of the application geode.

4 GenDisplay/GenPrimary

This describes the window objects in GEOS. The GenPrimary object provides an application's primary window; the GenDisplay and Display Control objects provide window objects for individual documents. With these objects, an application can provide multiple scrolling displays within its primary window.

5 GenTrigger

This covers **GenTriggerClass**, the class that implements basic triggers and buttons in the User Interface.

6 GenGlyph

This covers the Glyph class, which allows an application to display a small portion of text or graphics without the overhead of other, more complex objects.

7 GenInteraction

This discusses **GenInteractionClass**, a versatile class that implements both menus and dialog boxes.

8 GenValue

This discusses the GenValue object. This object allows the user to set a value within a specified range.

9 GenView

This covers the GenView object. The GenView provides a scrolling window for applications to draw graphics or otherwise display objects or data in.

10 The Text Objects

This describes how text is used throughout GEOS. Any application that expects to display text, accept text input, or provide text formatting features will use one of the text objects provided by GEOS. This chapter describes how and when to use each of these objects.

11 The List Objects

This details the different types of lists you can create with the various list-related classes including GenBoolean, GenBooleanGroup, GenItem, GenItemGroup, and GenDynamicList.

1.2

12 GenControl/GenToolControl

This details the controller and toolbox classes that allow an application to use and create controllers. Controller objects automatically build menus and dialog boxes to manage a certain feature set of an application. The Tool Control object allows the user to configure his or her system to place certain tools either in a floating tool box or in various menus.

13 GenDocument

This describes the GenDocument and the document control objects. These objects help applications manage data files (documents) and provide the common functionality of New, Open, Close, Save, SaveAs, and Revert.

14 GenFileSelector

This describes how and when to use a GenFileSelector object. The GenFileSelector provides the user interface that allows users to traverse their file systems and view their directories. It also lets them select a file for opening or other operations.

15 Help Object Library

This details the Help Object Library. The help object allows your application to provide context-sensitive help text in a system-standard way. The help object will create the user interface and will automatically provide text linking through your help documents.

16 Import/Export Library

This describes the import and export mechanism used by GEOS. The Impex Library connects GEOS to individual translation libraries. This chapter describes not only how to create new translation libraries but also how to link them into GEOS.

17 Spool Object Library

This describes the Spool Object Library. This library exports classes



which allow applications to print and otherwise interact with the GEOS spooler.

18 Graphic Object Library

This details the Graphic Object Library. This library offers several objects and classes that provide full graphic object editing and display features. Graphic objects include line, rectangle, ellipse, spline, polyline, and polygon, among others.

1.2

19 Ruler Library

This details the Ruler Library; this library contains vertical and horizontal ruler objects which can be connected to text, graphic, and spreadsheet views.

20 Spreadsheet Object Library

This covers the Spreadsheet Object Library. This library exports several classes that can be used as the basis of a spreadsheet application when used with the Cell, and Parse libraries.

21 Pen Object Library

This describes the built-in pen and Ink support of GEOS. Pen input can be managed by a set of special objects that an application may interact with.

22 VisClass

This details **VisClass**, the basic visible object class of GEOS. **VisClass** can be used to create objects that can draw themselves and accept input. **VisClass** is also the ancestor class of many special objects in GEOS (e.g. the Ruler Object, the Spreadsheet Object, etc.).

23 Config Library

This discusses how to write Preferences modules. This is useful for programmers who are defining new fields in the GEOS.INI file.

24 VisComp

This details **VisCompClass**. This is a subclass of **VisClass** and provides additional features such as object tree manipulation and automatic geometry management of a visible object's children.

25 VisContent

This details **VisContentClass**, a subclass of **VisCompClass**. The VisContent is used to display a visible object tree within a GenView graphic window and to manage the geometry of several visible objects within the window.

26 Generic System Classes

This covers several generic system classes that will not ever be used directly by application programmers. They are provided here, however, to give reference information about the messages that can be sent to them. Included in this chapter are GenSystem, GenScreen, and GenField.

1.2.3 The Tools Reference Manual

1.2

The Tools Reference manual describes many things about the tools and the development station setup that you will need to know throughout your development period. It contains the following chapters:

1 Introduction

This is an outline of the Tools Reference Manual.

2 System Configuration

This describes the host and target machine setups after the SDK has been installed and is working.

3 Swat Introduction

This introduces you to Swat, the powerful system debugger supplied in the SDK. This chapter also gives you the most popular Swat commands and explains how you can put them to use for you.

4 Swat Reference

This gives detailed command reference entries for each Swat command. You should use this section when you need to know the specifics of one or more Swat commands.

5 Debug

This explains how to use the Debug tool, which allows you to simulate a number of GEOS platforms and hardware configurations.

6 Tool Command Language

This describes Tcl, the Tool Command Language that allows you to extend Swat's functionality. Nearly all Swat commands are programmed in Tcl, and most are accessible as functions within other Tcl commands. By using Tcl, you can write your own debugger commands or extend the functionality of Swat's provided commands.

7 Icon Editor

This describes how to use the GEOS icon editor tool.

- 8 Resource Editor**
This describes how to use the GEOS resource editor tool for localization.
- 9 The INI File**
This describes the various categories and keys of the GEOS.INI file used by GEOS.
- 10 Using Tools**
This details all the other tools in the system, including the communications utilities, the make utilities, GOC, Glue, and others.

1.2.4 The Esp Book

The Esp book describes how to use Esp, the GEOS object-oriented assembly language. You can use this language to recode computation-intensive routines, or to write optimized applications, libraries, or drivers.

- 1 Introduction**
This chapter provides a brief introduction to Esp, and contains a roadmap to the Esp book.
- 2 Esp Basics**
This chapter describes the differences between Esp and other 80x86 assembly languages (such as MASM). It also describes how to define classes.
- 3 Routine Writing**
This chapter describes how to write routines and methods (message handlers). It also describes Esp's special facilities for managing stack frames and sending messages.
- 4 The User-Interface Compiler**
This chapter describes UIC, a tool for generating object-blocks for Esp programs.

1.3 Suggestions for Study

If you are unfamiliar with programming, you will most likely not be able to follow this documentation well. You should have working familiarity with the C programming language and with Object-Oriented Programming concepts.



This documentation will not attempt to teach those subjects. If you are not familiar with these subjects, you may have trouble following some of the discussions throughout.

For those ready to begin, it is suggested that you read the first six chapters of this book before branching out into different in-depth concepts. From there, you should read about the Generic UI objects, with focus on GenInteraction, GenTrigger, GenPrimary, and GenView. These will build on the introduction you gained from the first chapters of the Concepts Book.

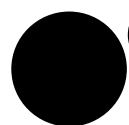
1.3



Introduction

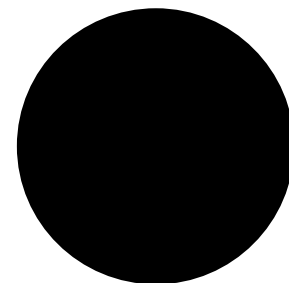
50

1.3



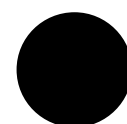
Concepts book

Building Your Application



2

2.1	What Everyone Should Read.....	53
2.2	Topics Listing	54
2.2.1	Defining Your User Interface	54
2.2.2	Providing Other User Interface	56
2.2.3	Documents and Data Structures	56
2.2.4	Accessing Hardware.....	57
2.2.5	Programming Topics	58
2.2.6	Other Topics.....	58





The GEOS system is complex and provides so many services that the documentation at first may seem rather daunting. This chapter is meant to help you through the application process by directing you to the proper chapters for various functions.

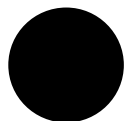
This chapter is by no means a comprehensive list of all functions supported by GEOS. It is intended to be a guide to the chapters that cover some of the more widely-used functions and features.

2.1

2.1 What Everyone Should Read

Anyone who wants to program a GEOS application should read at least the first several chapters of the Concepts Book. Among the chapters you should read before going too far are

- ◆ “System Architecture,” Chapter 3
This chapter describes the GEOS system in an overview format. It can give you a good idea of how the system works and the features it might provide for your application. You should read this chapter now if you haven’t already.
- ◆ “First Steps: Hello World,” Chapter 4
This chapter describes the Geode Parameters file as well as an example of a simple application. The application puts up a window, uses a menu and a dialog box, and draws graphic-based text within a scrolling window. It also shows how GEOS messaging works, how a message is sent and handled, and how objects interact with each other.
- ◆ “GEOS Programming,” Chapter 5
This chapter details the basics and even many of the finer points of programming with the Goc preprocessor. Goc has many keywords used for object and class definition, message sending and handling, and resource definition. Before programming for GEOS, you must know how to use several of these keywords (though you can learn several of the most important ones from the sample applications).
- ◆ “The GEOS User Interface,” Chapter 10
This chapter describes the basic generic and visual UI object classes



available. It also details a simple application which uses visual objects that draw themselves in a window, intercept mouse clicks, and allow the user to drag them around the screen. The sample application also illustrates how to create your own objects from the base GEOS visual classes.

2.2 Topics Listing

Listed below are many of the topics related to creating an application. Accompanying the topic is a list of chapters you will probably want to read to fully understand that topic.

2.2.1 Defining Your User Interface

“The GEOS User Interface,” Chapter 10, gives an overview of the User Interface and which objects you will want to use. Many user interface functions, objects, and features are listed below.

- ◆ Overview of all the UI Objects
All the generic and visual UI classes are discussed in “The GEOS User Interface,” Chapter 10. This chapter is a must-read for nearly everyone.
- ◆ The Primary Application Window
Almost all applications will have a primary window. “First Steps: Hello World,” Chapter 4 and “GenDisplay / GenPrimary,” Chapter 4 of the Object Reference Book discuss how to create and use the window in detail.
- ◆ Menus and Dialog Boxes
Menus and dialog boxes are created with the use of GenInteraction objects, described in “GenInteraction,” Chapter 7 of the Object Reference Book.
- ◆ Monikers and Icons
Monikers are labels of objects that are drawn on the screen to represent the object. Icons are special monikers. Both of these are discussed in “GenClass,” Chapter 2 of the Object Reference Book.

- ◆ **Scrolling and Non-Scrolling Graphics Windows**
The GenView will be used by most applications. Described in “GenView,” Chapter 9 of the Object Reference Book is the process of creating a graphics window and drawing into it.
- ◆ **Triggers and Buttons**
Triggers and buttons are described fully in “GenTrigger,” Chapter 5 of the Object Reference Book.
- ◆ **Lists** 2.2
Dynamic, static, and scrolling lists are all described in “The List Objects,” Chapter 11 of the Object Reference Book.
- ◆ **Value Setters**
The GenValue object allows the user to set a value within a given range. It is described in “GenValue,” Chapter 8 of the Object Reference Book.
- ◆ **Tool Boxes and Other Controllers**
An application can define or use controller objects that automatically get and apply a user’s choices. These controllers can be placed automatically within floating “tool boxes” by the user. Controllers and tool boxes are discussed in “Generic UI Controllers,” Chapter 12 of the Object Reference Book.
- ◆ **File Selector Dialog Boxes**
A standard dialog box for finding and selecting files is provided by GEOS and is described in “GenFile Selector,” Chapter 14 of the Object Reference Book.
- ◆ **Multiple Windows**
Your application can display multiple windows using the GenDisplay and GenDisplayControl objects described in “GenDisplay / GenPrimary,” Chapter 4 of the Object Reference Book.
- ◆ **Text Editing, Display, and Input**
All text functions are handled by the text objects, GenText and VisText. These are incredibly sophisticated and complete, and they are described in “The Text Objects,” Chapter 10 of the Object Reference Book.
- ◆ **Providing Help**
An application can include an object that automatically provides context-sensitive help to the user. This object is discussed in “Help Object Library,” Chapter 15 of the Object Reference Book.

- ◆ **Handling Input**
Many applications may want to track mouse or keyboard input. Input management is discussed in “Input,” Chapter 11.

2.2.2 Providing Other User Interface

2.2

Besides the generic UI functions and objects described above, GEOS provides a number of sophisticated graphics commands and powerful graphic objects. Graphics must be drawn to a GEOS document and the document displayed in a GenView.

- ◆ **The GEOS Graphics System**
The GEOS coordinate space is based on real-world units and is device-independent. It is described in full in “Graphics Environment,” Chapter 23.
- ◆ **Creating Your Own UI Objects**
Using the visible classes, you can create your own objects that draw themselves, handle user input, and do any number of other things. See “The GEOS User Interface,” Chapter 10 and “VisClass,” Chapter 23 of the Object Reference Book for descriptions of the visible object classes and how to use them.
- ◆ **Drawing Standard Graphics**
Graphics may be drawn by calling several different graphics commands. These commands are described in “Drawing Graphics,” Chapter 24.
- ◆ **Using Graphic Objects**
The Graphic Object Library provides many different graphic objects that know how to position, resize, and draw themselves as well as handle user input. This library is described in “Drawing Graphics,” Chapter 24 and “Graphic Object Library,” Chapter 18 of the Object Reference Book.

2.2.3 Documents and Data Structures

Applications that save files, print documents, or display multiple documents will be concerned with several of the topics listed below.

- ◆ **Creating and Using Documents**
The GenDocument and document control objects provide standard

document management including document file management and interaction with the display objects. See “GenDocument,” Chapter 13 of the Object Reference Book.

- ◆ **Importing and Exporting Data Formats**
Data of other applications can be imported and GEOS data files exported to other formats via the Impex Library and its associated format translators. These are discussed in “Impex Library,” Chapter 16 of the Object Reference Book.

2.2

- ◆ **Using Memory**
“Memory Management,” Chapter 15 describes the GEOS memory model and memory manager and how an application can use them.
- ◆ **Using Files**
Applications that save document files should use the document objects (see above). Otherwise, the application will use either GEOS virtual memory files (see “Virtual Memory,” Chapter 18) or normal files (see “File System,” Chapter 17).
- ◆ **Keeping Track of Database Items**
GEOS provides an item database manager that you can use to manipulate database items and files. This is described in “Database Library,” Chapter 19.
- ◆ **Using an Entire Spreadsheet**
GEOS also provides a spreadsheet object that implements the basic functionality of a spreadsheet and which can be included in an application. This object is described in “Spreadsheet Objects,” Chapter 20 of the Object Reference Book.
- ◆ **Printing**
If your application will print anything to a printer or to a file, you should read “The Spool Library,” Chapter 17 of the Object Reference Book.

2.2.4 Accessing Hardware

GEOS is designed to allow applications to be as device-independent as possible. Some applications will need to access hardware directly, however.

- ◆ **Serial and Parallel Ports**
An application that needs to access the serial and parallel ports can do so

using streams and the serial and parallel drivers. These are discussed in “Using Streams,” Chapter 21.

- ◆ **Disks, Drives, and CD ROM Drives**
Applications that work directly with disks and drives (utility programs especially) will use many of the features described in “File System,” Chapter 17.
- ◆ **Sound Hardware**
The standard PC speaker can be accessed through the sound library, discussed in “Sound Library,” Chapter 13. Drivers and libraries to allow applications to use more sophisticated sound hardware may be added later.
- ◆ **Video Hardware**
In general, GEOS takes care of all video driver operations; applications deal with the graphics system, which sends commands to the video drivers. For more information, see “Graphics Environment,” Chapter 23.

2.2.5 Programming Topics

A number of programming topics specific to Goc are described in “GEOS Programming,” Chapter 5. This chapter discusses the various Goc keywords and their uses. It also describes how to create and destroy objects, how to create classes, and how to send and receive messages. You should read this chapter if you plan on programming for GEOS using the C programming language.

For information on programming in assembly, see the Esp manual. It explains GEOS’s extensions to standard 80x86 assembly-language programming

2.2.6 Other Topics

Many other topics are discussed throughout the documentation. Some of those more commonly used are listed below.

- ◆ **Saving User Options**
Generic objects typically provide the features of option saving. If you need to enhance this, however, you can use the GEOS.INI file to save

application-specific user options or information. This is discussed in “Applications and Geodes,” Chapter 6.

◆ **Using Event Timers**

Many applications and libraries will use event timers. Timers are discussed in “Applications and Geodes,” Chapter 6.

◆ **Supporting Quick-Transfer and Using the Clipboard**

To support the Edit menu’s Cut, Copy, and Paste functions, see “The Clipboard,” Chapter 7.

2.2

◆ **Serving International Markets**

GEOS is designed to allow easy translation of applications and libraries to other languages. See “Localization,” Chapter 8 for full details.

◆ **Using Multiple Threads**

Because GEOS is multithreaded, any application can create new threads for itself at any time. The issues and procedures of multiple threads are described in “Threads and Semaphores,” Appendix B.

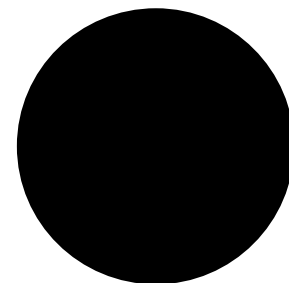


Building Your Application

60

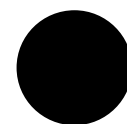
2.2

System Architecture



3

3.1	GEOS Overview	63
3.2	The System Architecture	64
3.2.1	Applications.....	66
3.2.2	Libraries	66
3.2.3	The Kernel.....	66
3.2.4	Device Drivers	67
3.2.5	The User Interface.....	67
3.3	Object-Oriented Programming	68
3.3.1	Objects, Messages, and Methods.....	69
3.3.2	Classes and Inheritance	73
3.4	Multitasking and Multithreading.....	75
3.5	The GEOS User Interface	76
3.5.1	The Generic User Interface	77
3.5.1.1	Attributes	77
3.5.1.2	Hints	78
3.5.1.3	The Generic UI Classes	78
3.5.2	The Scalable User Interface	79
3.5.3	Windows and Window Management.....	79
3.5.4	Input	80
3.5.5	Menus and Dialog Boxes	80
3.5.5.1	Menus	81
3.5.5.2	Dialog Boxes.....	82
3.5.6	Scrolling Views.....	84
3.5.7	Visible Object Classes.....	86
3.5.8	Geometry Manager	86
3.5.9	Lists.....	86
3.5.10	Other Gadgets	87
3.5.11	Managing Documents and Files	88



3.5.12	Multiple Document Interface.....	88
3.5.13	Clipboard and Quick-Transfer	88
3.5.14	General Change Notification	89
3.5.15	Help Object.....	89
3.6	System Services	90
3.6.1	Memory	90
3.6.1.1	Handles	90
3.6.1.2	The Global Heap	91
3.6.1.3	Allocating Memory.....	92
3.6.1.4	Accessing Memory	93
3.6.2	Virtual Memory	93
3.6.3	Local Memory and Object Blocks	93
3.6.4	Item Database Library.....	94
3.6.5	Graphics System.....	95
3.6.5.1	The Coordinate Space.....	95
3.6.5.2	Graphic States	95
3.6.5.3	Graphic Primitives and Graphic Objects.....	96
3.6.5.4	Paths.....	96
3.6.5.5	Regions	96
3.6.5.6	Graphics Strings	97
3.6.5.7	Bitmaps	97
3.6.5.8	Color	97
3.6.6	Text	98
3.6.7	Print Spooler and Printing.....	99
3.6.8	Timers	99
3.6.9	Streams.....	100
3.6.10	Math support	100
3.6.11	International Support	100
3.7	Libraries	101
3.8	Device Drivers.....	102



This chapter describes the structure of GEOS and the various system components and services used by applications. Nearly all programmers will want at least to browse this chapter before continuing with GEOS application development. The structures and architectures of the 8088 and 80x86 processors are reviewed in Appendix A.

3.1

3.1 **GEOS Overview**

GEOS is a state-of-the-art graphical operating system combining the latest software technology with numerous innovations. Its tightly coded assembly language and object-oriented system turns the humblest 8088-based PCs with as little as 512 K of RAM into graphical workstations. On higher-end machines, GEOS is in a class by itself—performance on 80286, 80386, and 80486 machines with extended or expanded memory is exceptional.

Besides being coded entirely in object-oriented assembly language (a concept and language developed by Geoworks), GEOS is built on one of the most sophisticated operating system architectures available for PCs. Most of the GEOS kernel and system is composed of dynamically-linkable libraries; in addition, developers can create their own libraries to support entire generations of applications.

GEOS also includes true pre-emptive multitasking, multiple threads of execution within single applications, built-in outline font technology, and drivers for various devices such as printers, mice, video cards, file systems, keyboards, parallel and serial ports, task switchers, power management systems, and font engines. Development for GEOS is easy and quick; much code traditionally required by applications has been provided within the system software, thereby reducing time spent by application programmers on coding and debugging.

In addition, GEOS employs a breakthrough generic user interface. Applications specify their user interface needs, and the system dynamically chooses the proper manifestation of the UI at run-time through the use of specific UI libraries. Complex drawing and geometry are handled

automatically by the UI library. The generic UI allows applications to work with various UI specifications without being recompiled. Users can choose the look and feel they want without burdening the application programmer with extra coding.

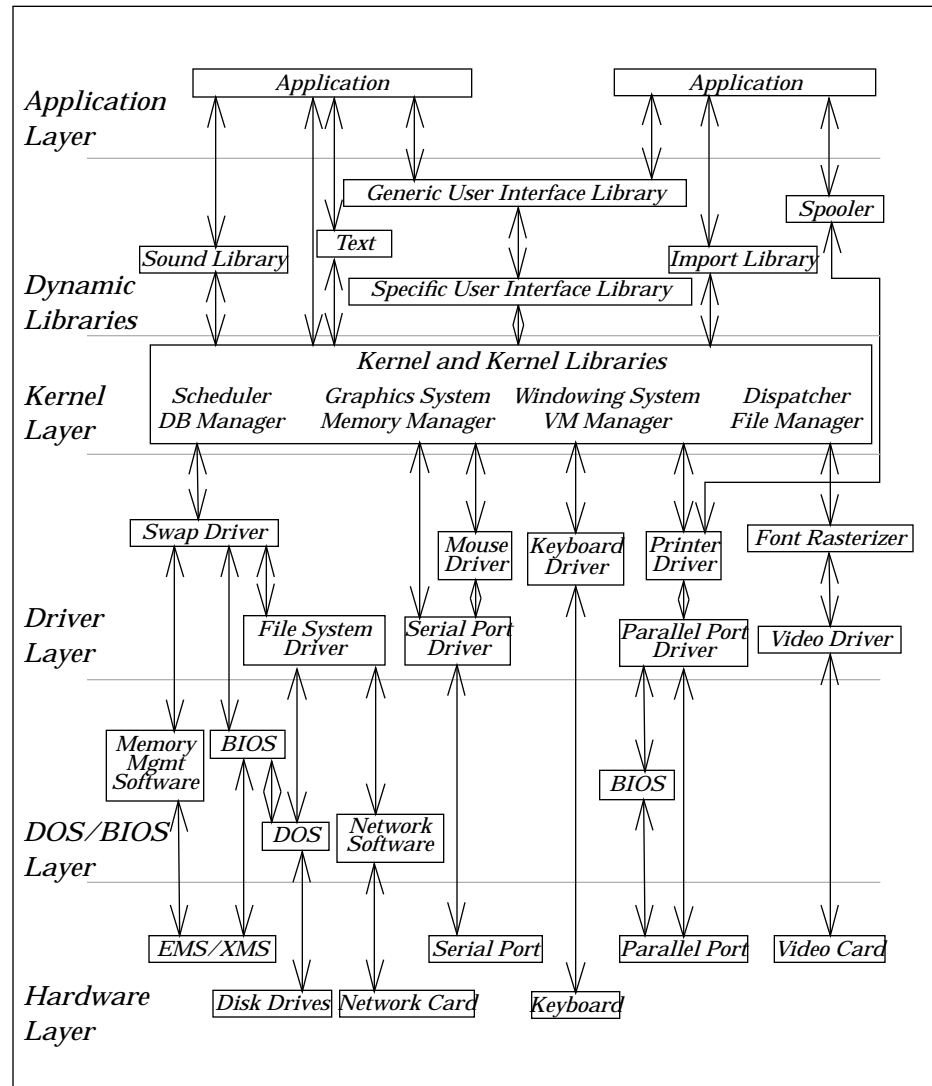
3.2

Among the other advanced features included in GEOS are a single imaging model providing true WYSIWYG (a single image is used both for drawing to the screen and for printing); automatic scaling and scrolling of displayed data; dynamic memory management and virtual memory; an item database manager; a graphic object library including support for resize, move and scale operations; complex and complete text formatting and editing; specialized generic controller objects providing standard user interface and functionality; and full network support. Several other mechanisms and libraries not mentioned here are also built into the system.

The following sections of this chapter describe many of these concepts and implementations, providing a quick-view of the GEOS system as a whole. More information may be found on individual topics elsewhere in this developer's kit. This chapter provides the background information necessary, however, to take full advantage of all that GEOS has to offer the application developer.

3.2 The System Architecture

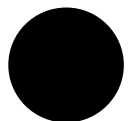
The GEOS system as a whole consists of several layers. The topmost layer contains applications. Users interact directly with these applications, which use dynamic libraries on the next layer. One of the major libraries is the User Interface Library, commonly known as the UI. Both the libraries and applications interact with the kernel, the heart of the operating system. The kernel communicates with device drivers, which function as the final GEOS interface to physical devices such as video screens, printers, and file systems. For a diagram outlining these layers and some of the components within them, see Figure 3-1 on page ● 65.



3.2

Figure 3-1 The Layers of GEOS

GEOS contains several logical layers of abstraction, from device drivers to dynamic libraries to applications. The Kernel is the heart of the operating system, managing most interactions between device drivers and applications and libraries. Note that this is a simplified diagram, and not all system components are represented.



3.2.1 Applications

A GEOS application can take advantage of the various kernel and UI mechanisms to provide some set of functionality to the user. An application is made up primarily of two parts: The first part defines the application's user interface using objects and functions provided by the User Interface Library. The second part consists of the code of the application including procedures, functions, and object-specific code.

3.2

3.2.2 Libraries

GEOS libraries are bodies of code or collections of objects that are dynamically loaded into memory when called for by applications or by other libraries. The use of libraries reduces the amount of code within application resources and increases application performance. In addition, library routines and objects may be shared by several different applications at once, reducing duplicated code in memory.

Many standard libraries are provided with GEOS and are available to all developers. Among these are a text object library, a graphic object library, a an import/export library and others. In addition, developers can easily create their own libraries with the developer kit tools.

3.2.3 The Kernel

The GEOS kernel is the core of the operating system, providing all the basic services needed by the system. It provides multitasking, memory management, object creation and manipulation, windowing, process management, file system access, inter-application communication, message dispatching, and scores of basic graphic primitives. The kernel provides nearly 1000 routines usable by applications and libraries, though most applications will use only a portion of them.

3.2.4 Device Drivers

GEOS device drivers isolate the system and applications from the specific needs and foibles of a user's hardware. Drivers translate the generic actions of an application (such as opening a file) into the specific needs of the hardware or system in use (such as the specific call to MS-DOS to open the file). By isolating the system from hardware specifics, drivers allow GEOS to be expanded in the future to use new devices and engines without recompiling applications.

3.2

Drivers are provided with the system for printers, video cards, mice, keyboards, serial and parallel ports, file systems, task switchers, power management hardware, and font engines. In the future, developers will be able to develop their own drivers for new hardware and file systems.

Most applications and libraries will access drivers through kernel calls. Drivers are also accessible to applications and libraries directly.

3.2.5 The User Interface

The GEOS User Interface (UI) is a special entity. While it consists primarily of dynamic libraries, it can be thought of as a user interface driver.

Applications define their UI needs in a high-level language (called Goc and developed by Geoworks) interpreted by a special C preprocessor. Applications in general do not have to manifest windows, scrollbars, buttons, menus, or dialog boxes; instead, they use generic objects from the provided UI object library. These generic objects have no inherent visual representation; instead, they have a strictly-defined functionality and API.

A Specific User Interface library, at run-time, determines the visual representation each object takes depending on its context. For example, a GenInteraction generic object may appear as a menu, a dialog box, or a merely a grouping of other generic UI gadgets. Additionally, a menu GenInteraction may be represented as a vertical menu in one Specific UI but as a horizontal bar of choices in another.

Although such a system may appear to take some control of the application's user interface away from the programmer, in actuality quite the opposite is true. The use of a generic UI frees the programmer to concentrate on other

issues and features without worrying about whether the application conforms to one or more specifications. The programmer does not have to worry about window positioning or clipping areas (unless he or she wants to), and menus and dialog boxes are almost entirely automatic.

Additionally, the programmer can set *hints* for individual generic objects; hints can help the specific UI decide how the object should appear or be arranged on the screen. Not all hints are allowed in all specific UIs, so a particular specific UI may ignore certain hints while implementing others. For example, a dialog box that has three buttons within it may have the hint `HINT_ORIENT_CHILDREN_VERTICALLY` to indicate the buttons should appear in a column rather than a row.

3.3

3.3 Object-Oriented Programming

Object-Oriented Programming (OOP) is a popular way of organizing programs, especially programs using graphical user interfaces. GEOS is an object-oriented system, programmed entirely in Object Assembly, a Geoworks innovation. This section describes the concepts behind OOP and how it is implemented in GEOS. If you are familiar with OOP concepts, you may skip this section; because terminology can differ from system to system, however, you will probably want to at least skim the rest of the section. (A full discussion of GEOS messaging and object manipulation can be found in section 5.4 of chapter 5.)

Object-Oriented Programming is simply a way to organize code and data differently from traditional procedural programming. Anything done with procedural programming can be done in an object-oriented way and vice versa. However, OOP offers several significant advantages over procedural programming:

- ◆ **Modularity**
Especially in large systems, modularity can lead to cleaner, more manageable, and more easily expandable programs. OOP provides tight modularity with each component (*object*) having a strict API. This modularity also makes OOP systems ideal for the multithreaded environment of GEOS.

- ◆ **Simplicity**
People interact with things individually. Each object in our environment has a certain set of characteristics and certain behaviors. OOP systems use this point of view as a foundation for programs that are easy to understand.
- ◆ **Efficiency**
The efficient design of objects in an OOP system can lead to less repetition of reusable code. Several objects working independently may use the same code, thus reducing the impact on memory usage.
- ◆ **Reduction of code**
Because functionality (functions, routines, etc.) is inherited by object types from other object types (known as classes), it doesn't have to be implemented again and again. Instead, one version of the code can exist in a particular class and be inherited by other classes, thus reducing coding and debugging time.

3.3

3.3.1 Objects, Messages, and Methods

Procedural programming uses routines that act with globally-accessible or locally-defined data. Those routines must know about other routines in the program and must understand the data structure and organization used by other routines. Although some amount of isolation is applied to each routine, in essence they are all part of the greater whole of the program.

Objects, in their simplest sense, consist of data (*instance data*) and routines (*methods*) that act on that data. Objects may interact with each other but may not alter other objects' instance data; this rule leads to strict modularity and cleaner program design. Objects do not need to understand how other objects work or how another object's instance data is arranged; instead, only the interface to the other object is required.

Objects interact with each other via *messages*. Messages may indicate status, provide notification, pass data, or instruct the recipient to perform an action. When an object receives a message, it executes an appropriate *method*. A method may change the object's instance data, send messages to other objects, call kernel routines, or perform calculations—anything that can be done in a normal program procedure can be done in a method. (Note that occasionally the term *message handler* is used for *method*.)



Every object is represented by an *Object Pointer* (optr), a data structure that uniquely identifies the object's location in the system. This data structure is a combination of two special memory handles that provide abstraction of the object's location—this allows the object to be moved in memory or swapped to disk, or even saved to and retrieved from files. Object pointers are used to identify objects in many situations, the most common being when a message is sent. The intended recipient of the message is indicated by its optr.

3.3

Objects may interact with each other even if they are in different threads of execution. This is made possible by *message queues* and the kernel's message *dispatcher*. When a message is sent, it is first passed to the kernel's dispatcher with the recipient's optr. The dispatcher then puts the message in the recipient object's message queue. If other messages have been sent to the recipient but not handled yet, then the message will wait in the queue until the others have been handled. Otherwise, the message will be handled immediately. (For timing-critical messages, the sender can indicate that the message must be handled immediately; this is important in a multithreaded system.)

A thread is a single entity that runs a certain set of code and maintains a certain set of data. Only one thread may be using the processor at any given time; when a context switch occurs, one thread loses the processor and another takes it over. Each thread may run code for many objects, and an application can have several threads.

As an example, Figure 3-2 on page ● 71 shows the conceptual model of a Counter object. The Counter maintains a single instance data field, **currentCount**. It also has three predefined actions:

- ◆ **CounterIncrementCount()**
This method handles the message MSG_COUNTER_INCREMENT. Its purpose is to increment the **currentCount** instance data field. When the count reaches the maximum (in this case, 100), then the counter resets itself.
- ◆ **CounterReturnCur()**
This method handles the message MSG_COUNTER_RETURN_CUR. It returns the value of the **currentCount** instance data field.
- ◆ **CounterReset()**
This method handles the message MSG_COUNTER_RESET. It sets the **currentCount** instance data field to zero.

Messages can carry data (parameters) with them and can also have return values. For example, if the Counter in the example could be set to an arbitrary value (rather than reset to zero), the object sending the set message would have to indicate the value to be set. This would be passed in the same way as a normal C parameter.

If a message is supposed to return a value to the sender, the sender must make sure the message is handled immediately. In this case, the message acts like a remote routine call, blocking the thread of the caller until the message has been handled and the value returned.

3.3

Objects can also share data when necessary. Rather than share instance data, however, each object can have an instance data field containing the

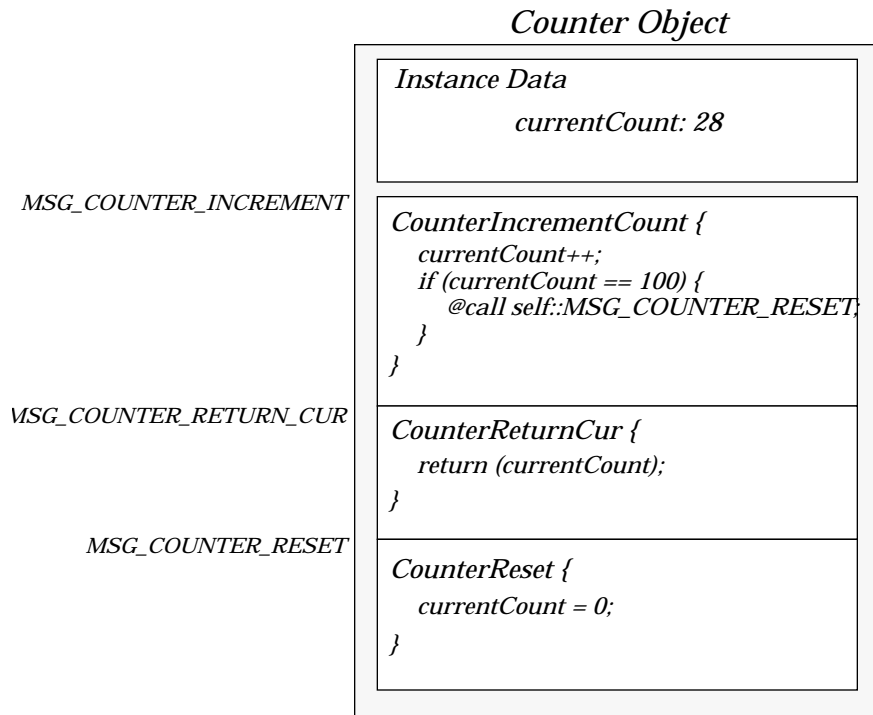


Figure 3-2 A Sample Counter Object

This counter object handles three messages and maintains a single instance data field. Any object may send the messages; in this way, a single counter can be used by several objects (not necessarily a desirable thing). Receipt of a message invokes the appropriate method.



handle of a sharable block of memory holding the subject data. Then each object could access the shared block through the normal memory manager. Handles are offsets into a table maintained by the kernel; this table contains pointers to the actual memory. This abstraction of the pointer allows the kernel to move the memory block around without forcing everything else to update their pointers. Handles can also be passed as message parameters, allowing large blocks of data to be passed along from object to object.

3.3

Figure 3-3 shows an example of two objects interacting via messages. In this example, the Requestor object requires the Calculator object to perform a calculation given two numbers. However, since the timing of the calculation

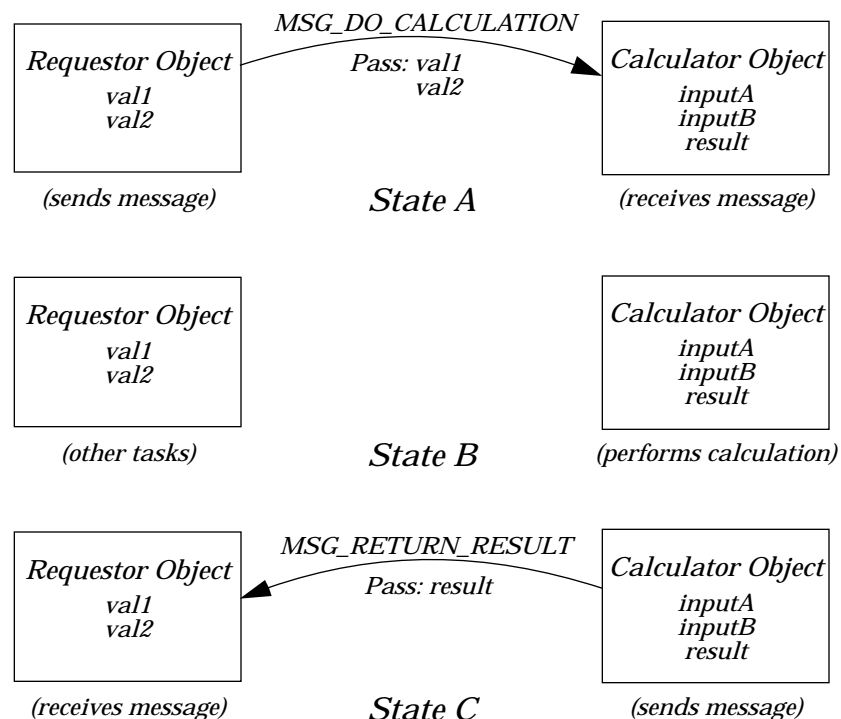


Figure 3-3 *Objects Using Messages*

The Requestor object sends *MSG_DO_CALCULATION* to the Calculator, passing the values in its instance data fields *val1* and *val2* (State A). The Requestor then does other tasks while the Calculator performs the calculation (State B). When the calculation is finished, the Calculator sends the result back to the Requestor with *MSG_RETURN_RESULT* (State C).

is not critical, the result is not returned but is instead sent back by the Calculator object in a subsequent message. This allows the Requestor object to continue with other tasks until the result is ready. (Note that if the Calculator object could be used by more than one requestor, it would have to also be passed the requestor's object pointer. Otherwise, it would have no idea where to send the return message. This example assumes the Calculator object is used by only the one requestor and inherently knows the requestor's optr.)

3.3

3.3.2 Classes and Inheritance

If every object had to be coded and debugged once for each time it was used, OOP would provide few benefits over standard procedural programming. Many objects share similar functionality and could easily make use of the same code over and over again. The concepts of *class* and *inheritance* allow objects with similar data structures and methods to use common code.

The class is actually where the functionality of objects is defined. Every object is simply an *instance* of a class, a manifestation of the instance data and methods defined for the class. For example, the Counter object of “Objects, Messages, and Methods” on page 69 could be an instance of a class called **CounterClass**. Other counters sharing the same characteristics would also be instances of **CounterClass**, each having its own value in its own instance data field.

A main benefit of the implementation of classes in GEOS is that objects can be created and destroyed during execution. Class definitions are stored separately from an individual instance's data; each instance knows where its class is located. Therefore, if two instances of a particular class exist, they both point to a single copy of the class definition. Since the code is stored in the class definition, the instances can use up only as much memory as their data fields require; each instance does not need to store its own method code.

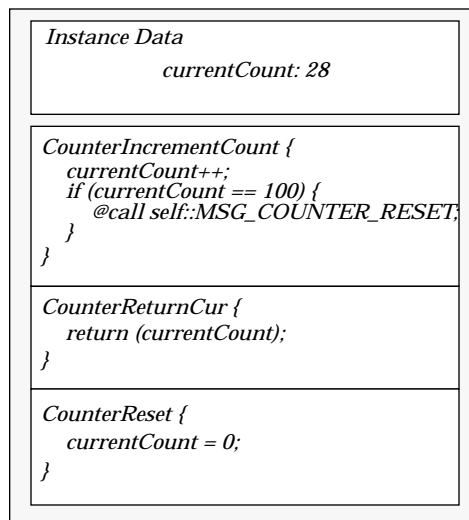
Another, more powerful, benefit is gained through the implementation of inheritance. In many cases, objects will be similar but not identical. This means that their classes are also similar but not identical. Inheritance overcomes this problem through the use of *superclasses* and *subclasses*.

A class provides a certain set of instance data and methods. A subclass inherits all the instance data structures and methods of its superclass, modifying or enhancing them to provide some different functionality.

Take, for example, the class **CounterClass**. An instance of this class would provide a counter with certain features: The counter goes from zero to 100, it can be reset manually, it resets automatically when it reaches 100, and its value is retrieved with the use of a certain message. However, suppose we need a counter that does all those things *and* allows the counter to be set to an arbitrary value with the use of a new message.

3.3

CounterClass



CounterSetClass = subclass of CounterClass

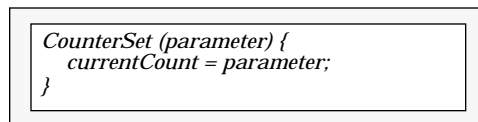


Figure 3-4 A Sample Subclass

CounterSetClass is a subclass of *CounterClass*, thereby inheriting the instance data field and the three methods of that class. It also creates a new method, *CounterSet*, that takes one parameter and sets the *currentCount* field accordingly.

Without inheritance, we would have to write all the code for **CounterClass** over again as a new class, adding the new method. However, by creating a subclass of **CounterClass** (let's call it **CounterSetClass**), we can inherit all the methods and data structures of **CounterClass** without recoding them. Instead, we simply add the new message and the code for the new method. (It is also possible to modify inherited methods in a subclass.) This example of **CounterClass** and **CounterSetClass** is shown in Figure 3-4.

The inheritance implemented in GEOS is actually much more complex and sophisticated than this simple example shows. You may never need to know more than the above concepts, however.

3.4

GEOS classes may be of a special type known as a *variant*. Variant classes do not initially know what their superclass is; instead, the superclass is determined by context for each instance of the variant class. This is a complicated topic that is discussed later in the documentation.

3.4 Multitasking and Multithreading

Multitasking is the ability to have multiple processes running simultaneously on a single system. In actuality, PCs have only one processor and only one task can run at any given moment; the operating system must manage the processor and allocate time to all the different programs running at once.

A *thread* is a single entity that runs in the GEOS system. Threads may be event-driven or procedural. An event-driven thread runs objects and has a message queue (also called an event queue). The thread receives messages for its objects and is only active when the objects are handling the messages. If an event-driven thread never receives a message, it will never use the processor. Procedural threads execute sequential code such as functions or procedures in C. Procedural threads do not have a message queue and do not run objects.

Cooperative multitasking, used in some operating systems, requires all processes running to cooperate with the operating system. The system has a

routine which each process must call periodically; this routine (called a Context Switch routine) checks if any other processes are waiting to run. If other processes are waiting, priorities are checked and the process with the highest priority takes over the processor.

GEOS uses *preemptive multitasking*, in which threads are given the illusion that they have nonstop access to the processor. A preemptive system periodically interrupts the thread running and checks for other waiting threads; if there are any, the system will switch context automatically. Programs do not have to call a context switch routine, a cumbersome requirement in many cases.

GEOS also maintains a knowledge of thread *priority*. Every thread (including the kernel and user interface) has a base priority and a current priority; the base priority rarely changes, while the current priority decreases with recent processor usage. GEOS maintains these priorities intelligently, making sure that no thread ever uses the CPU more than its share of the time.

Any application running in GEOS may have multiple threads. This is useful, for example, in a spreadsheet application, where complex and time-consuming calculations may be done by a background thread that wakes up only when calculations need to be made. Use of multiple threads can make the application appear to be extraordinarily fast; the user can continue to navigate menus and use other UI gadgetry while the application is also doing something else.

3.5 The GEOS User Interface

The GEOS User Interface (UI) maintains the interaction between each application and the user. It provides several necessary and many extra services for applications, relieving programmers from building much of the basic UI functionality into their applications. The structure, components, and services of the GEOS UI are outlined below.

3.5.1 The Generic User Interface

A User Interface (UI) specification is, essentially, a set of rules and conventions used to determine the interaction between a user and an application. Over the years, UI specifications have evolved from basic command-line shells to the standard graphical and pen-based systems in use today. No doubt you have come across several graphical UI specifications currently in use—Macintosh, Open Look, Presentation Manager, and OSF/Motif are a few examples.

3.5

Each of these specifications represents a particular implementation of the basic functionality required for application–user interaction. The basic functions are similar; only the implementation is different. Therefore, each of these specifications is referred to as a *Specific UI*.

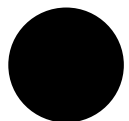
GEOS employs a *Generic UI*—an implementation-independent way of defining an application's user interface. To the user, this means a single application executable can be used in any of several specific UI implementations. To the developer, this means relief from many of the UI definition tasks normally associated with graphical user interfaces.

The Generic UI can be thought of as a UI driver—a layer of abstraction between the application and the UI implementation. The programmer defines his needs in terms of basic functionality and user involvement, leaving the actual implementation (decisions about color, shape, menu structure, etc.) up to the system at run-time.

The application defines its UI by selecting certain objects from the Generic UI Library. Each of these objects has certain *attributes* that determine its specific implementation. In addition, each generic object may be assigned *hints*. Hints help the Specific UI Library determine the proper context and subjective information required to fully flesh out the proper manifestation of each generic object (for example, a GenPrimary window object can have special sizing restrictions suggested with various hints such as HINT_FIXED_SIZE and HINT_INITIAL_SIZE).

3.5.1.1 Attributes

Attributes represent objective information which does not change with the specific UI in use. In general, attributes determine the basic functionality of



the object regardless of its manifestation on the screen. For example, a GenView may have its scrollable attribute set; this means that in all Specific UIs, the View will have some sort of scrollers. However, one Specific UI might implement the scrollers as bars and another might implement them as dials or cranks.

Attributes are always allocated space in an object's instance data regardless of whether the attribute is set. For example, all GenView objects have a bit specifying whether it is scrollable or not. Attributes are used, therefore, primarily for determining the object's functionality rather than its appearance.

3.5.1.2 Hints

Hints represent subjective or contextual information about objects. Hints are different from attributes in two main aspects: First, hints may be ignored by certain Specific UIs. Suppose, for example, that a given Specific UI only allowed horizontal ordering of components within dialog boxes. This specific UI would ignore the hint `HINT_ORIENT_CHILDREN_VERTICALLY` applied to any GenInteraction object implemented as a dialog box.

Second, hints are dynamically added to each object's instance data in a special format known as *variable data*. Variable data fields are not allocated within the object's instance data unless they are used. Thus, objects with fewer hints use less memory. Attributes, on the other hand, must exist in the object's instance data whether they are used or not, and they therefore require a given amount of memory.

3.5.1.3 The Generic UI Classes

The Generic UI Library contains a number of object classes that implement nearly all the UI functions an application will ever need. For a full description of the API of each of these classes, see the Objects Book. Some of the more common implementations (e.g. menus, dialog boxes, and scrolling views) of these objects are outlined in the sections below.

3.5.2 The Scalable User Interface

As application technology advances, applications gain more and more features. Despite recent improvements in user-friendliness, however, over-featured applications can sometimes intimidate and frustrate users.

GEOS therefore implements a *Scalable User Interface*, a system in which a single application's source code can be compiled into different versions of the application, each version at a different level of complexity. Scalability of applications is shown in Geoworks Pro—the appliances are all created from exactly the same source code as the advanced versions of those applications.

3.5

3.5.3 Windows and Window Management

A window is an object that defines an area on the screen in which drawing and user interaction can occur. Windows may be resizable and movable, and automatically clip any drawing so it does not appear outside the window's edges.

The GEOS Window Manager maintains information about the size, shape, and visibility of all the windows in the system. In general, applications do not access the Window Manager directly; instead, generic UI objects and the kernel handle all window manipulation. Several different types of generic objects are “window” objects: GenPrimary is used as an application's primary window, GenInteraction is used for dialog boxes and menus, GenDisplay displays documents in a scrollable window, and GenView is used to create a scrollable graphics window into which an application can draw.

The Window Manager is closely tied to the GEOS Graphic System. These two parts of the system handle all window resizing, reshaping, drawing, and clipping automatically. They also interact with the User Interface's input mechanism to ensure that the proper windows receive the proper input. Applications do not need to know the shape, position, or size of a window when drawing inside it.

Applications can also allocate one or more windows in which they can present data (text, graphics, or other information). Each of these windows is called a View and is an instance of the GenView object (see below). The View manages all scrolling, scaling, sizing, and clipping automatically for the application;

programmers don't have to worry about managing scrollbars or window size in order to ensure the proper portion of the data is drawn. Additionally, applications can modify any of the scrolling, scaling, or sizing behavior with a little effort to provide custom functionality.

3.5.4 Input

3.5

GEOS has an input manager that tracks the mouse, gets keyboard input, and passes the input events on to the proper windows and objects on the screen. The input manager is part of the UI.

The UI keeps track of three different active objects in the system: the Focus, the Target, and the Model. The focus is the active object that should receive all keyboard input. The target is the currently selected object with which the user can interact. The model represents a non-visible “selection” that can be maintained by the application. These three active objects can be accessed directly by other objects (e.g. menu items); for example, a “change color” menu item may want to work on whatever object is currently active—it would want to change the color of the target object, and by contacting the target directly, it does not have to know what object is the target.

Any object in the system can *grab* mouse or keyboard events. When an object has been granted a mouse grab or keyboard grab, it will receive the events until it relinquishes the grab. This fact is most useful when dealing with visible object trees inside a View (see GenView, below)—generic objects handle input automatically (including the Text objects).

Applications may or may not need to handle input directly; complex applications likely will want to filter certain input or handle special cases. Simple applications or utilities will probably not have to deal with input (keyboard or mouse events) directly at all.

3.5.5 Menus and Dialog Boxes

Among the generic objects of the GEOS UI library is GenInteraction. This object can be used to implement menus, dialog boxes, and error boxes, and it can also be used for grouping and arranging other generic objects (such as triggers, lists, etc.). It provides extreme flexibility and functionality, and

experienced programmers can usually get a menu and dialog structure up on the screen quickly just by setting a few attributes.

Remember that because the Interaction is a generic object, the Generic UI and Specific UI libraries will translate it into its proper implementation at run-time. The Interaction serves the primary purpose of grouping objects; the grouping may be implemented in various ways (e.g. a menu or a dialog box) depending on the attributes and hints applied.

The sections below outline some of the practical functionality of these objects; for a full description of what these do and how they can be used, see the Object Reference Book.

3.5

3.5.5.1 Menus

Menus in GEOS are subject to the rules and conventions of the specific UI in use by the user. However, several basic concepts are supported. (See Figure 3-5 for an illustration of sample menus.)

◆ Standard menus

Some specific UIs have differing menu structures. Therefore, several standard generic menus (e.g. the File menu) can be implemented by giving the menu a standard set of attributes. The UI will automatically build and manage this menu according to the UI specification in use at the time.

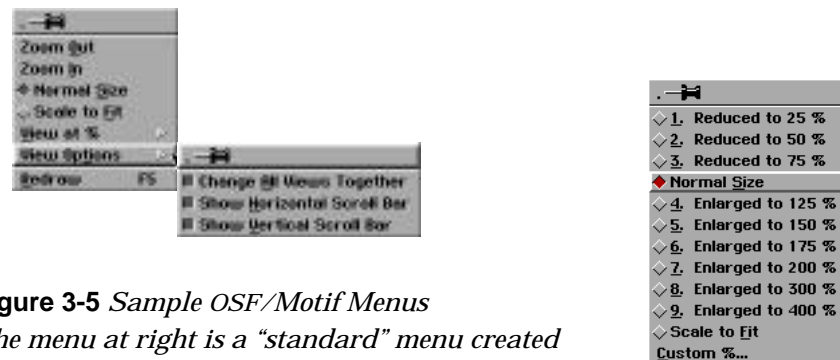
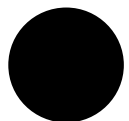


Figure 3-5 Sample OSF/Motif Menus

The menu at right is a “standard” menu created almost entirely by the UI. The menu above shows cascading menus as well as lists. Items followed by ellipses bring up dialog boxes.



- ◆ **Cascading or nested menus**
Because Interactions may be nested, cascading (or nested) menus are easy to implement. Cascading menus are useful if an application has many menu items that may logically be grouped into different areas of functionality.
- ◆ **Keyboard accelerators**
Because the UI manages both input and menus, you can easily specify accelerator keystrokes for each menu item. When the user hits the proper keystrokes, the same action is performed as if the user had clicked on the associated menu item. This happens automatically without additional application code.
- ◆ **Keyboard menu navigation**
Every menu item may have a character that the user may press during keyboard navigation. This is important because it allows users to use your entire application without having a mouse.
- ◆ **Lists and checkboxes within menus**
Menus may contain not only triggers but also lists of items. Lists are implemented with the `GenItemGroup`, `GenItem`, `GenBooleanGroup`, and `GenBoolean` generic objects. Lists may be exclusive or not, and each entry may have a checkbox indicating its on/off status. Again, all these functions are determined by the attributes set for the interaction and item objects.
- ◆ **Menu items that bring up dialog boxes**
By setting certain attributes in an interaction, you can create menu items that automatically bring up dialog boxes. The UI will put a trigger in the menu representing the dialog box, and when the trigger is activated, the UI will put up the box.
- ◆ **Pin-up menus**
In the OSF/Motif implementation of GEOS, all menus are automatically given the ability to be pinned in place and moved around the screen. Because this is a function of the specific UI, applications are completely unaware of the fact.

3.5.5.2 Dialog Boxes

Dialog boxes are standard ways of having an application interact with the user. For example, a dialog box may contain a number of controls that

determine how the application displays its data. See Figure 3-6 for an illustration of sample dialog boxes.

Dialog boxes, like menus, are implemented through the use of GenInteraction objects. Applications may also call a kernel routine that will put up standard dialogs in certain situations (e.g. errors or warnings).

Several features of dialog boxes are implemented automatically with very little additional code in the application:

3.5

- ◆ **Automatic geometry management**
When a dialog box contains generic objects (e.g. triggers, lists, and text fields), all geometry is handled automatically. By using various hints, you can modify the normal organization, but sizing and placement of the generic objects and the dialog box are entirely automatic.
- ◆ **Standard response triggers and reply bars**
Just as there are standard menu types, UI specifications often include rules for dialog box response triggers (e.g., the “OK” button on the left and a “Cancel” button on the right). Standard response triggers normally appear in an area of the dialog box known as the “reply bar.” You can set up a reply bar with standard attributes that will automatically be implemented properly by the UI. As an alternative, you can even use pre-defined dialog box types that have their own reply bars.
- ◆ **Modality**
Sometimes, dialog boxes require a user action before the application or system can continue with other tasks. Dialog boxes can have various modality states: system-modal, which does not allow the user to interact with anything but the dialog box; application-modal, which allows the user to switch to other applications but disallows interaction within the

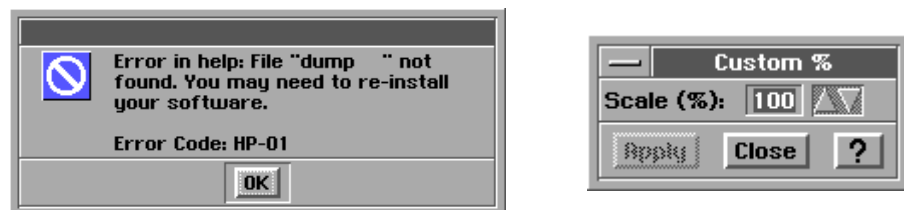


Figure 3-6 *Sample OSF/Motif Dialog Boxes*

The above dialog box is a standard, modal dialog. That to the right demonstrates the concept of the standard response triggers and automatic geometry management (the box is sized to fit all the objects).

dialog's application (except for the dialog box); and non-modal, which allows the user to switch to any other part of the system.

- ◆ **Standard dialog types**

To increase the amount of standardization across all applications, GEOS includes several standard dialog box types accessible through the use of UI routines. These include errors, warnings, and questions. Also, you can call up custom dialog boxes with special kernel routines.

3.5

3.5.6 Scrolling Views

Although the basic generic objects provided by GEOS are powerful and useful in many situations, applications still need a way to display their own specific data. In most systems, this consists of allocating a window for the application's use and then implementing additional functionality for that window—scrolling, scaling, and document management.

The GenView generic object does all these things for an application. The application can allocate any number of views for its own use, and each view can display anything from regular graphics to hierarchies of visible objects.

Most applications will use at least one view. Its power and flexibility give programmers the opportunity to concentrate on their own application's functionality without having to worry about display issues such as scaling, clipping, and scrolling—they are all handled automatically. Some of the features of a view are listed below (see “GenView,” Chapter 9 of the Object Reference Book for more detailed information):

- ◆ **Automatic clipping and updating**

When an application uses a GenView, the programmer can forget the worries involved in figuring out which portions of his document need to be redrawn when. The GEOS Window Manager will automatically clip the document to the View's boundaries and will notify the application whenever a portion of the View has become invalid (whether by scrolling, sizing, moving, or being uncovered by movement of another object). The application simply must draw the document whenever notified that a portion has become invalid; it does not have to do any calculations regarding what is visible on the screen (though it might if the document is especially complex).

- ◆ **Automatic scrolling**
By setting two attributes in a GenView's definition, an application can automatically make its view scrollable. The UI will automatically create scroller objects that will interact directly with the view—the application never has to know how far or in what direction the view has been scrolled. It simply will receive a message saying that the window has been invalidated and needs to be redrawn.
- ◆ **Automatic scaling**
The view provides automatic scaling of the contents of its window. The application may request scalings or include a standard controller to let the user control this transparently to the application. 3.5
- ◆ **Automatic sizing**
The Geometry Manager will automatically size the view to fit within its parent window. When the parent is resized, the view will follow. This process is transparent to applications. However, applications can customize this behavior by setting a fixed or desired size or by altering the sizing behavior of the view. They can also override this to make the parent window follow the view's size.
- ◆ **Comprehensive input management**
Many applications will display objects within a view. In many cases, these objects will require input (e.g. the cards in Solitaire). The view gets input events from the User Interface and can pass them on to the proper objects in the hierarchy.
- ◆ **Flexibility**
All the functions described above may be customized. Scrolling, for example, can be tracked and altered. Custom scroller objects can be defined in place of the automatically-generated scrollers. Sizing behavior can be adjusted. If necessary, you could get the view's bounds in order to draw only the visible portion of your document (though this often does not increase drawing performance significantly).
- ◆ **Customizable background color**
Each view has a default background color determined by the specific UI. This can, however, be set to any RGB value or to any GEOS color index. This is used, for example, in Solitaire to create the green background (so the application does not have to draw green under all the other objects).

3.5.7 Visible Object Classes

Many applications will need to create objects that draw themselves and interact directly with the user. Rather than forcing the programmer to create each of his objects from scratch, GEOS offers several visible object classes that already understand several system services and constructs. When an application needs a visible object, it simply defines a subclass of one of these visible classes, thus ensuring that his object will handle all the UI messages that may be sent to it.

3.5

3.5.8 Geometry Manager

The GEOS User Interface includes a sophisticated Geometry Manager that does all the calculations for organization of objects on the screen. The Geometry Manager manages the positions of generic UI objects in dialog boxes and windows, for example, and it can be used to automatically position visible objects within a view.

3.5.9 Lists

The GenItem, GenItemGroup, GenBoolean, GenBooleanGroup, and GenDynamicList objects can be used to provide several different types and styles of lists. Lists may appear within menus or dialog boxes. The actual visual implementation of a list depends on the specific UI in use; however, there are several basic types of lists available (for complete information, see “The List Objects,” Chapter 11 of the Object Reference Book):

- ◆ **Dynamic lists**
Lists can be either static (a set number of elements) or dynamic (a varying number of elements).
- ◆ **Scrolling lists**
Scrolling lists are usually implemented for long lists or dynamic lists.
- ◆ **Exclusive lists**
Exclusive lists have one and only one element chosen at all times.

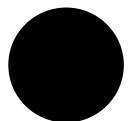
- ◆ **Non-exclusive lists**
Non-exclusive lists may have any number of elements set at a time. Each element is set or reset independently of the others.
- ◆ **Non-exclusive lists with override**
Non-exclusive lists with an override have a non-exclusive list with an additional element that overrides all the others (an example is the text styles menu—the “Plain” option overrides each of the others).

3.5

3.5.10 Other Gadgets

Besides the objects listed above, the generic UI supplies a number of other gadgets. These include

- ◆ **Text Display and Edit Objects**
The GenText object provides a full-featured word processing object. It can be used for every text function from displaying a word to implementing a simple yet full-featured word processor. It supports multiple styles and paragraph attributes, character and paragraph spacing, and all the other features supported in the Text Library (see section 3.6.6 on page 98.)
- ◆ **Triggers**
The GenTrigger object implements all the functionality of a simple push-button. When a trigger is pressed, it sends a message to a pre-specified object.
- ◆ **Value**
The GenValue object allows the user to set a value within a given range.
- ◆ **Controllers and Tool Boxes**
The GenControl and GenToolControl objects allow an application to set up controllers that provide standard menu entries and their associated features. For example, the GenEditControl object creates the standard Edit menu with Cut, Copy, and Paste triggers. The GenToolControl object allows the user to specify which items in which controllers may be displayed within a floating tool box.
- ◆ **File Selector**
The GenFileSelector provides all the features of a file selector dialog box that allows the user to traverse their file system's directories and select a file.



3.5.11 Managing Documents and Files

Most applications will save data files. GEOS provides a convenient storage format for data files and an even more convenient mechanism for managing the files.

3.5

The Document Control objects interact with the GEOS file system and virtual memory system to implement the Open, Close, Save, Save-As, and Revert functions of the File menu as well as many other common document-related features. In fact, the Document Control also controls user interaction for these basic functions, creating and updating the File menu and putting up the proper dialog boxes for each operation.

The Document Control objects, like GenView and other components of GEOS, take basic functionality common to most applications and implement it in the system software. Using a Document Control for your data file management can save weeks of coding and debugging time.

3.5.12 Multiple Document Interface

Some applications may benefit from allowing the user to have multiple data files open at any given time. GeoWrite and GeoDraw, for example, allow the user to work with multiple documents at once. Each document appears within its own window; each of these windows is a GenDisplay object within a Display Control object.

The Display Control objects, like the Document Control objects, allow applications to use multiple data files with a minimum of extra coding. The Display Control works closely with the Document Control to ensure that the proper data files are being operated on at all times. The Display Control also creates and maintains the Window menu, just as the Document Control creates and maintains the File menu.

3.5.13 Clipboard and Quick-Transfer

A common feature of GUIs is the Clipboard. Users can copy or cut data out of documents for pasting into other documents (or even the same document) later. When data is cut or copied, it is placed in a transfer item, a special data

structure used by the Clipboard. All applications can access the Clipboard via Cut, Copy, and Paste; in addition, the Edit menu can be implemented automatically with the use of a GenEditControl object.

GEOS also offers another method of transferring data—the quick-transfer mechanism. In OSF/Motif, the user clicks the move/copy button (in most cases the right mouse button) over some selected data. He can then drag to another area and drop the selected data, either moving or copying it depending on certain contextual factors. Applications may or may not support the quick-transfer mechanism, though supporting it is not difficult and is strongly encouraged.

3.5

Both the Clipboard and the quick-transfer mechanism are supported automatically by all text edit and display objects. Applications may define their own custom data formats (transfer items) for use with both the Clipboard and the quick-transfer mechanism.

3.5.14 General Change Notification

In a multithreaded system, one thread may change some information that another thread depends on. In this case, the thread that makes the change must notify all other users of the information that it has been changed and that they must update their status appropriately.

GEOS provides a mechanism for allowing all threads or objects to register for notification of certain events. For example, GeoManager is notified of all file system changes so it can update its display in real time. Although notification is automatically sent for certain system changes, applications can set up an automatic notification of custom changes. For example, if a multiuser network game depended on each user knowing a certain status flag, automatic notification could be set up to notify all the users whenever any user changed it.

3.5.15 Help Object

On-line help is a convenient and powerful way to provide documentation to your users. Because graphical user interfaces are easy to understand, many



users do not read documentation and prefer to discover an application's features on their own. On-line help can speed their learning process up.

GEOS provides an object that you can include in your User Interface that will automatically create a "Help" icon or menu item and display help text when that item is invoked. The help text displayed will be context-sensitive as defined by the application. The application simply has to provide the help text and a few other attributes of the help object, and the system will take care of displaying the help when the user requests it.

3.6

3.6 System Services

GEOS provides a number of services useful or necessary to applications. These services range from dynamic memory management to an item database to a sophisticated graphics system and a print spooler. Many of the most useful or integral services are described in this section.

3.6.1 Memory

GEOS uses all RAM available to the system, even expanded and extended memory. Memory is managed by the kernel and is accessed in the segmented scheme implemented by the 8086 processor. (The protected mode of the more advanced processors is not supported in the Version 2.0 system software but is expected to be in future releases.)

The GEOS Memory Manager is sophisticated. It uses dynamic allocation and access of blocks on a global heap to provide high performance, optimized to run very efficiently even on systems with only 640 K.

3.6.1.1 Handles

GEOS maintains control of all the memory, objects, and other entities in the system through the *Handle Table*. The Handle Table is a section of memory set aside for kernel use, and it contains a number of entries, each 16 bytes that can contain various information about many different items in the system. These entries are accessed by *handles*, 16-bit offsets into the Handle

Table. Applications and libraries may use handles but may not access the contents of a handle table entry. The data structure is opaque to all but the kernel.

Handles are used for many different things. They can reference threads, memory blocks, VM blocks, or files; they can represent data structures such as timers, queues, semaphores, or events (messages); and they can be used by the kernel for optimization (such as when several words of data are passed with a message).

3.6

Through the Preferences application, a user can set the number of handles in the Handle Table. However, the size of the table does not change during a single execution of GEOS. Applications and libraries may allocate handles dynamically for most of the above-mentioned purposes (e.g. memory and file reference).

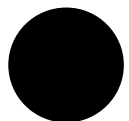
3.6.1.2 The Global Heap

A certain amount of memory is set aside for the file system (DOS, typically), the GEOS kernel (approximately 64 K), and any TSRs the user may have running on his system. Above this is the GEOS handle table. All RAM left (up to 1 megabyte) after these requirements are satisfied is used for the *global heap* (RAM over 1 megabyte is used as preferred swap space). The global heap is the space available to all the applications running in the system.

Applications allocate dynamic *blocks* of memory on the heap. Blocks may be of arbitrary size between 16 bytes and 64 K. Each block is designated a handle, an offset into the handle table where information about the block (such as size and location) is stored. Because blocks may be shuffled in the heap by the Memory Manager, applications must use handles as indirect references to the blocks; applications should not save pointers to specific locations in memory.

GEOS uses four basic types of blocks:

- ◆ **Fixed**
A fixed block will never move or swap out to disk. Having many fixed blocks can seriously degrade system performance. Fixed blocks are generally used for an application's global variables and primary code.



- ◆ **Moveable**
Moveable blocks may be shuffled around the heap. Moveable blocks may also be designated discardable and/or swappable (below). These blocks must be locked into memory before they can be accessed (see below).
- ◆ **Discardable**
Discardable blocks may be flushed whenever the Memory Manager requires more memory.
- ◆ **Swappable**
Swappable blocks may be swapped at any time by the Memory Manager. When a subsequent access is made to a swapped block, the Memory Manager will automatically read the block back into memory.

3.6.1.3 Allocating Memory

The GEOS kernel provides several routines to allocate memory. Some memory is allocated automatically, such as memory for code resources as they are loaded in. Other memory, however, must be allocated as it is needed during execution—for example, as the user types more and more text into a word processor. There are three basic ways to allocate memory on the heap:

- ◆ **MemAlloc()**
This routine allocates a new block on the heap for an application's use.
- ◆ **MemReAlloc()**
This routine reallocates a given block; this is useful for adding memory to a block already allocated.
- ◆ **malloc()**
Although use of **malloc()** may help in porting previous C code to GEOS, it is discouraged. The **malloc()** routine will allocate small amounts of memory within a fixed resource. Extensive use of **malloc()** leads to large, fixed blocks on the heap, degrading system performance.

When an application is done with a memory block, it can free the block with the routine **MemFree()**. This will allow the Memory Manager to free up that memory space and re-use the block's handle if required.

3.6.1.4 Accessing Memory

If a block is allocated as fixed, an application can use a far pointer to access any byte within the block. However, because fixed blocks are not always allocated in the same portion of memory each time an application is loaded, applications should *not* save pointers as state information.

Non-fixed blocks, however, can not be accessed by far pointers without locking them into their position in memory. **MemLock()** will take a block's handle and lock the block, thereby assuring that the Memory Manager will not move it in the middle of an access. **MemLock()** provides its caller with a far pointer to the block. When access to the block is finished, the thread that locked the block must call **MemUnlock()**, which marks the block as unlocked so it may once again be moved or swapped.

3.6

3.6.2 Virtual Memory

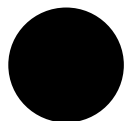
GEOS implements a powerful virtual memory concept to accomplish both memory block swapping and data file storage. Virtual memory (VM) can be thought of as a disk-based heap—a VM file is segmented into blocks, each of which is designated a VM handle analogous to a global memory handle. When a VM block is required by an application, it is locked into memory from the file with a call to the kernel routine **VMLock()**, which loads the VM block into memory and locks it on the global heap.

VM is extremely useful for data file storage; indeed, the Document Control objects use it extensively.

3.6.3 Local Memory and Object Blocks

Often applications will want to store large numbers of very small data items. In these cases, it's cumbersome and inefficient to allocate a global handle and a block on the global heap. For these situations, GEOS offers the concept of *local memory* (LMem).

Local memory can be thought of as a mini-heap within a memory block. Each individual unit allocated in this mini-heap is known as a *chunk*, and each chunk is designated a *chunk handle*. Thus, to access any piece of information



within an LMem block, you need to have the block's global handle and the item's chunk handle.

Chunks may be allocated or freed, and any chunk may be moved within an LMem heap at any time (e.g. when the LMem heap is compacted). Locked LMem heaps may have chunks shuffled only when space is allocated within the heap. Access to chunks is gained by locking the LMem block and dereferencing the chunk handle. Chunks may be resized, and bytes may be added to or removed from chunks.

3.6

LMem blocks are used primarily for two purposes: Database files and object blocks. Database files are managed by the Item Database Library (see below). Object blocks are used extensively by the UI and applications; object blocks store objects in memory and often are attached to VM files.

Each object in an object block occupies one chunk. Thus, the combination of global handle and chunk handle of an object is known as the *Object Pointer* (optr) of the object. The optr is used to identify the object uniquely in the system for all types of access.

3.6.4 Item Database Library

The GEOS database library allows applications to use LMem heaps and VM files together easily and cleanly. The database library allows applications to add a layer of abstraction to their data files.

An application may allocate database files, logical groups of items within those files, and individual items within the groups. The database library automatically manages the LMem and VM blocks that implement the database, keeping them desirable sizes and moving them into and out of memory as needed.

Additionally, items may be allocated as “ungrouped,” in which case the database manager will intelligently distribute these items among the various VM blocks in the database file for efficient access.

3.6.5 Graphics System

The graphics system provided in the GEOS kernel is extremely powerful and is designed to make creating graphics simple and fast for applications. It includes some advanced concepts and state-of-the-art technology that make GEOS rise well above most other GUIs.

3.6.5.1 The Coordinate Space

3.6

The GEOS graphics system uses a single imaging model like that used by PostScript. Applications and libraries draw their graphics on a generic rectangular grid with a resolution of 72 points per inch. When drawing to the screen or a printer, the graphics system will translate the graphics commands to the highest resolution of the output device, ensuring true WYSIWYG output. This system allows programmers to specify their graphics in real-world coordinates without worrying about the size of various resolution devices.

Normal graphics coordinates are represented by signed, 16-bit integers. Normal documents can be up to about nine feet on a side. Additionally, the graphics system allows applications to use an extended coordinate system (called the *large document model*) of signed, 32-bit integers. Large documents can be up to 900 miles on a side! Large documents, however, can incur a certain amount of additional overhead; if you do not need to use large documents, you probably should not.

3.6.5.2 Graphic States

To simplify complex drawing, the graphics system maintains a *graphics state*, or *GState*. GStates can be created or destroyed dynamically to allow several different GStates for a given window (only one may be active at any given moment, however).

The GState is essentially a data structure that contains all the relevant information about the current state of a window. The GState's information includes current color, drawing position, angle of rotation, scale, translation in the coordinate space, line and fill attributes, clipping paths, text attributes, and other items.

Graphics commands are issued relative to a particular GState. For example, the command **GrDrawEllipse()** must be passed a GState so the kernel knows exactly how to draw the ellipse; if the current GState is rotated 45 degrees and has green as its current line color, a green-outlined ellipse will be drawn rotated 45 degrees.

Programs may apply transformations to GStates—rotation, translation, or scaling. You can also define custom transformation matrixes to apply to your GStates if complex operations (e.g. shearing) are required.

3.6

3.6.5.3 Graphic Primitives and Graphic Objects

GEOS provides a complete set of graphics drawing primitives including lines, arcs, Bézier curves, splines, outline-defined text, rectangles, ellipses, polylines, polygons, and bitmaps.

GEOS also has a standard library of graphic objects such as those used in GeoDraw. These objects (such as rectangle and polygon objects) already contain all the code necessary to draw themselves, to respond to user input, and to resize, reshape, reposition, and rotate themselves. These objects are available to all applications and other libraries, and they provide a powerful base of user-interactive graphics tools.

3.6.5.4 Paths

A graphics *path* in GEOS is a continuous trail that defines the outline of an area. A path is an outline description of an arbitrarily shaped area, useful when an application must define a mathematically-precise shape. Paths may also be filled, combined with other paths, and scaled to any size without loss of resolution.

One example of the powerful application of paths is creation of arbitrary clipping regions. It is possible, for example, to clip drawings to an ellipse, to a Bézier curve, or even to text.

3.6.5.5 Regions

Regions perform essentially the same function as paths—definition of an arbitrarily shaped area or clipping region. However, because regions are

defined as resolution-dependent, they are typically used only for optimized drawing of UI gadgetry.

3.6.5.6 Graphics Strings

A *graphics string* (or *GString*) is a collection of graphics commands; GStrings are useful for saving complex graphic operations and playing them back later. GStrings are created by calling a special routine and are then filled by executing graphics commands just as if you were drawing to the screen. GStrings may be pre-defined in source code as data resources to be played during execution.

3.6

GStrings are extremely useful for sharing graphic data between processes. For example, applications use GStrings to define their program icons, which may then be displayed by File Manager applications. The Clipboard supports the GString data format for cut, copy, and paste operations. GStrings are also flexible—they may contain comments and may be set up to be executed with parameters.

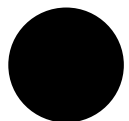
3.6.5.7 Bitmaps

Bitmaps are pixel-by-pixel images defined at a specific resolution; they are automatically scaled to match the resolution of the display. Bitmaps are used in general to define complicated pictures, usually with many colors.

GEOS allows the creation of bitmaps with specified resolutions. Bitmaps can be created off-screen and may be edited with standard graphics system commands. Bitmaps are used mainly as monikers of generic objects and as icons for applications and data files.

3.6.5.8 Color

GEOS allows applications to specify colors in two ways: with the system *palette* and with RGB values. The palette contains 256 entries, each of which is given a standard RGB value. If an application only uses 256 colors, it can modify the palette so it has the colors it needs. If an application uses more than 256 colors, it can specify any RGB value for any given graphics operation. The palette may also be expanded to support other color models. If the system palette is not sufficient, each application may create its own.



3.6.6 Text

Text in GEOS is handled by the Text Object, a sophisticated and powerful object that is the base for both the GenText and the VisText classes. The Text Object handles more situations than most applications will ever face; only powerful word processors and text-formatting programs may require more functionality. Those that require additional features can subclass the text object and add those features.

3.6

The Text Object implements text editing, manipulation, and display for GEOS. It represents a major portion of the system software and as such should be used by all applications that plan on providing any text-editing features. Among its many features are

- ◆ **Display and Text Wrapping Within Editable Bounds**
The Text Object is aware of its visible bounds and will wrap text appropriately to fit within the bounds. If the Text Object is given a set width, it will wrap text appropriately to that width.
- ◆ **“Editable” and “Selectable” Attributes**
The Text Object allows users to select and edit text. It can also act as a display-only object or a display-with-select object.
- ◆ **Character and Paragraph Formatting**
Sophisticated and extensive formatting options are included with the Text Object such as: Font selection, text styles (underline, bold, italic, etc.), text size (from four to 792 points), adjustable track kerning, text color, justification, adjustable margins, tabs with settable features (such as leaders, lines, and justification), paragraph borders, paragraph background colors, and paragraph spacing. All of these formatting options are settable and changeable.
- ◆ **Full Keyboard Editing**
The Text Object implements all the features necessary for editing text with the keyboard.
- ◆ **Embedded Graphics**
The Text Object can hold embedded graphics and graphics strings and display them properly.
- ◆ **Support for the Clipboard and Quick-Transfer**
The Text Object automatically supports the quick-transfer mechanism of the UI and handles the cut, copy, and paste commands of the Edit menu.

- ◆ Many other powerful features
For full feature listings and how to use the text object, see “The Text Objects,” Chapter 10 of the Object Reference Book.

3.6.7 Print Spooler and Printing

GEOS contains a print spooler that manages printing for all applications in the system. The spooler executes in its own threads and manages a print output queue for each printer connected to the system. The spooler also provides standard user interface components such as the print dialog box. Applications can customize the print dialog box by adding their own gadgetry for certain types of printing (such as the customizations in GeoDex and GeoPlanner).

3.6

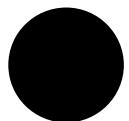
For the application, printing is essentially the same task as drawing to the screen; all the same graphics commands are used. The only difference is that the output is sent to a spool file rather than to the screen.

If the paper size in a user's printer is different from the size of the document being printed, the spooler will do all the calculations and operations necessary to print the document properly. For example, a document eleven inches by seventeen inches will be rotated and printed on two standard-sized sheets of paper.

3.6.8 Timers

The GEOS kernel allows applications and libraries to set up different types of timers that can be used for various purposes. When a timer ticks, it calls a routine specified by the geode that initiated it. The different types of timers that can be set up include

- one-shot** A timer that executes the specified routine once.
- continual** A timer that calls the routine once every given time period.
- message** Either a one-shot or continual timer that sends a specified message rather than call a specified routine.
- millisecond** A one-shot timer with millisecond accuracy.



sleep A timer that puts the calling thread to sleep for a given length of time.

semaphore A timer that provides a “time-out” for a given semaphore queue (if the thread does not gain the semaphore in the given length of time, it is removed from the queue).

3.6

3.6.9 Streams

Streams are like one-way pipes that allow a thread to send data to or receive data from another thread or I/O port. Typically, streams are used when dealing with serial or parallel ports; however, they may also be used for communication between two threads.

GEOS has a Stream Driver that manages input to and output from streams. Additionally, a Parallel Driver manages output to a parallel port, and a Serial Driver manages input from and output to a serial port.

3.6.10 Math support

GEOS has a number of routines that accomplish complex mathematical calculations. There are also routines that do matrix manipulation and complicated linear algebra operations. In addition, the system provides a library of routines to implement simple fixed-point math and complicated floating-point operations.

3.6.11 International Support

International markets are extremely important to Geoworks due to the large number of PCs in use in countries other than the United States. Therefore, localization of applications was made a priority when designing the system software.

GEOS provides a number of tools for localizing your code. The most important is the GEOS application ResEdit, which allows a programmer to edit data resources. When an application is programmed correctly, translation can easily be accomplished by a support team.

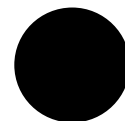
The GEOS kernel also contains a number of routines that allow applications to access different DOS character sets. These routines ensure that an English-language program will work on machines with different DOS character sets.

3.7 Libraries

3.7

The use of dynamically-linkable libraries is key to the efficiency of GEOS applications. Several libraries are included in addition to the libraries provided by the kernel (such as the Graphic Object Library and the Database Library, discussed earlier):

- ◆ **Import/Export Library**
The Impex Library (“impex”) contains a number of utilities to convert standard data files into GEOS data files and vice versa.
- ◆ **Sound Library**
GEOS provides a number of standard sounds and a small library (“sound”) to play individual notes on the PC’s internal speaker.
- ◆ **Ruler Object Library**
The Ruler library (“ruler”) provides a ruler that applications can use along with a View. It can be set to different measurement units and scales and scrolls itself along with the scale and position of the View.
- ◆ **Cell Library**
The Cell Library (“cell”) provides the lowest-level data manipulation and access routines required by a typical database or spreadsheet application.
- ◆ **Parse Library**
The Parse Library (“parse”) provides a parser for a spreadsheet language and may fill the role for applications that require a language based on mathematical expressions.
- ◆ **Spreadsheet Object Library**
The Spreadsheet Object Library (“ssheet”) provides the basic functionality of a spreadsheet application.
- ◆ **CD ROM Library**
The CD ROM Library (“cdrom”) provides all the routines necessary to interface with standard Microsoft CD ROM extensions.



3.8 Device Drivers

GEOS does as much as possible to isolate applications from the hardware in use. Not only does this allow applications to run with thousands of different system configurations, but it also allows them to work properly with any new technology not yet developed—when a new device is introduced and a driver written for it, applications will automatically work with it (in most cases).

3.8

Device drivers are used by the kernel for all of the following:

- ◆ **Video Cards**
Most video cards with graphics are supported including HGC, CGA, EGA, VGA, and Super VGA.
- ◆ **Keyboards**
International keyboards are supported for most languages using available character sets.
- ◆ **Pen Input Devices**
New input devices such as digitizers, touch screens, and other types of devices are or can be easily supported by drivers that translate the input into standard mouse-and-keyboard style input.
- ◆ **Printers**
Standard 9-pin and 24-pin dot matrix printers as well as most laser printers are supported. PostScript output is also supported.
- ◆ **Mice**
Dozens of different mice are supported.
- ◆ **Serial and Parallel Ports**
Serial and parallel ports are driven by special types of Stream Drivers and can be used to create fax and other peripheral drivers.
- ◆ **File Systems**
File systems (MS-DOS and DR DOS, for example) are supported through file system drivers. Future releases of file systems and alternatives will be supported through the addition of new file system drivers.
- ◆ **Task Switchers**
GEOS interacts with DOS task-switchers through task switch drivers.
- ◆ **Power Management Hardware**
Many portable computers (notebooks and palmtops) use power

management hardware to extend the life of their batteries. GEOS works with the power management systems through a power management system driver.

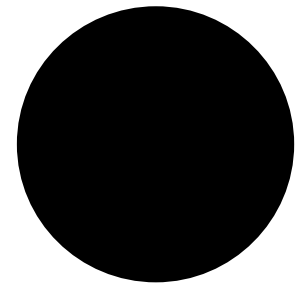
◆ **Font Rasterizers**

GEOS uses a combination of outline and bitmap fonts. Different font engines are supported through font rasterizers.

3.8

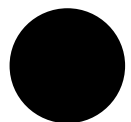


First Steps: Hello World



4

4.1	Application Structure	107
4.2	Hello World	108
4.2.1	Features of Hello World	109
4.2.2	Strategy and Internals	109
4.2.2.1	The Menu and Dialog Box	111
4.2.2.2	The Scrolling View and Drawing the Text	111
4.2.2.3	Changing the Text Color	112
4.2.3	Naming Conventions	112
4.3	Geode Parameters File	114
4.4	The Source File and Source Code	116
4.4.1	Inclusions and Global Variables	117
4.4.2	The Process Object	119
4.4.3	UI Objects	120
4.4.3.1	The Application Object	121
4.4.3.2	The Primary Window and the View Window	122
4.4.3.3	The Hello World Menu	125
4.4.3.4	The Dialog Box and Its Triggers	126
4.4.4	Code and Message Handlers	128
4.4.4.1	Handling the Window Messages	130
4.4.4.2	Handling MSG_META_EXPOSED	131
4.4.4.3	Handling Messages from the Triggers	135
4.5	Exercises and Suggestions	136





This chapter provides you with an application shell to which you can add as you learn more about GEOS programming. It describes each portion of the introductory (and compulsory) “Hello World” sample program. All programmers new to GEOS should read this chapter for a firm understanding of the basics of a GEOS application. The source code for Hello World is given in this chapter in pieces. It is also available as the Hello3 sample application.

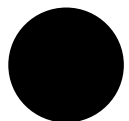
4.1

4.1 Application Structure

Every GEOS application and library will have at least two files: The Geode Parameters file provides the Glue linker with necessary information about the geode being compiled and linked. The source file contains the source code. Source code files may be split up into modules of source and definition files—this allows the programmer to keep large applications in a series of smaller files for easy organization and easier debugging.

Essentially there are four different types of files your application may use:

- ◆ **Parameters File (.gp)**
The Parameters file, as stated above, provides the Glue linker with necessary information. It details specifics about the geode being compiled that will be necessary for dynamically linking at runtime. Each geode may have only one Parameters file. The Parameters file for Hello World is detailed in section 4.3 on page 114. Full reference information for this file type can be found in the Routines reference book.
- ◆ **Source File (.goc)**
The Source file contains a combination of GEOS UI code and standard C code. Many source code files can be used for a single geode; this can help you organize your application's functionality into manageable groups of routines and objects. An introduction to the basics of the source file is given in section 4.4 on page 116.
- ◆ **C Header File (.h)**
C Header files may be used to hold definitions of data structures, classes, macros, routines, and other items. There are several standard GEOS



header files you must include in your geode for it to compile properly. These are outlined in section 4.4 on page 116.

◆ **GEOS Header File (.goh)**

This file is essentially the same as the C header files described above. It can contain class and routine definitions as well as constants and variables. The primary difference between these files and the C header files is that **.goh** files must be included before the geode is run through the Goc preprocessor. C header files do not have to go through the Goc preprocessor. Simple geodes might have none of their own header files. These files are also described in section 4.4 on page 116.

4.2

4.2 Hello World

The Hello World sample application (Hello3) is very simple and yet accomplishes a great deal. With just a few simple steps, this program does the following:

- ◆ Creates a complete primary window
- ◆ Creates a scrollable window and scroller objects
- ◆ Draws 48-point text into the scrollable window and redraws the text when the user changes a parameter
- ◆ Handles scrolling and window resizes
- ◆ Uses a pinnable menu to bring up a dialog box
- ◆ Uses a dialog box to allow the user to change colors of the text
- ◆ Uses keyboard mnemonics for both the menu and dialog box commands

The Hello World program uses some concepts and constructs with which you may not be familiar yet. This section will describe each of those constructs and will refer to other sections of the documentation for complete information.

4.2.1 Features of Hello World

To the user, the Hello World program is very simple. It consists of a primary window with a single menu and a scrollable window within the primary. In the scrollable window is a phrase of text: “Welcome to GEOS!”

The menu has a single item, “Color,” which brings up a dialog box with two buttons in it. One button is labeled “Blue” and turns the text blue, and the other is labeled “Gold” and turns the text yellow. The menu is pinnable, like all GEOS menus, and the dialog box may be moved around the screen and closed like any other independently-displayable dialog box. See Figure 4-1 for an illustration of the application on the screen.

4.2



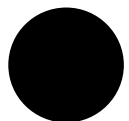
Figure 4-1 *The Hello World Sample Application*

The Hello World application draws text in a scrolling View and has one menu and one dialog box. The Blue trigger changes the text to blue; the Gold trigger changes the text to yellow.

4.2.2 Strategy and Internals

The code for Hello World, as you will see, is quite simple. It consists mainly of User Interface gadgetry and uses just a few message handlers.

The main component of the application is the Process object, an instance of **HelloProcessClass** (a subclass of **GenProcessClass**). This object makes



all drawing and color changes by handling messages sent from the window manager and the triggers in the dialog box. The Process object basically manages the application, keeping track of the relevant data and interacting with the UI gadgetry. (See Figure 4-2 for an illustration of the Hello World application's structure.)

The Process object is event-driven, meaning it has no **main()** routine run when the program is launched. Instead, the program does nothing until an event occurs (such as the view window sending MSG_META_EXPOSED when first opened). When the event (message) is received, the Process object responds to it and then waits until the next event occurs. Note, however, that

4.2

UI Gadgetry

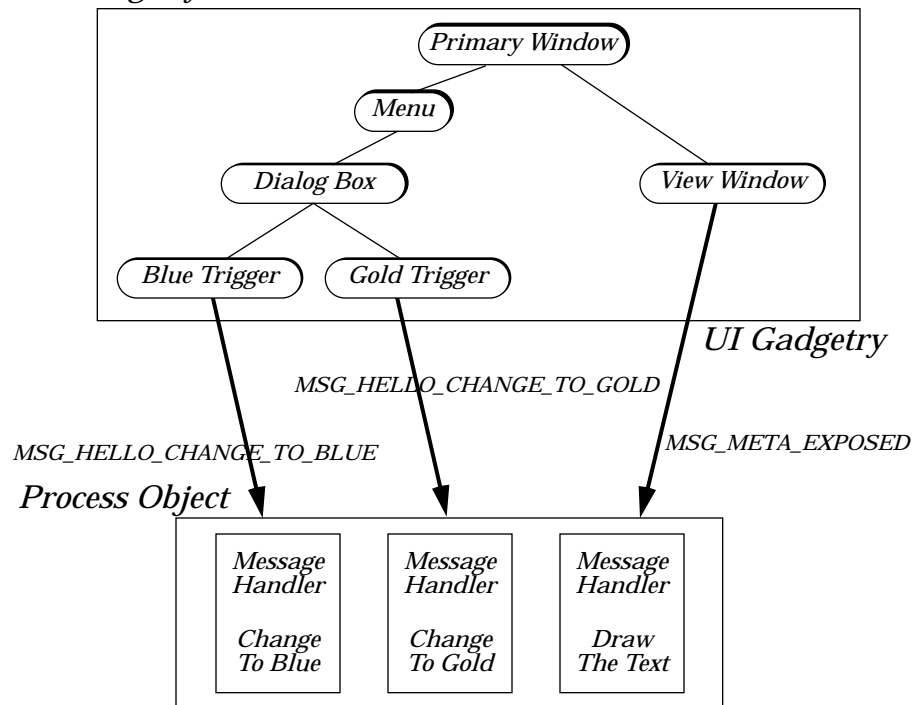


Figure 4-2 Structure of Hello World

The Process object contains message handlers (methods) for messages sent to it by the UI triggers and View. The other UI gadgets operate completely independently of the application's code; all of their functionality is implemented within the UI.

the vast majority of events a Process object will receive and handle are not generated by your code but by the kernel and the UI.

4.2.2.1 The Menu and Dialog Box

Both the menu and the dialog box, once defined in the source code as objects, are implemented automatically by the system software. The application does not have to draw or otherwise manage these objects; they will interact directly with the UI to do the proper thing.

4.2

The triggers within the dialog, however, request actions that must be carried out by the application (changing the color to blue and gold, respectively). Although the application does not have to instantiate, draw, or otherwise modify the trigger objects, it must handle messages sent out by them when they are pressed by the user. This is discussed below, under “Changing the Text Color.”

4.2.2.2 The Scrolling View and Drawing the Text

Almost everything is handled automatically by the User Interface for the Hello World application. This includes implementation of the system menus for the primary window and the scrolling functionality of the scrollable View window.

The view object (of **GenViewClass**) is powerful and provides a lot of what most applications need. It automatically handles all window resizes and scrolls, and it will cause proper redrawing when another object (such as a pinned menu) is moved across it. The only thing it does not do is actually draw the application's images.

When the view senses that some portion of its window has become invalid (through scrolling or when the view window is first opened, for example), it will send a `MSG_META_EXPOSED` message to the Hello World application's Process object. The Process object will respond by drawing the text appropriately—it does not, however, have to worry about what portion of the text is visible or what portion of the screen the view window occupies. The view will automatically clip the text properly and display it within the window's bounds.

4.2.2.3 Changing the Text Color

4.2

In all, the Process object can handle six events specific to this application: MSG_META_CONTENT_VIEW_WIN_OPENED (sent by the view when it first creates its window), MSG_META_CONTENT_VIEW_WIN_CLOSED (sent by the view when its window is being destroyed), MSG_META_EXPOSED (described above), MSG_HELLO_CHANGE_TO_BLUE (sent by the Blue trigger), MSG_HELLO_CHANGE_TO_GOLD (sent by the Gold trigger), and MSG_HELLO_REDRAW_DOCUMENT (sent by the Process object to itself).

The Process object maintains two global variables: **helloTextColor** contains the current color of the text, and **winHan** contains the window handle of the view's window. When it draws the text, the Process object checks **helloTextColor** before drawing. Therefore, the handlers for the change-color messages change the value of **helloTextColor**.

However, changing the text's color is not quite that easy. Because the view window does not have any way of knowing that the Process object has changed the text, the application must inform the UI of the change. Otherwise, the change will be made in the document but will not appear on the screen until the view window is moved or resized by the user.

Therefore, we must force a window invalidation when we change the color. This will cause the View window to generate a new MSG_META_EXPOSED that will force the redrawing of the text in the new color. The window handle is cached in **winHan** for just this purpose; when the text color changes, we must invalidate the window so the UI will redraw its contents. We invalidate the window by calling the special graphics routine **GrInvalRect()**, passing the window's handle. We get the window handle when the view first creates the window—it will send out a MSG_META_CONTENT_VIEW_WIN_OPENED. When the window closes, the view will send MSG_META_CONTENT_VIEW_WIN_CLOSED in which Hello World destroys the cached window handle.

4.2.3 Naming Conventions

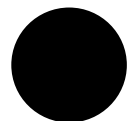
You might have noticed some of the names of variables, messages, and routines and seen a pattern of naming. Geoworks has developed a few naming conventions for various different items in the system. You don't have

to follow these conventions, of course; it may help, however, if you keep to our conventions until you're familiar with most of the system. Some of the items that have special naming conventions are

- Variables** Typically, variables begin with a lower case letter and have each subword in the name capitalized. The variable **helloTextColor** is a good example. Variables with just one word in the name are all lower case.
- Classes** Class names should always have the suffix **Class**. Typically, the first portion of a class' name will describe the particular application, library, or module it is associated with. The second portion of the name should reflect the class' superclass. Each portion of the class name should be capitalized. Thus, **HelloProcessClass** conforms because it begins with the application's name (Hello), then has the superclass' name (Process), and finally has the suffix Class. 4.2
- Constants** Constants are typically all upper case with underscores to delineate the portions of the name. **C_BLUE** is a good example; it is a set value that does not change. The structure of the name should reflect the use of the constant. (In this case, the enumerated type **Color** is reflected in the **C_** portion of the constant's name.)
- Routines** Kernel and UI routines, as well as method names, should have each portion capitalized. They should begin with some abbreviation relating to the module they belong to. For example, **GrCreateState()** is a graphics system routine, as is **GrInvalRect()**.
- Messages** Message names should be all uppercase and begin with **MSG_**. Each portion of the name should be separated with underscores, and the first portion after **MSG_** should reflect the class in which the message is defined. For example, **MSG_HELLO_CHANGE_TO_BLUE** follows all these conventions.
- Objects** Object names are typically capitalized in the same manner as routines and classes. The object's name should reflect both the object's module and its class (or its function if not its class). **HelloView** is a good example.

Instance Data

Instance data fields typically have as their first portion an



all-caps acronym for the class name with an I tacked on for “instance.” The second portion is like a variable name, and the two portions are separated by an underscore. For example, *GI_visMoniker* is a field of **GenClass** (hence the *GI_*), and *visMoniker* is the variable name of the field.

4.3

4.3 Geode Parameters File

Code Display 4-1 shows the Geode Parameters (**hello3.gp**) file for the Hello World sample application. Each of the components of the parameters file is described in detail in the reference-style entries in the Routines book.

Code Display 4-1 The Hello World Parameters File

```
#####
#
#           Copyright (c) GeoWorks 1991, 1993-- All Rights Reserved
#
# PROJECT:      GEOS V2.0
# MODULE:      Hello World (Sample GEOS application)
# FILE:        hello3.gp (Hello World Application Geode Parameters File)
#
# DESCRIPTION:  This file contains Geode definitions for the "Hello World" sample
#               application. This file is read by the Glue linker to
#               build this application.
#
#####
#
#   Permanent name: This is required by Glue to set the permanent name
#   and extension of the geode. The permanent name of a library is what
#   goes in the imported library table of a client geode (along with the
#   protocol number). It is also what Swat uses to name the patient.
#
name hello3.app
#
#   Long filename: This name can be displayed by GeoManager.
#
longname "Hello World"
```

```
#
#   Specify geode type: This geode is an application, will have its own
#   process (thread), and is not multi-launchable.
#
type    appl, process, single

#
#   Specify the class name of the application Process object: Messages
#   sent to the application's Process object will be handled by
#   HelloProcessClass, which is defined in hello3.goc.
#
class   HelloProcessClass

#
#   Specify the application object: This is the object in the
#   application's generic UI tree which serves as the top-level
#   UI object for the application. See hello3.goc.
#
appobj  HelloApp

#
#   Token: This four-letter name is used by GeoManager to locate the
#   icon for this application in the token database. The tokenid
#   number corresponds to the manufacturer ID of the program's author
#   for uniqueness of the token. Eight is Geoworks' manufacturer ID for
#   sample applications.
#
tokenchars "HELO"
tokenid 8

#
#   stack: This field designates the number of bytes to set aside for
#   the process' stack. (The type of the geode must be process, above.)
#   The default stack size is 2000 bytes.
#
stack 1500

#
#   Heapspace: This is roughly the non-discardable memory usage (in words)
#   of the application and any transient libraries that it depends on,
#   plus an additional amount for thread activity. To find the heap space
#   for an application, use the Swat "heap space" command.
#
heap space 3644
```

4.3



```
#
#      Resources: List all resource blocks which are used by the application.
#      (Standard discardable code resources do not need to be mentioned.)
#
resource APPRESOURCE ui-object
resource INTERFACE ui-object
resource MENURESOURCE ui-object

#
#      Libraries: List which libraries are used by the application.
4.4 #
library geos
library ui

#
#      User Notes: This field allows the geode to fill its usernotes field
#      (available to the user through GeoManager's File/Get Info function)
#      with meaningful text.
#
usernotes "Sample application for GEOS version 2.0."
```

4.4 The Source File and Source Code

The Hello World program's source code resides in a single file, **hello3.goc**. Portions of this file are presented throughout this section.

This sample application demonstrates the basics of all GEOS programs, and it's likely you will refer to it often until you become proficient with both the system and the reference documentation. Among the concepts and implementations it demonstrates are the following:

- ◆ Declaration and use of resources
- ◆ Standard inclusion files
- ◆ Basic UI construction
- ◆ Creation of the Primary window

- ◆ Creation of a scrolling view and drawing to the view
- ◆ Creation of a menu
- ◆ Creation of a simple dialog box
- ◆ Declaration of subclasses and messages
- ◆ Definition of message handlers
- ◆ Sending messages to another object

4.4

This discussion assumes you have a solid understanding of general programming concepts and C constructions. If you don't, you should most likely get to know the C programming language before continuing.

4.4.1 Inclusions and Global Variables

As Code Display 4-2 shows, the first thing in a **.goc** file is a list of other files and libraries that must be included. These are designated in the standard C protocol, with one exception (described below). All the inclusions and libraries a basic application will need are shown in the sample, and only the following inclusions is required of every GEOS application:

```
@include <stdapp.goh>
```

Other inclusions may be required for more complex applications and libraries, and these inclusions will be listed in the topics that require them. Goc accepts both the standard C **#include** for **.h** files and the GEOS **@include** for **.goh** files. The difference between them is that **.h** files may not include Goc constructs (e.g. **@object**) whereas **.goh** files can.

After the inclusions are listed, you should declare any global variables used throughout your application. Be aware that global variables, though accessible by any object, are owned by the application's Process object and by the application's primary thread. Objects running in other threads (such as UI objects) should not access these global variables directly because this can cause synchronization problems between threads.

Whenever possible, you should avoid using too many global variables. Global variables are typically put in a fixed-block resource, and having too many can bog down a low-memory machine.

The Hello World application uses only two global variables. The first, **helloTextColor**, holds the value of the currently-displayed text color and is initialized to the value `C_BLUE` (dark blue). The second, **winHan**, contains the window handle of the scrollable view into which we draw our text. How these variables are used will be shown later.

Code Display 4-2 Inclusion Files and Global Variables

4.4 *This is the first portion of the `hello3.goc` file.*

```
/*
 * Copyright (c) GeoWorks 1991, 1993-- All Rights Reserved
 *
 * PROJECT:      GEOS
 * MODULE:       Hello World (Sample GEOS application)
 * FILE:         hello3.goc (Code file for Hello World Sample Application)
 *
 * DESCRIPTION:
 *   This file contains the source code for the Hello World application.
 *   This code will be processed by the goc C preprocessor and then
 *   compiled by a C compiler. After compilation, it will be linked
 *   by the Glue linker to produce a runnable .geo application file.
 *
 */
/*
 * Include files
 *   These files are the standard inclusion files to use the basics of
 *   the GEOS system and libraries. All applications should include
 *   at least the following files. Note that all inclusion files
 *   have the suffix .h or .goh indicating they are header files.
 */
#include <stdapp.goh>          /* Standard GEOS inclusion file */

/*
 * Global Variables
 */
word          helloTextColor = C_BLUE;
WindowHandle  winHan;
```

4.4.2 The Process Object

Every GEOS application has an object called the Process object. This object is run by the application's primary thread and is an instance of a subclass of **GenProcessClass**. Because the Process is an event-driven object, there is no **main()** routine that is executed when the program is launched. Instead, the object will wait until it receives messages (events), at which time it will execute the proper methods.

4.4

Applications can be of two basic models: The procedural model puts the entire functionality of the application within the Process object, not using any other objects in the application's thread (UI objects are run by the UI thread). All messages sent to the application are handled by the Process object, and most OOP issues can be avoided. The object-oriented model allows for other objects to be run by the application's primary thread; each of these objects will have its own instance data and be located in an object block (a resource) associated with its own message queue.

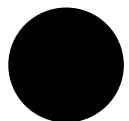
There is no command or directive that determines which model is used—the distinction is inherent within the format of message handlers and therefore can be virtually ignored. Hello World, for example, is simple enough to use the procedural model, handling all functionality with the Process object.

The Process object of Hello World is shown in Code Display 4-3.

Code Display 4-3 Hello World's Process Object

```
This code display is part of hello3.goc and follows the previous display directly. */
/*****
 *
 *          Class & Method Definitions
 * This section contains the definition of the application's Process
 * class and its methods. Other classes can also be defined here,
 * along with the message each handles.
 *****/

/*
 * Here we define "HelloProcessClass" as a subclass of the system-provided
 * "GenProcessClass". As this application is launched, an instance of this class
 * will be created automatically to handle application-related events (messages).
```



First Steps: Hello World

120

```
* The application thread will be responsible for running this object,  
* meaning that whenever this object handles a message, it will be executing  
* in the application's thread.  
*/  
  
/* You will find no object in this file declared to be of this class. Instead,  
* this class is bound to the application thread in hello3.gp.  
  
@class HelloProcessClass, GenProcessClass;  
  
4.4 /* The messages HelloProcessClass objects can handle that are not  
* system-defined are enumerated here. Each of these messages is sent  
* by one of the triggers in the dialog box. This is where class-  
* specific messages for this application (not system-defined messages)  
* are defined. */  
  
@message void MSG_HELLO_CHANGE_TO_BLUE(); /* sent by Blue trigger */  
@message void MSG_HELLO_CHANGE_TO_GOLD(); /* sent by Gold trigger */  
@message void MSG_HELLO_REDRAW_DOCUMENT(); /* sent by Process to itself */  
  
@endc /* signifies end of class definition */  
  
/* Because this class definition must be stored in memory at runtime,  
* we must declare it here along with a flag indicating how it should  
* be loaded. The "neverSaved" flag is used because Process classes  
* are never saved to state files and therefore no relocation tables  
* need be built. */  
@classdecl HelloProcessClass, neverSaved;
```

4.4.3 UI Objects

As stated earlier, the bulk of the Hello World application consists of User Interface objects. These objects are defined just after the Process object in your application's code, and they are given certain attributes and instance data. Once they have been defined, in general you will not have to bother with them again.

UI objects are organized into a hierarchy. The hierarchy for Hello World is shown in Figure 4-3, and the objects required for all applications are boxed. UI objects are also arranged into resources. Each resource is allocated a block on the global heap when the program is launched; therefore, resources should be kept to reasonable sizes whenever possible.

The application object resides within its own resource so the application takes up very little memory when iconified (minimized). Menus for complex applications are usually put in a menu resource. Most other UI gadgetry is put in a resource called “Interface” (though this name is not required).

4.4.3.1 The Application Object

Every application must have an application object, an instance of the class **GenApplicationClass**. The application object handles and manages many application-related things such as dispatching input sent by the input manager to the application. The application object must be the top-level generic object in every application. The name of the application object is also stated in the geode parameters file line **appobj** (see Code Display 4-1 on page ● 114).

4.4

Code Display 4-4 shows Hello World’s application object definition. The comments in the code are extensive and explain the purpose of each line.

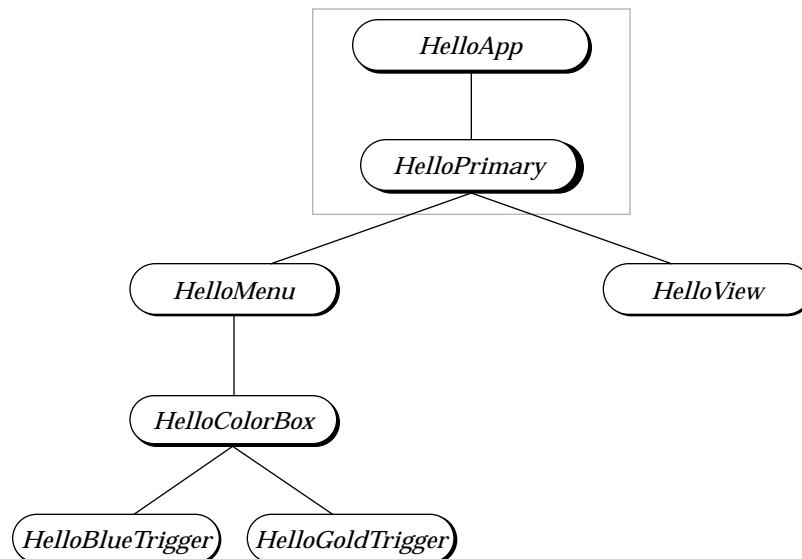


Figure 4-3 *Generic Tree of Hello World*

Seven objects make up the generic tree of the Hello World application. Those outlined by the box are required of all applications; the others are specific to Hello World.

Code Display 4-4 Hello World's Application Object

This display is part of `hello3.goc` and follows the previous display directly.

```
4.4 /*
 *                               Application Object
 * The very top-level generic object of an application MUST be a GenApplication
 * object. The hello3.gp file contains an "appobj" statement which indicates
 * that this "HelloApp" object is in fact the top-level UI object.
 * This object should be in its own resource so it takes up very little memory
 * when minimized. Note that the name of the resource may be whatever you choose;
 * it does not have to be AppResource.
 */

@start AppResource;                /* Begin definition of objects in AppResource. */

@object GenApplicationClass HelloApp = {
    GI_comp = @HelloPrimary;
    /* The GI_comp attribute lists the generic children
     * of the object. The HelloApp object has just one
     * child, the primary window of the application. */

    gcnList(MANUFACTURER_ID_GEOWORKS, GAGCNLT_WINDOWS) = @HelloPrimary;
    /* This GCN list determines which of the application's
     * window objects must be made visible when the
     * application first starts up. */
}

@end AppResource                /* End definition of objects in AppResource. */
```

4.4.3.2 The Primary Window and the View Window

Every application must have a primary window object of class **GenPrimaryClass**. This object will draw and manage the primary window and will work with the UI and the geometry manager to arrange all the children of the window properly. It will also automatically put up the system-controlled gadgets (such as the system window menu, the minimize/maximize buttons, and the Express menu).

The Hello World primary window has only two children, one of which is a menu. The other is the view object, which occupies the remaining space within the primary. The view object, as stated earlier, automatically handles all scrolling and clipping of documents. Its scrollable area is eight and a half

inches by eleven inches. A description of what the view window does and how it interacts with the Process object to draw the text is given in section 4.2.2.2 on page 111.

Code Display 4-5 shows the definition and attributes of each of these two objects. See Figure 4-4 for a diagram of the two objects implemented under the OSF/Motif specification.

Code Display 4-5 Hello World's Primary and View Objects

4.4

This display is part of hello3.goc and directly follows the previous display.

```
@start Interface;          /* This resource is for miscellaneous UI objects. */

@object GenPrimaryClass HelloPrimary = {
    GI_visMoniker = "Hello World Sample Application";
    /* This title will appear at the top of the primary
     * window as the name of the application. */

    GI_comp = @HelloView, @HelloMenu;
    /* This window has two children, the GenView object
     * and the GenInteraction menu object. */
}
```

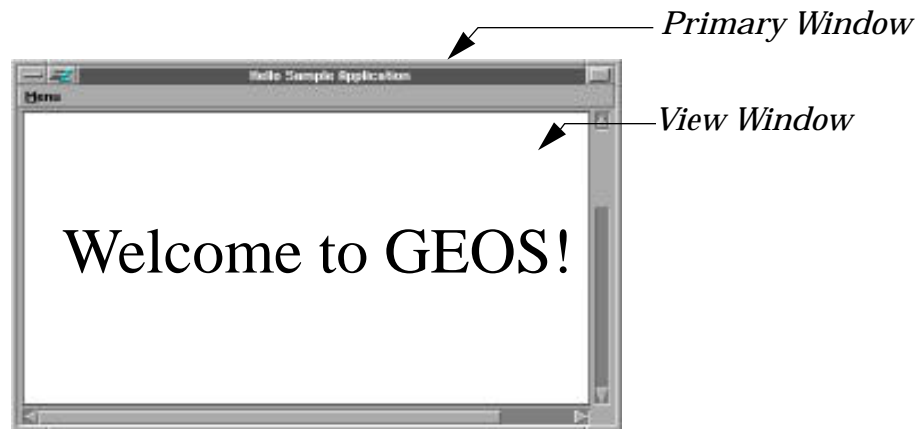


Figure 4-4 Hello Primary and View Windows

The Primary window groups other UI objects and automatically is given the system window menu, the Express menu, and the maximize button. The menu bar is also created automatically. The View object is automatically given scrollers and a window for painting by the application.



First Steps: Hello World

124

```
/* For simplicity, this application is not given an icon. Therefore, we must
 * prevent the user from being able to minimize the application. This is
 * done by applying the following attribute to the GenPrimary object. Note that
 * GenPrimaryClass is a subclass of GenDisplayClass, in which this attribute
 * is actually defined. */

ATTR_GEN_DISPLAY_NOT_MINIMIZABLE;

/* The following hint allows the primary window object to size itself
 * according to its children (the view object). */
4.4 HINT_SIZE_WINDOW_AS_DESIRED;

/*
 * When the specific UI permits, let's not show the menu bar on
 * startup. Some applications would find the extra space this leaves
 * helpful, in particular on the small screens of pen-based devices,
 * though for this simple application it doesn't really matter.
 */
ATTR_GEN_DISPLAY_MENU_BAR_POPPED_OUT;
HINT_DISPLAY_MENU_BAR_HIDDEN_ON_STARTUP;
}

/*          GenView object
 * This GenView object creates a window where the application can display portions
 * of a document as necessary. We want it to be scrollable, so the specific UI
 * (OSF/Motif) will create scroll bars which the user can interact with. Whenever a
 * portion of the window needs to be redrawn, the GenView object will invalidate
 * a portion of the window, causing a MSG_META_EXPOSED to be sent to the
 * application. The application will draw the document into the window. The
 * window keeps track of a mask which is used to clip the application's
 * drawing operations so only the invalid portion of the window is drawn. */

@object GenViewClass HelloView = {
    GVI_horizAttrs = @default | GVDA_SCROLLABLE | GVDA_NO_LARGER_THAN_CONTENT;
    /* This makes the View scrollable in the
     * horizontal dimension and keeps it from
     * growing larger than our document. */

    GVI_vertAttrs = @default | GVDA_SCROLLABLE | GVDA_NO_LARGER_THAN_CONTENT;
    /* This makes the View scrollable in the
     * vertical dimension and keeps it from
     * growing larger than our document. */

    GVI_docBounds = { 0, 0, 72*17/2, 72*11 };
    /* This sets the document size (scrollable
     * bounds) of the GenView. */
}
```



```
GVI_content = process;      /* This sets the output of the View--where it will
                             * send its MSG_META_EXPOSEDs--to be the
                             * application's Process object. */

/*
 * This view will not take text input, so specify that no floating
 * keyboard should come up. Otherwise, we would get a floating
 * keyboard by default on pen-based systems.
 */
ATTR_GEN_VIEW_DOES_NOT_ACCEPT_TEXT_INPUT;
}

@end      Interface      /* End definition of objects in this resource. */
```

4.4

4.4.3.3 The Hello World Menu

The Hello World program has one menu, called “Menu” and located in the primary window’s menu bar. Menus are instances of **GenInteractionClass** with the GIV_POPUP attribute set in the *GII_visibility* field. The moniker of the menu object appears on the menu bar (see Code Display 4-6 for the definition of Hello World’s menu).

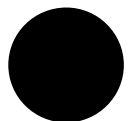
The menu should have one child for each entry in the menu. In Hello World, the only child is the dialog box, whose moniker appears as the text of the menu item that brings up the box.

Code Display 4-6 The Hello World Menu

/This display is part of hello3.goc and follows the previous display directly.

```
/*      HelloMenu Menu
 * Menus are of GenInteractionClass. The moniker of a menu is used to show the menu
 * on the primary's menu bar (thus, "Menu" will show up in Hello World's menu bar).
 * Each of the menu's children (in the GI_comp field) will be an entry or a
 * collection of entries in the menu. To separate a menu from a dialog
 * box GenInteraction, you must apply the visibility GIV_POPUP. A dialog box will
 * have the visibility GIV_DIALOG. */

@object GenInteractionClass HelloMenu = {
    GI_visMoniker = 'M', "Menu"; /* The moniker of the menu is used in
                                * the primary window's menu bar. */
```



```
GI_comp = @HelloColorBox; /* The only child of the menu (the only
                           * item in the menu) is the dialog box. */

GII_visibility = GIV_POPUP; /* This attribute designates the GenInteraction
                             * as a menu or a submenu. */
}
```

4.4

4.4.3.4 The Dialog Box and Its Triggers

Code Display 4-7 shows the code for the dialog box and its triggers.

Dialog boxes in GEOS may be of **GenInteractionClass**, or dialogs may be called up and instantiated during execution with kernel routines. Hello World uses a GenInteraction in order to have a “floating” dialog box that may be retained and moved around the screen.

When a dialog box is brought up by a menu item, the moniker of the dialog box object will be used as the text of the menu item. Thus, the word “Color” will appear both at the top of the dialog box and in the menu.

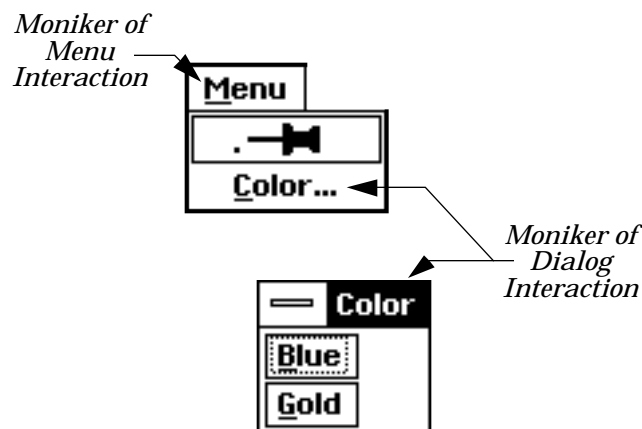


Figure 4-5 Hello World's Menu and Dialog Box

The menu consists of a *GenInteraction* object; its moniker will appear as the menu's title in the menu bar. The dialog consists of another *GenInteraction* object; its moniker will appear as both the title of the dialog box and the name of the menu item that brings the box up.

Children of a dialog box are arranged by the geometry manager. In general, children will be arranged top to bottom (or left to right, depending on the specific UI) as they are defined. Thus, the two triggers will appear “Blue” first and “Gold” second (on bottom). The dialog box will automatically size itself to fit all its children properly. Dialog boxes, unlike windows, are generally not resizable.

Each of the triggers will appear as a simple button big enough to hold its moniker. Each trigger, when pressed, sends a specified message to a specified object. The Blue trigger sends MSG_HELLO_CHANGE_TO_BLUE to the application’s Process object, and the Gold trigger sends the message MSG_HELLO_CHANGE_TO_GOLD to the Process object. These messages are placed in the Process object’s message queue and require no return values.

4.4

Code Display 4-7 The Hello World Dialog Box and Its Triggers

This display is part of hello3.goc and follows the previous display directly.

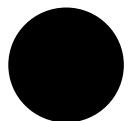
```
/*
    HelloColorBox Dialog Box
* Dialog boxes are of GenInteractionClass. The moniker of the dialog box will
* appear at the top as the box’s title. Additionally, if the GenInteraction
* is a child of a menu interaction, the moniker will also serve as the text that
* appears in the menu item that brings up the dialog box. The dialog will
* automatically size itself to be large enough to hold all its children. Thus,
* this dialog will be small (with just two children, each a trigger with a short
* moniker). */

@object GenInteractionClass HelloColorBox = {
    GI_visMoniker = 'C', "Color";/* The moniker will be displayed both as the
    * dialog’s title and as the menu item that
    * brings the dialog up. */

    GI_comp = @HelloBlueTrigger, @HelloGoldTrigger;
    /* The two triggers are the only gadgets
    * in the dialog box. */

    GII_visibility = GIV_DIALOG; /* In order for this interaction to be
    * a dialog box, this attribute must be set. */
}

/*
    GenTriggers
* Buttons are implemented by GenTriggerClass. When a trigger is pushed by the user
* (clicked on with the mouse), it will send the specified message to the specified
* output object. In both cases below, the trigger will send an application-defined
```



```
* message to the application's Process object. A trigger's moniker is displayed
* within the trigger. In this case, both are text, but any graphics could be used
* instead to create graphical triggers easily. (E.g. a blue flag and a gold flag
* rather than the words "Blue" and "Gold.") */

@object GenTriggerClass HelloBlueTrigger = {
    GI_visMoniker = 'B', "Blue"; /* The 'B' indicates the keyboard navigation
                                * character for this trigger. */
    GTI_destination = process; /* Send the message to the Process object. */
    GTI_actionMsg = MSG_HELLO_CHANGE_TO_BLUE; /* Send this message. */
}

4.4

@object GenTriggerClass HelloGoldTrigger = {
    GI_visMoniker = 'G', "Gold"; /* The 'G' indicates the keyboard navigation
                                * character for this trigger. */
    GTI_destination = process; /* Send the message to the Process object. */
    GTI_actionMsg = MSG_HELLO_CHANGE_TO_GOLD; /* Send this message. */
}
```

4.4.4 Code and Message Handlers

One of the first things a C programmer might notice when looking at the Hello World program is that it has no **main()** routine. This illustrates the primary distinction between an object-oriented system and a system that emulates object-orientedness.

Applications in GEOS consist of a Process object and, optionally, other objects in the same or different threads. The Process object of an application is *event-driven*. This means that until it receives a message, it does nothing; when it receives a message, however, it will automatically be woken up with the instruction pointer pointing at the proper routine's entry point.

GenProcessClass, the superclass of every Process object, handles many messages that most applications may never need to know about. For example, when the program is first launched, the Process object will receive a series of messages from the UI and the kernel telling it how it should start up. It automatically responds by setting up the proper message queues and executing the proper code. These are things you, as the programmer, do not need to know about to create a working GEOS application (though they are documented in "Applications and Geodes," Chapter 6).

Throughout the program's life, then, the Process object will receive and respond to messages as they are received. Each message has at most one corresponding method; if no method exists for a message, the message is ignored.

The Hello World Process object can handle six different messages, each of which is sent by a UI object. It also uses one routine defined internally and not available for use by other objects. Of the six messages it handles, three are specific to Hello World and three are universal to all applications using a GenView object: MSG_META_EXPOSED, MSG_META_CONTENT_VIEW_WIN_OPENED, and MSG_META_CONTENT_VIEW_WIN_CLOSED are sent by the view, and MSG_HELLO_CHANGE_TO_BLUE, MSG_HELLO_CHANGE_TO_GOLD, and MSG_HELLO_REDRAW_DOCUMENT are defined specific to **HelloProcessClass** and are sent by the triggers.

4.4

The function **HelloDrawText()** is internal to Hello World and is called by the MSG_META_EXPOSED handler. It is declared before the handler to satisfy normal C constraints.

Additionally, two constants are defined to determine the document size. These constants, along with the declaration of **HelloDrawText()**, are shown in Code Display 4-8.

Code Display 4-8 Constant and Routine Definition

This display is part of hello3.goc and follows the previous display directly.

```

/*****
 *
 * Code for HelloProcessClass
 * Now that all the UI gadgetry has been defined, we must provide the
 * methods and routines used by the application. For simplicity, all
 * messages will be handled by the HelloProcessClass object.
 *****/

/* Define constants used by the color-setting methods. Each of these
 * is a document size parameter in points. Therefore, the document is
 * 8.5 inches wide by 11 inches tall (one point is 1/72 of an inch). */
#define HORIZ_DOC_SIZE (72*17/2)
#define VERT_DOC_SIZE (72*11)

```



First Steps: Hello World

130

```
/* Declare that we will use the function HelloDrawText(), and define its
 * return and parameter values. It has no return value and has one parameter:
 * a graphics state handle called "gstate." */
void HelloDrawText(GStateHandle gstate);

/* The following constants are used by HelloDrawText(). */

#define TEXT_POINT_SIZE 48      /* point size */
#define TEXT_X_POSITION 30     /* x position, in document coords. */
#define TEXT_Y_POSITION 100    /* y position, in document coords. */
```

4.4

4.4.4.1 Handling the Window Messages

As stated earlier, the **winHan** global variable contains the window handle of the view's window. To set the variable, Hello World must intercept and handle the message `MSG_META_CONTENT_VIEW_WIN_OPENED`. This message passes the window handle, which **HelloProcessClass** simply stuffs into its **winHan** variable.

When the view window is destroyed, the application must make sure it forgets its window handle. Otherwise, we could try to draw to a nonexistent window, which is an error. This will not be a problem for Hello World because the only time the view can be destroyed is when the application is being shut down. For completeness, however, Hello World handles `MSG_META_CONTENT_VIEW_WIN_CLOSED` and sets **winHan** to zero.

Both of the methods for the above messages are shown in Code Display 4-9.

Code Display 4-9 Messages from the View

This display is part of `hello3.goc` and follows the previous display directly.

```
/* NOTE:
 * Because these are simple methods, the structure and syntax of methods are not
 * handled here. See the handler for MSG_META_EXPOSED, later in this chapter. */

/*****
 * MSG_META_CONTENT_VIEW_WIN_OPENED for HelloProcessClass
 *****/
* SYNOPSIS:      Record the handle of the view window when the view
*                  creates it. This allows us to more-easily update the
*                  document when the user changes the text color.
```



```

* PARAMETERS: void ( word viewWidth,
*                  word viewHeight,
*                  WindowHandle viewWin)
* SIDE EFFECTS: winHan is set to viewWindow
*****/

@method HelloProcessClass, MSG_META_CONTENT_VIEW_WIN_OPENED {
    /* Get the window handle of the View. We need this handle in order to
    * force a window invalidation, causing the View to send a MSG_META_EXPOSED
    * to the Process object and thereby forcing a redraw of the window. */
    winHan = viewWindow;
}

/*****
* MSG_META_CONTENT_VIEW_WIN_CLOSED for HelloProcessClass
*****
* SYNOPSIS:      Take note that the view is now closed, so we don't
*                  try and draw to it or invalidate it any more (at
*                  least until it reopens)
* PARAMETERS:    void (WindowHandle viewWindow)
* SIDE EFFECTS:  winHan is set to 0
*
*****/

@method HelloProcessClass, MSG_META_CONTENT_VIEW_WIN_CLOSED {
    /* Set our window handle variable to zero. */
    winHan = 0;
}

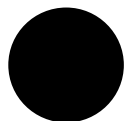
```

4.4

4.4.4.2 Handling MSG_META_EXPOSED

As discussed in our `HelloView` object declaration (see Code Display 4-5), the view will send a `MSG_META_EXPOSED` to the Hello World Process object. Receipt of this message indicates that part of the scrollable view window has become invalid and must be redrawn.

Therefore, the class of the Hello World Process object (**HelloProcessClass**) must know how to draw the document in response to this message. Note that this message did not have to be defined specifically in the earlier definition of **HelloProcessClass**—this is because the message is already defined for **MetaClass**, the superclass of all GEOS classes.



Code Display 4-10 shows the method that handles MSG_META_EXPOSED for **HelloProcessClass**. Notice that the method calls the **HelloDrawText()** routine rather than drawing the text directly. While this may appear inefficient at first (and is for such a small, simple application), there are two main reasons why this is done:

First, it takes advantage of the GEOS single imaging model. The method simply creates the proper **GState** for drawing to the view window, then calls the drawing routine. A similar message for printing (i.e. when the user clicks on a “Print” trigger, a print message may be sent to the Process object) can use the same drawing routine—its handler would simply set up a **GState** for drawing to a Spool file and then call the drawing routine. Thus, one function is used for two purposes, cutting down code size.

Second, it allows more modularity in the testing of your code. If you need to make sure, for example, that the message is being received and handled, but you don’t (yet) care if the drawing is done properly, you can set up **HelloDrawText()** as a dummy function. This would allow you to ensure the message is handled properly without having to debug all the drawing code.

Code Display 4-10 MSG_META_EXPOSED Handler

This display is part of hello3.goc and follows the previous display directly.

```
/* *****  
 * MSG_META_EXPOSED for HelloProcessClass  
 * *****  
 * SYNOPSIS:      Redraw the recently-exposed portion of the View  
 * PARAMETERS:    void (WindowHandle win)  
 * SIDE EFFECTS:  The invalid region of the window is cleared out  
 *  
 * STRATEGY:      This message is sent by the windowing system when a  
 *                  portion of the GenView has become invalid, either  
 *                  because a window that was obscuring it has been moved,  
 *                  or because some called GrInvalRect.  
 *  
 *                  We redraw the entire document, after telling the  
 *                  graphics system we’re drawing to the invalid portion  
 *                  of the window.  
 *  
 * ***** */
```

```

/* The method is declared with the goc keyword @method. This is followed by
 * the name of the class that knows how to handle the message (in this case,
 * the class is HelloProcessClass). Finally, the name of the message that
 * invokes this method is specified. Other items may also be specified (such
 * as a routine name that can be used instead of a message), but these are not
 * required. */
@method HelloProcessClass, MSG_META_EXPOSED {

    /* The local variable gstate will hold a GState handle. We will do
     * our drawing to this gstate. */
    GStateHandle gstate;

    /* Get a new, default graphics state that we can use while drawing.
     * We must allocate a new graphics state for the View window using
     * the kernel routine GrCreateState(). We pass the window handle of
     * the View window, which we received in a parameter called "win". */
    gstate = GrCreateState(win);

    /* Next, start a window update. This tells the windowing system that
     * we are in the process of drawing to this window. This is very
     * important—it ensures the window will be in a consistent state while
     * we're drawing. Specifically, it locks in the invalidated region to
     * which we'll be drawing; this makes sure that other threads drawing
     * to this window will not have any effect on our GState. A window
     * update is started by calling the kernel routine GrBeginUpdate()
     * with the GState handle. */
    GrBeginUpdate(gstate);

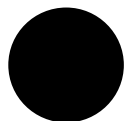
    /* If we had background graphics to draw, we could call the appropriate
     * graphics routines now. But we don't. */

    /* Draw our text into the window (pass the GState). This is done here
     * by calling the function HelloDrawText(), which knows how to draw
     * the appropriate document. (See below.) */
    HelloDrawText(gstate);          /* Special Hello World routine (below). */

    /* Now end the window update (unlock the GState and its window)
     * with the routine GrEndUpdate(), and free the GState handle by calling
     * the kernel routine GrDestroyState(). */
    GrEndUpdate(gstate);           /* Signal that we are done with
                                   * the window update. */
    GrDestroyState(gstate);        /* Destroy the temporary GState. */
}

```

4.4



First Steps: Hello World

134

4.4

```
/* *****
 * MSG_HELLO_REDRAW_DOCUMENT for HelloProcessClass
 * *****
 * SYNOPSIS:      Force the document to be redrawn by marking the
 *                entire document area in the view as invalid.
 * PARAMETERS:    void (void)
 * SIDE EFFECTS:  Any drawing to the document area between this
 *                message and the MSG_META_EXPOSED that it generates
 *                will not show up, as the entire window will be invalid.
 *
 * *****
 * *****/

@method HelloProcessClass, MSG_HELLO_REDRAW_DOCUMENT
{
    /* Now create a temporary GState to use for window invalidation
     * if the window handle is valid, then redraw the window. */
    if (winHan != 0) {
        GStateHandle gstate = GrCreateState(winHan);
        /* Call GrInvalRect using the GState. Invalidate the entire
         * document. This will cause the View to redraw itself and send
         * a MSG_META_EXPOSED to the Process object. */
        GrInvalRect(gstate, 0, 0, HORIZ_DOC_SIZE, VERT_DOC_SIZE);
        GrDestroyState(gstate);          /* Free the GState. */
    }
}

/* *****
 * HelloDrawText
 * *****
 * SYNOPSIS:      Draw a single line of text onto the document. Note
 *                that it has no concept of the screen or the view --
 *                it is given a graphics state and draws through it.
 * CALLED BY:     (INTERNAL) HelloProcess::MSG_META_EXPOSED
 * RETURN:        nothing
 * SIDE EFFECTS:  attributes in the gstate are altered
 *
 * STRATEGY:      We separate the drawing from the exposure handler
 *                so the same routine can be used for both window
 *                refresh and for printing.
 *
 *                Set the font, point size and color, then draw the text.
 *
 * *****
 * *****/
```



```

/* Functions are declared as they would be in C. Parameters are defined using
 * the ANSI calling convention: The type of the parameter is given, followed
 * by the parameter name. Multiple parameters are separated by commas. This
 * function has a single parameter. */

void HelloDrawText(GStateHandle gstate) {

    /* First change some of the default GState values such as the font
     * and point size. This is done with the routine GrSetFont(). */
    GrSetFont(gstate, FID_DTC_URW_ROMAN, MakeWWFixed(TEXT_POINT_SIZE));

    /* Set the text color to the value in helloTextColor by calling the
     * graphics routine GrSetTextColors(). */
    GrSetTextColors(gstate, CF_INDEX, helloTextColor, 0, 0);

    /* Draw the text onto the document by using the GrDrawText() routine. */
    GrDrawText(gstate, TEXT_X_POSITION, TEXT_Y_POSITION,
               "Welcome to GEOS!", 0);
}

```

4.4

4.4.4.3 Handling Messages from the Triggers

When the user clicks on one of the two triggers in the Color dialog box, the pressed trigger sends off a message to the Hello World Process object. The Blue trigger sends MSG_HELLO_CHANGE_TO_BLUE, and the Gold trigger sends MSG_HELLO_CHANGE_TO_GOLD. The Process object must be able to handle both of these messages.

The methods that handle these messages are similar. Each sets the global variable **helloTextColor**, and each forces the view window to redraw itself (by sending MSG_HELLO_REDRAW_DOCUMENT to the Process) so the color is changed on the screen as well as in our variable. Code Display 4-11 shows the code and comments of both these methods.

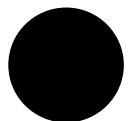
Code Display 4-11 Handlers for MSG_HELLO_...

This display is part of hello3.goc and follows the previous display directly.

```

/*****
 * MSG_HELLO_CHANGE_TO_BLUE for HelloProcessClass
 *****/
 * SYNOPSIS:      Set the text color to dark blue and redraw the text.

```



```
* PARAMETERS: void (void)
* SIDE EFFECTS:helloTextColor is set to C_BLUE
*
*****/

@method HelloProcessClass, MSG_HELLO_CHANGE_TO_BLUE {
    helloTextColor = C_BLUE; /* Set the helloTextColor variable to blue. */
    @call self::MSG_HELLO_REDRAW_DOCUMENT();
}

4.5 /*****
* MSG_HELLO_CHANGE_TO_GOLD for HelloProcessClass
*****
* SYNOPSIS: Set the text color to yellow and redraw the text.
* PARAMETERS: void (void)
* SIDE EFFECTS:helloTextColor is set to C_YELLOW
*
*****/

@method HelloProcessClass, MSG_HELLO_CHANGE_TO_GOLD
{
    helloTextColor = C_YELLOW; /* Set the helloTextColor variable to gold. */
    @call self::MSG_HELLO_REDRAW_DOCUMENT();
}
```

4.5 Exercises and Suggestions

After studying the Hello World sample application and reading the System Architecture chapter, you may be ready to try some exercises before embarking on your application. It is strongly suggested that you spend some time modifying the Hello World program to get used to the programming environment and the Goc UI definition syntax. Some things you might try are outlined below—some are extremely easy; some are more difficult. Some will require you to read other chapters in this and the User Interface manual. In any case, you should go on to read “GEOS Programming,” Chapter 5, to understand the various keywords and constructs of the Goc programming language.

Some simple UI exercises you can incorporate into the Hello World program include the following suggestions:

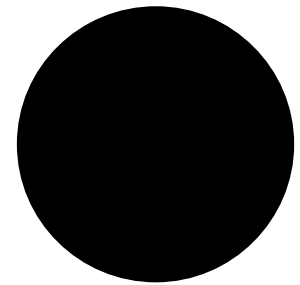


- ◆ Add a new menu item that brings up a dialog box letting the user select a point size for the text. In the dialog box you would use a `GenValue` object or perhaps a list of point sizes (or even one or more triggers) to take the user's input. The `GenValue` sends out a message with a new number when the user sets it, and your `Process` object would handle this message and set the point size. Your **`HelloDrawText()`** routine would then have to set the point size of the text as well as the color.
- ◆ Add a feature that lets the user set the position and/or rotation of the text. This exercise will help you get used to the GEOS coordinate space. 4.5
- ◆ Make `HelloView` scalable. To do this, you will create a new menu or dialog box that lets the user specify the scale. Most likely you will create a number of `GenTriggers` that apply a pre-specified scale; each of these triggers will send `MSG_GEN_VIEW_SET_SCALE_FACTOR` to `HelloView`. You may even try using a **`GenViewControlClass`** object.
- ◆ Add background graphics to get used to the coordinate system. Perhaps draw a colored rectangle behind the text that changes colors to be the inverse of the text (i.e. when the text turns blue, the rectangle turns yellow).
- ◆ Add printing. This is not nearly as difficult as you might think. To do this, you should read "The Spool Library," Chapter 17 of the *Object Reference Book*, about the spooler and the spool control object. This exercise essentially consists of adding a trigger to invoke printing and a message handler for the printing message. This handler will create a printing `GState` and call the drawing routine. Instead of drawing to the screen, it draws to the Spooler's input.

Some of the exercises may have been implemented in the sample programs provided. You should look through the samples for insight into how the exercises may be completed. You are also encouraged to make up your own exercises to build on `Hello World`.

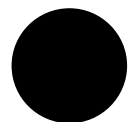
If you are new to GEOS programming, you should work through the Tutorial; otherwise, you will likely be overwhelmed by the complexity and depth of the system and not know where to begin.

GEOS Programming



5

5.1	Basic Data Types and Structures.....	141
5.1.1	Records and Enumerated Types.....	142
5.1.2	Handles and Pointers	144
5.1.2.1	Handles.....	144
5.1.2.2	Chunk Handles and Object Pointers	146
5.1.2.3	Pointers	146
5.1.3	Fixed Point Structures	147
5.2	Goc and C.....	149
5.2.1	Goc File Types	149
5.2.2	Conditional Code in Goc	151
5.2.3	Macros in Goc	151
5.2.4	Using Routine Pointers in Goc.....	152
5.3	The GEOS Object System	154
5.3.1	GEOS Terminology.....	154
5.3.1.1	General Terms.....	154
5.3.1.2	Class Terms	155
5.3.1.3	Object Terms	156
5.3.1.4	Messaging Terms	157
5.3.2	Object Structures	158
5.3.2.1	Instance Chunk Structures	159
5.3.2.2	Master Classes	163
5.3.2.3	Class Structure and Class Trees.....	164
5.3.2.4	Variant Classes	171
5.3.2.5	An In-Depth Example.....	175
5.3.3	The GEOS Message System.....	180
5.3.3.1	The Messaging Process.....	180
5.3.3.2	Message Structures and Conventions	182
5.4	Using Classes and Objects.....	182
5.4.1	Defining a New Class or Subclass.....	184
5.4.1.1	Defining New Messages for a Class	187



5.4.1.2	Defining Instance Data Fields	192
5.4.1.3	New Defaults for Subclassed Instance Data Fields	194
5.4.1.4	Defining and Working With Variable Data Fields	195
5.4.1.5	Defining Relocatable Data.....	202
5.4.2	Non-relocatable Data.....	203
5.4.3	Defining Methods.....	204
5.4.4	Declaring Objects.....	209
5.4.4.1	Defining Library Code	209
5.4.4.2	Declaring Segment Resources and Chunks.....	210
5.4.4.3	Declaring an Object	212
5.4.5	Sending Messages.....	219
5.4.6	Managing Objects	226
5.4.6.1	Creating New Objects.....	227
5.4.6.2	Working With Object Blocks	231
5.4.6.3	Working With Individual Objects	232
5.4.6.4	Managing Object Trees.....	232
5.4.6.5	Detaching and Destroying Objects	234
5.4.6.6	Saving Object State	237



Because GEOS implements its own messaging and object system, standard C programming must be supplemented with GEOS-specific programming. This chapter describes the syntax and commands available to GEOS programmers in the Goc preprocessor.



This chapter will not endeavor to teach C or object-oriented programming concepts—you should be familiar with both before continuing. Additionally, you should have read both “System Architecture,” Chapter 3 and “First Steps: Hello World,” Chapter 4.

5.1

5.1 Basic Data Types and Structures

In addition to the standard data types available in C, the Goc preprocessor handles several other types specific to GEOS. These are all defined in the file **geos.h**. Some of these types were carried over from the world of assembly language and (along with the standard C types) are shown in Table 5-1.

Table 5-1 *Basic Data Types*

Type Name	Description
byte	An unsigned, 8-bit field.
sbyte	A signed, 8-bit field (same as char).
word	An unsigned, 16-bit field.
wchar	An unsigned, 16-bit field (same as word).
sword	A signed, 16-bit field (same as short).
dword	An unsigned, 32-bit field.
sdword	A signed, 32-bit field (same as long).
Boolean	A type (16 bits) used for Boolean functions.

Additional basic types supported by GEOS. The standard types defined in the C programming language are also supported.

The Boolean type behaves as most Boolean types—any nonzero value represents a *true* state, and zero represents the *false* state. Throughout the documentation, *true* and *false* are taken to be these meanings. Note that the constants TRUE and FALSE are defined and may be used as return values from your functions and methods. Do not compare Boolean variables, however, against these constants. A Boolean may be *true* without actually equaling the TRUE value.

5.1

5.1.1 Records and Enumerated Types

GEOS objects and routines make extensive use of flag records and enumerated types. A flag record is a byte, word, or dword in which each bit represents the state (on or off) of a particular attribute or function. An enumerated type is a byte or word in which each enumeration has a unique constant value.

There are three basic types of flag records, shown in Table 5-2. To define a flag record, you should use one of these types and then define the flags to be bits within the record. To set flags, OR them (bitwise OR) with the record; to clear them, AND their bitwise inverses (bitwise AND) with the record. Creating and working with flag records is shown in Code Display 5-1.

There are two basic enumerated types: The standard enumerated type supported by your C compiler uses word-sized values. GEOS also allows byte-sized enumerated types with the **ByteEnum** type. Use of this type is shown in Code Display 5-1.

Table 5-2 *Flag Records and ByteEnum*

Type Name	Description
ByteFlags	An 8-bit record of bit flags.
WordFlags	A 16-bit record of bit flags.
DWordFlags	A 32-bit record of bit flags.
ByteEnum	An 8-bit enumerated type to complement the 16-bit type supported by standard C compilers.

Types of GEOS flags records and ByteEnum.

Code Display 5-1 Flag Records and ByteEnums

```
/* Define flag records to be the optimized length for the number of flags. For
 * example, the sample type MyFlag has six flags and therefore should be a byte.
 * Flag values should be constants equivalent to having a single bit set in the
 * flag record. */

typedef ByteFlags MyFlag;
#define MF_FIRST_FLAG    0x01
#define MF_SECOND_FLAG   0x02
#define MF_THIRD_FLAG    0x04
#define MF_FOURTH_FLAG   0x08
#define MF_FIFTH_FLAG    0x10
#define MF_SIXTH_FLAG    0x20

/* In a section of code, to set a flag, bitwise OR it with the record. To clear the
 * flag, bitwise AND its inverse with the record. You can set any number of flags
 * at a time as shown in the following examples. */

...
MyFlag      myFlagsRecord; /* Set up a variable of the flag record type */

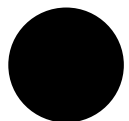
/* Set the second and fourth flag. */
myFlagsRecord = MF_SECOND_FLAG | MF_FOURTH_FLAG;

/* Set the first flag and then clear the fifth and sixth flags. */
myFlagsRecord = (myFlagsRecord | MF_FIRST_FLAG) & ~(MF_FIFTH_FLAG |
                                                    MF_SIXTH_FLAG);

/* The ByteEnum type can be used instead of the standard enumerated type, which
 * is implemented by most compilers as a word type. To define a ByteEnum, define
 * the type and then a unique constant value for each enumeration as in the
 * following example. */

typedef ByteEnum USCity;
#define USC_HARTFORD      0x00
#define USC_CHARLOTTE     0x01
#define USC_WICHITA       0x02
#define USC_PIERRE        0x03
#define USC_ORLANDO       0x04
```

5.1



5.1.2 Handles and Pointers

Handles and pointers are present everywhere in GEOS—they are the essential elements that make dynamic linking and efficient memory management possible.

5.1

GEOS pointers are all far pointers; that is, they are 32-bit addresses that reference specific locations in memory. They are normal C pointers and can be used as such. Two other pointer types are also used by GEOS: Object pointers (optrs) and segment pointers. Object pointers are described below; segment pointers are 16-bit addresses described in “Memory Management,” Chapter 15.

5.1.2.1 Handles

Handles are 16-bit, unsigned values used for several purposes. They provide abstraction when the exact address of a data structure or other item is not known or is an inconsistent state. The kernel maintains a handle table to keep track of many of the handles in the system. Each entry in the handle table is 16 bytes that contains information about the item referenced by the handle; these 16 bytes are opaque to applications and libraries and can not be accessed or altered except by the kernel. Other handle types are managed in other tables by the kernel.

Handles are used for the following primary purposes. For a full description of how handles are used, see “Handles,” Chapter 14.

- ◆ **Memory reference**
Entries for memory handles contain pointers to memory blocks; when the blocks move, the pointers are updated. However, the handle’s integrity is preserved, and applications keep track of just the handle value.
- ◆ **Virtual Memory reference**
Entries for VM handles indirectly reference VM blocks similar to the way memory handles reference blocks of memory. VM handles may be used whether the VM block is resident in memory or not.
- ◆ **File and disk reference**
Entries for file and disk handles contain information about the location and status of the referenced file/disk. They provide indirect access to files and disks in a manner similar to memory handles.

- ◆ **Data structure implementation**
Certain frequently-used system data structures require a storage format that provides for quick and convenient access at all times. These data structures (for example, threads, event queues, messages, timers, and semaphores) are stored in Handle Table entries.
- ◆ **Optimization**
The kernel will, if space permits, sometimes use Handle Table entries for optimized temporary storage. (For example, when passing parameters on the stack with messages, the kernel will occasionally use handles for storing the parameters.)

5.1

The **NullHandle** value (zero) is used to indicate a null handle.

There are over a dozen different types of handles that can be used by any sort of geode. These are listed in Table 5-3, along with a brief description of each. All are 16-bit unsigned integers.

Table 5-3 *Handle Types*

Type Name	Description
Handle	All-purpose handle.
MemHandle	References a block of memory.
DiskHandle	References a particular disk.
FileHandle	References a particular file.
ThreadHandle	References a thread.
QueueHandle	References an event queue structure.
TimerHandle	References a timer data structure.
GeodeHandle	References a geode.
GStateHandle	References a graphic state.
WindowHandle	References a window.
SemaphoreHandle	References a semaphore data structure.
EventHandle	References a particular event in an event queue.
ThreadLockHandle	References a thread lock data structure.
VMFileHandle	References a VM file.
VMBlockHandle	References a VM block (with a VM file handle).
NullHandle	The null value of any handle.

The internal structure of every handle type is opaque and can not be accessed except by the kernel. Swat, the GEOS debugger, provides commands that allow you to access the data referenced by the various handles.



5.1.2.2 Chunk Handles and Object Pointers

5.1

Objects and small data structures are stored in small memory pieces called chunks. Chunks are stored in memory blocks known as local memory heaps, and each local memory heap can contain several chunks. Each chunk is referenced by a combination of two handles: The MemHandle handle locates the local memory heap, and the ChunkHandle locates the chunk within the block. (The ChunkHandle type is not shown in Table 5-3 because it is not a normal handle and must be used with a MemHandle.) A null chunk handle value is specified by **NullChunk**.

Objects are referenced in the same way as chunks, but the handle and chunk handle are combined into a single structure called an Object Pointer, or optr. Each optr uniquely identifies a particular object in the system. Note that optrs are often used to reference non-object chunks and data structures. A null value is specified by **NullOptr**.

GEOS provides several macros, all defined in **geos.h**, for creating and parsing optrs.

◆ **ConstructOptr()**

This macro constructs an optr from a MemHandle and a ChunkHandle.

◆ **OptrToHandle()**

This macro extracts the MemHandle portion of the given optr.

◆ **OptrToChunk()**

This macro extracts the chunk handle portion of a given optr.

5.1.2.3 Pointers

Pointers can be used normally as in C. All Goc-generated pointers are far pointers; that is, they are 32-bits long, composed of a 16-bit segment and a 16-bit offset.

GEOS provides macros for extracting the segment and handle portions of pointers.

◆ **PtrToSegment()**

This macro returns the segment portion of the given pointer (returned as type "word").

◆ PtrToOffset()

This macro returns the offset portion of the given pointer (returned as type “word”).

GEOS automatically loads code resources when needed. However, when you call routines through pointers, you must take special measures to see to it that the routine is properly loaded into memory. This is discussed below in section 5.2.4 on page 152.

5.1

5.1.3 Fixed Point Structures

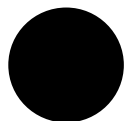
When you want to represent non-integral numbers (i.e., real numbers), you can use either the standard C floating-point format or the following special structures of GEOS for fixed point math. Note that fixed-point calculations are faster than the corresponding floating-point math, so if you want to optimize your code, you should use the GEOS fixed-point structures with the GEOS math routines.

Code Display 5-2 GEOS Data Structures

```
/* Fixed-Point Structures
 * The following structures are used to represent fixed-point numbers:
 * numbers with a fractional portion and an integral portion. Notice that
 * there are several formats of fixed-point numbers; each uses a different
 * number of bits for the parts of the number. Choose whichever is most
 * appropriate (for optimization). */

/* BBFixed
 * One byte integer, one byte fraction */
typedef struct {
    byte    BBF_frac;    /* fractional portion */
    byte    BBF_int;     /* integral portion */
} BBFixed;

/* BBFixedAsWord
 * Sometimes it is convenient to refer to a BBFixed value as type word.
 * The BBFixedAsWord type is used for this purpose. */
typedef word    BBFixedAsWord;
```



```
/* WBFixed
 * One word integer, one byte fraction */
typedef struct {
    byte    WBF_frac;    /* fractional portion */
    word    WBF_int;     /* integral portion */
} WBFixed;

/* WWFixed
 * One word integer, one word fraction */
5.1 typedef struct {
    word    WWF_frac;    /* fractional portion */
    word    WWF_int;     /* integral portion */
} WWFixed;

/* WWFixedAsDWord
 * Sometimes it is convenient to refer to a WWFixed value as type dword.
 * The WWFixedAsDWord type is used for this purpose. */
typedef dword    WWFixedAsDWord;

/* DWFFixed
 * two words (one dword) integer, one word fraction */
typedef struct {
    word    DWF_frac;    /* fractional portion */
    sdword  DWF_int;     /* integral portion */
} DWFFixed;

/* Three-byte structure
 * The WordAndAHalf structure is used when you need a 24-bit value and you want to
 * optimize and avoid using a 32-bit value. */
typedef struct {
    word    WAAH_low;    /* the low 16 bits */
    byte    WAAH_high;   /* the high 8 bits */
} WordAndAHalf;
```

Three special macros are also available to work with the **WWFixed** type. These are listed below:

◆ **MakeWWFixed**

This macro creates a **WWFixed** structure from a given floating-point number or dword number.

◆ **WWFixedToFrac**

This macro produces the fractional portion of a **WWFixed** structure.

◆ **WWFixedToInt**

This macro produces the integral portion of a **WWFixed** structure.

Two other macros are provided for use with **WWFixedAsDword** structures:

◆ **IntegerOf()**

This macro returns the integral portion of a **WWFixedAsDword** structure.

◆ **FractionOf()**

This macro returns the fractional portion of a **WWFixedAsDword** structure.

5.2

5.2 Goc and C

Goc is a superset of the standard C programming language. Goc actually acts as a sort of preprocessor before the code is run through a standard C compiler. There are several differences you must be aware of, though. These differences are covered in the following sections as well as throughout the documentation.

5.2.1 Goc File Types

`@include`, `@optimize`

When programming in Goc, you will use several different types of code files. Files ending in **.goh** are the Goc equivalent of C **.h** files—they contain routine headers, constants, and other included data and structures necessary for the program. Files ending in **.goc** are the Goc equivalent of C **.c** files—they contain code and data for the program. You should keep any Goc-specific code or header information strictly in the Goc files, and standard C code should be kept in C files. C code can also be put in **.goc** and **.goh** files, but for consistency, you should try to keep it separate.

Not all of your program's header files need be **.goh** files—if the header file contains only C constructions (structures, routine definitions, and so on), then you may leave it as a standard C **.h** file, included by means of the `#include` directive.

The rule of thumb is that if a header file contains any Goc code or includes a **.goh** file, then it must be a **.goh** file. Note also that **.goh** files are allowed to contain simple standard C code; if you are not sure, then, you can make all your header files **.goh**.

Standard C programs use the **#include** directive to include header (**.h**) files. When using Goc, you can use this directive in standard **.c** and **.h** files; when including **.goh** files in **.goc** files, though, you have to use the **@include** directive, which is Goc-specific. It has the same syntax as **#include**.

An example of using **@include** comes from the Hello World application, which includes the **stdapp.goh** file. (All GEOS applications will need to include this file to compile properly; it must be **@included** before any standard C headers are **#included**.) The line from Hello World is

```
@include <stdapp.goh>
```

The syntax of this directive, as stated above, is the same as for the C directive **#include**. One exception is that the **@include** directive will include a file just once, even if it is included by an included file—there is no need to conditionally include a file (checking first to make sure it hasn't already been included).

If you will be including a Goc file in many different applications, or if it is very long and elaborate, it is a good idea to put the keyword **@optimize** at the top of the file. This instructs the Goc preprocessor to generate a special stripped-down version of the file with a **.poh** suffix. The compiler will then automatically keep the **.poh** file up to date, and use it in compilations instead of the **.goh** file. The **.poh** file contains all the data of the **.goh** file, but is somewhat faster to compile into an application; thus, by using the **@optimize** keyword, you incur a longer compilation whenever you make a change to the **.goh** file, but a shorter compilation time when the **.goh** file is unchanged since the last compilation. You may choose to leave the **@optimize** directive out while the header is being developed, then put it in when the header is fairly stable.

5.2.2 Conditional Code in Goc

`@if`, `@ifdef`, `@ifndef`, `@endif`

Many C programs use the directives **#if**, **#ifdef**, **#ifndef**, and **#endif** to define conditional code—code that should be compiled into the program only if certain conditions are met. When working with standard C code in your GEOS applications, you should still use these directives; when working with Goc code, however (in **.goh** and **.goc** files), you should use the Goc directives **@if**, **@ifdef**, **@ifndef**, and **@endif**. 5.2

Goc conditionals are more limited than C conditionals. Conditional expressions may be based on numbers, names of macros, and the Boolean operators OR (`|`) and AND (`&&`). Some examples of Goc conditional expressions are shown below:

```
@ifdef (MyMacro)
    /* code compiled if MyMacro is defined */
@endif
@if 0
    /* code that will not be compiled at all */
@endif
@if defined(MyMacro) || MY_CONSTANT
    /* code compiled if either MyMacro is
       * defined or MY_CONSTANT is not zero */
@endif
@ifndef 0
    /* code always compiled */
@endif
```

5.2.3 Macros in Goc

`@define`

The C programming language allows definition and use of macros, and most programmers use macros extensively. You can use the **#define** directive in standard C code in your GEOS programs to define macros that use only standard C code.

Similarly, you can use the **@define** Goc directive to create macros in Goc. (Macros must be defined with **@define**; otherwise, the Goc processor will skip the **#define** directive and process the macro as if it were standard code to be processed normally.)

Macros in Goc have a somewhat different syntax than standard C macros though they are very similar. Some examples of simple Goc macros follow below:

```
5.2      @define mply(val1,val2)      val1 * val2
          @define defChunk(a)      @chunk char a[] = "text"
```

When using Goc macros in your code, you must preface them with the "@" Goc marker, indicating to the processor that it is a macro. If you do not preface the macro with "@", then Goc will pass over it and will not process it, leaving it up to the C compiler—which will likely give an error. For example, using the second macro defined above (defChunk), you could create a number of chunks easily:

```
@defChunk(firstText)
@defChunk(secondText)
@defChunk(thirdText)
```

The above would equate to the following:

```
@chunk char[] firstText = "text";
@chunk char[] secondText = "text";
@chunk char[] thirdText = "text";
```

Using "defChunk" without the "@" marker would most likely result in a compilation error in the C compiler.

5.2.4 Using Routine Pointers in Goc

```
ProcCallFixedOrMovable_cdecl(),
ProcCallFixedOrMovable_pascal()
```

Most GEOS code is kept in movable resources. If you call a routine explicitly from source code the Goc preprocessor generates appropriate directives to see to it that the resource is loaded into memory when it is called. However, if you call a routine with a routine-pointer, GEOS cannot take these precautions.

Accordingly, when you are calling a routine with a pointer, you must either see to it that the resource is loaded, or use one of the two

ProcCallFixedOrMovable routines to instruct the kernel to lock the appropriate resource.

If you know the routine is in a resource which is locked or fixed in memory, you can use the routine pointer exactly the way you would in standard C. This is usually because the calling routine is in the same resource as the routine or routines which may be called.

5.2

If you are not sure that the resource is loaded, you should call the routine with either **ProcCallFixedOrMovable_cdecl()** or **ProcCallFixedOrMovable_pascal()**. Each of these routines is passed the following arguments:

- ◆ A pointer to the routine to be called
- ◆ All the arguments passed to the routine, in exactly the order which the routine expects.

Both routines return whatever the called routine returns.

If the routine to be called was defined with standard C calling conventions (the default), you should use **ProcCallFixedOrMovable_cdecl()**. If the routine was declared with the keyword **_pascal**, it uses Pascal's calling conventions; you must then use the routine

ProcCallFixedOrMovable_pascal(). Most kernel and system-library routines are declared with Pascal's calling conventions.

Code Display 5-3 Using ProcCallFixedOrMovable_cdecl()

```
extern int
SomeRoutineCalledViaAPointer(int anArg, int anotherArg, const char *someText);

int (*funcPtr) (int, int, const char *); /* A function pointer */

funcPtr = SomeRoutineCalledViaAPointer;

/* We want to do
 *     SomeRoutineCalledViaAPointer(1, 2, "Franklin T. Poomm");
 * but we want to call it through the pointer, even though it's in another
 * resource:
 */
```



```
ProcCallFixedOrMovable_cdecl(funcPtr,    /* The pointer to the routine */  
    1, 2, "Franklin T. Poomm");
```

5.3 The GEOS Object System

5.3

GEOS is almost entirely object-oriented. Its object system supports true object-oriented principles such as encapsulation, inheritance, and message dispatching through the kernel.

The following section describes the class and object structures of GEOS, how to declare and define classes and objects, and how the messaging system and the kernel's message dispatcher work.

5.3.1 GEOS Terminology

Though you should be familiar with general object-oriented programming terms, there are quite a few for which the meaning is slightly different in GEOS, and there are others which are entirely new to GEOS. This section is divided into four categories: General Terms, Class Terms, Object Terms, and Messaging Terms.

5.3.1.1 General Terms

chunk	A chunk is a small section of memory located in a Local Memory Heap. Object instance data is stored in a chunk, one chunk per object. Local Memory and chunks are described fully in "Local Memory," Chapter 16.
fptr	An fptr is a "far pointer"—a 32-bit pointer to a specific location of memory. It is a standard C pointer.
handle	A handle is a 16-bit index into a Handle Table and is used to reference memory blocks. For more information, see "Handles," Chapter 14.

object block

An object block is a specific type of Local Memory block that contains object chunks.

optr

An optr is a unique identifier for an object and is therefore also referred to as an “Object Pointer.” An optr is used to locate and access any object in the system, and because it is made up of handles and chunk handles, it will always stay the same even when the object moves in memory.

thread

A thread is a single executable entity that runs either procedural code or one or more objects. If a thread is “event-driven,” it executes code for a given set of objects, receiving messages and dispatching them to the proper objects.

5.3

5.3.1.2 Class Terms

class

A class is the definition of a set of instance data structures and the methods that work on those structures. An object is called an “instance” of its class.

class tree

A class tree represents the hierarchy of inheritance from superclass to subclass. If a message is not handled by a given class, it will be automatically passed up the class tree until it is handled or the root of the tree is reached, after which the message is discarded.

inheritance

Inheritance is the term given to the way an object of a particular class has all the same instance variables and methods as an instance of the object’s superclasses.

initialize

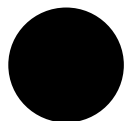
Initialization of an object is when a master part (or master group) of the object’s instance data is filled out. This occurs whenever a class in the master group not yet initialized in the object receives its first message.

master

The term “master” is used in several cases, all related. A master class is generally considered the top class in a single class tree. Although the master class may have superclasses, it provides a conceptual break and creation of a new subtree.

master group

The section of an object’s instance data belonging to a particular master class and all its subclasses is called a master



		group. A master group is initialized when the master class (or one of its subclasses) receives its first message.
	resolve	Resolution of a variant class occurs when the variant's superclass is determined. Each instance of a variant class must be resolved individually.
5.3	subclass	The term "subclass" is used to show relationships between classes. A subclass is defined on another class, from which it inherits instance data and methods. This other class is known as a "superclass," below.
	superclass	The term "superclass" is used to show relationships between classes. A superclass passes on its instance data and methods to all classes defined as subclasses of it.
	variant	A variant class may have different superclasses. However, an instance of a variant class may have only one superclass at any given moment. The use of variant classes can provide much the same functionality as the multiple inheritance found in some other object systems.

5.3.1.3 Object Terms

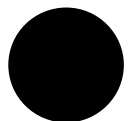
child	A child object is one that sits below another object in an object tree. The terms child, parent, and sibling are used only to show relationships between objects.
composite	A composite object is one that can have children. The composite has a "composite link" (an optr) to its first child and a "sibling link" to its next sibling. If it has no next sibling, the sibling link instead points to the object's parent object.
instance	An instance is a particular manifestation of a class. This term is almost always interchangeable with "object," though sometimes it specifically refers to the chunk containing the object's instance data rather than to the object as a whole.
link	A link is typically an optr pointing to an object's next sibling in an object tree. It is also used more generally to refer to any optr linking two objects in an object tree (parent and child, or last sibling and parent).

object	An object is a specific manifestation of a class. Typically, this term is interchangeable with “instance”; however, sometimes the term “object” refers to the combination of an object’s methods and instance data whereas the term “instance” refers to just the object’s instance data chunk.
object tree	An object tree is a means of organizing objects into a hierarchy for display or organizational purposes. Do not confuse it with the “class tree,” the structure which represents class relationships. An object tree is made up of composite objects, each of which may have children or be the child of another object. The topmost object in the tree is called the “root,” and the bottommost objects are called the “leaves” or “nodes.” Non-composite objects may be placed in the tree as leaves but may not have children.
parent	A parent object is one that has children in an object tree. The parent contains a composite link (an optr) to its first child and is pointed to by its last child.
state file	A state file is a Virtual Memory file used to store the state of objects. Typically, object blocks will be written to or extracted from the state file. Generic UI objects have this functionality built in automatically; other objects may manage their own state saving by managing the state file.

5.3

5.3.1.4 Messaging Terms

blocking	A thread “blocks” when it must wait for resources or return values from messages sent to objects in another thread. Specifically, a thread blocks when one of its objects sends a message to another thread with the “call” command; if the “send” command is used, the thread will continue executing normally.
call	A message sent with the call command causes the calling thread to block until the message is handled by the recipient. If the recipient is in the calling thread, the code will be executed immediately.
dispatcher	The GEOS dispatcher is internal to the kernel and passes messages on to their proper recipients. The dispatcher will



		dynamically locate the proper object and method and will invoke the method.
	message	A message is a directive, query, or other instruction sent from one object to another. Messages may take parameters and may return information.
5.3	method	A method, also called a “message handler,” is the code invoked by a message. A method may do anything a normal function or procedure may do, including alter instance data. It is poor style and highly discouraged for one object’s method to alter another object’s instance data directly.
	send	A message sent with the send command will be placed in the recipient’s event queue and will not cause the sender to block. Messages that return information or pass pointers should never be dispatched with the send command; use the call command in those cases.

5.3.2 Object Structures

You do not need to know what data structures are used to store objects and classes; understanding them can make programming GEOS much easier, however.

Each object is implemented in two parts: the instance data chunk and the class definition. Although both are integral parts of the object and they are interconnected, they are stored in different places.

An object’s instance data is stored in an instance chunk. This instance chunk is sometimes referred to as the object itself, but this isn’t quite accurate—the instance chunk contains only the object’s data along with a pointer to its class structure. The structure of the instance chunk is given in section 5.3.2.1 on page 159.

An object’s class structure contains all the code for the class. Since the class code may be accessed by many objects, the class definition resides in a geode’s fixed memory resource. Every class (except the root, **MetaClass**) has a pointer to its superclass so it can inherit that class’ methods and structures.

All objects of a given class use the same code—the class’ code—for their functions. They dynamically access this code so the code blocks need to be in

memory only once, no matter how many objects are actively using them. Additionally, each class dynamically accesses its superclass' code, so any class may be accessed by all the objects of the subclasses as well. Class structures are shown in section 5.3.2.3 on page 164.

5.3.2.1 Instance Chunk Structures

Each object's instance data is stored in a Local Memory chunk. Several chunks are stored in one memory block, called a local memory heap. (See "Local Memory," Chapter 16.) This local memory heap, containing objects, is known as an *object block*.

5.3

Each object block has a special header type that distinguishes it from normal local memory heaps. After the header in the block is the chunk handle table: a table containing offsets to each object in the block. Following the chunk handle table are the objects.

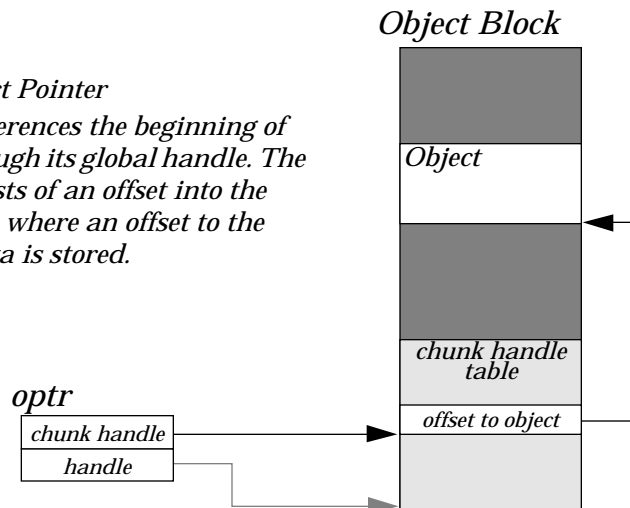
Each object is referenced by an object pointer, or *optr*. The *optr* contains two items: the global memory handle of the object block and the chunk handle of the object. Note that because the *optr* is made up of handles, an object must be locked before its instance data can be accessed. The GEOS kernel takes care of this automatically when executing methods. For an illustration of how an *optr* references an object's instance chunk, see Figure 5-1.

Only the object's instance data is stored in the chunk pointed to by the *optr*; the method table and code used by the object are stored in the class' data structures, not the object's. To reconcile this separation of code and data, every object's first four bytes of instance data are a pointer to the object's class definition. This pointer is traversed by the kernel automatically and should not be altered or accessed by applications. A simplified diagram of how this pointer allows the kernel to traverse class trees when handling messages is shown in Figure 5-2.

Included in an object's instance chunk are certain fields generated and filled by either Goc or the kernel. Following these fields is the object's instance data, grouped by master part. It's unlikely you'll ever have to know the actual structures used in the instance chunk because the kernel automatically calculates the proper offsets to individual instance data fields. However, understanding the underlying structures may help in understanding how the object system of GEOS works.

Figure 5-1 *An Object Pointer*

The optr (below) references the beginning of the object block through its global handle. The chunk handle consists of an offset into the chunk handle table, where an offset to the object's instance data is stored.



Instance data within an instance chunk is stored in “master parts” or “master groups.” A master group is simply a number of instance data fields grouped according to their appropriate master class levels. Master classes are detailed in section 5.3.2.2 on page 163.

A class designated as a master class resembles a normal class in all respects save one: it determines how instance data is grouped in a chunk. Each master class is the head of a class subtree; all the classes below it in the class tree (down to the next master class) are considered to be in that master class’ group. Instance data for all classes in the master group are lumped together in the instance chunk; each master group’s instance data within the chunk is accessed via a special offset stored within the chunk.

Sample instance chunks are shown in Figure 5-3. The first four bytes of an object’s chunk contain a pointer to the object’s class structure. The class structure (described in section 5.3.2.3 on page 164) resides in fixed memory. (A variant-class object has a slightly different structure; this is detailed in section 5.3.2.4 on page 171.)

An object that has no master classes in its class ancestry (unusual) has its instance data directly following its class pointer. Objects belonging to master

classes or their subclasses, however, are somewhat more complex. This distinction can be mostly ignored by application and library programmers (with the exception of deciding which classes should be master classes and which should not).

Each master part of the chunk is located by an offset inserted directly after the object's class pointer in the chunk. The position of the word containing this offset is then stored in the master class structure so the class can find its instance data later. The combination of the class pointer and the various master offsets make up the object's "base structure." When a typical object is instantiated, the base structure is all that is created.

5.3

Each master part is left unallocated (for efficiency) until it is first accessed via a message sent to a class in the master group. When a class in the master group receives its first message, the entire master part of the chunk is allocated and initialized. This means that an object's chunk remains as small as possible until it absolutely must grow larger. Some classes even detect when a master part of the object will no longer be needed and actually remove (shrink to zero) the unwanted instance data from the chunk (**GenClass** does

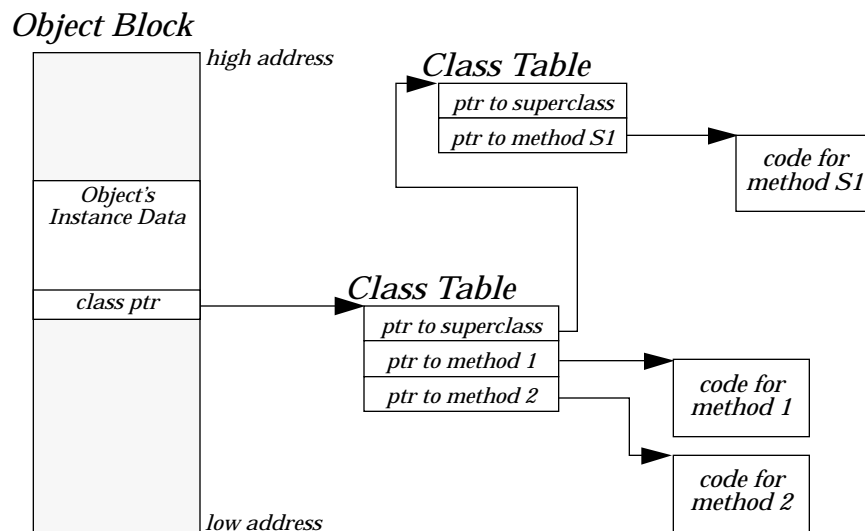


Figure 5-2 Object and Class Interaction

The kernel uses pointers internal to classes to traverse the class tree to the appropriate message handlers and data structures.



this by shrinking the Vis master part to zero size when a gadget is taken off the screen).

Any object may have “variable data” instance data fields; these are fields that may be added or removed dynamically to keep from having unused space in the instance chunk. Generic UI hints are “variable data” (also called *vardata*)—if an object has the hint, it appears in its instance chunk, if the object does not have the hint, the chunk does not have unused space in it.

5.3

Vardata entries are stored all together at the end of the instance chunk, regardless of their master groups. An object with two master groups and three variable data fields, for example, would look like Figure 5-4. Variable data and its use are discussed in full in section 5.4.1.4 on page 195.

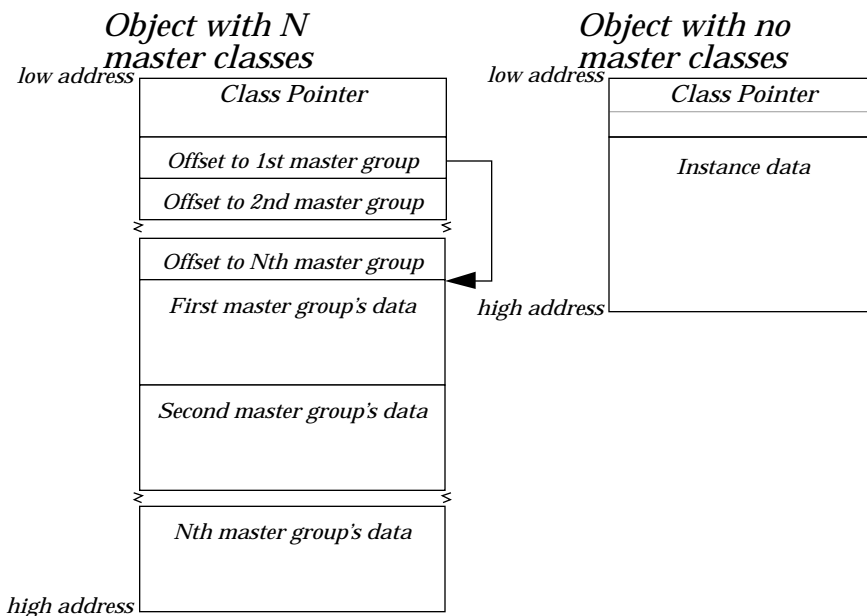


Figure 5-3 *Structures of an Object*

All objects have class pointers, though only those with master classes in their ancestries have master groups and master group offsets.

5.3.2.2 Master Classes



Advanced Topic

Most programmers will not have to understand the implementation of master and variant classes.

A master class provides a conceptual break between levels within a class tree. Each master class is the head of a class subtree, and all its subclasses are considered to be in its “master group.” Figure 5-5 shows a simplified diagram of a class tree and its master groups.

The purpose of making a class a master class is to separate its instance data from that of its superclass. Each master group’s instance data is lumped together in one section of the object’s instance chunk and is not initialized until a class in the master group is accessed. The initialization (allocation of extra memory within the instance chunk) occurs automatically.

5.3

As shown in Figure 5-5 and Figure 5-6, an object of **RookClass** would have an instance chunk with two master groups, one for the **PieceClass** master class and one for the **GamePcClass** master class. The first of the two master parts represents the instance data for **PieceClass** only; the second master part represents the object’s instance data for all of **GamePcClass**, **ChessClass**, and **RookClass**.

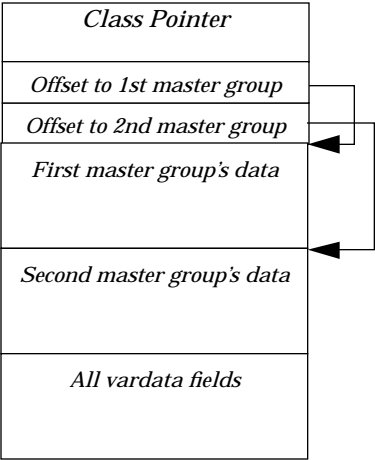


Figure 5-4 *An Object with Vardata*
All vardata entries are stored at the end of the instance chunk, regardless of the master group with which they are associated.

5.3

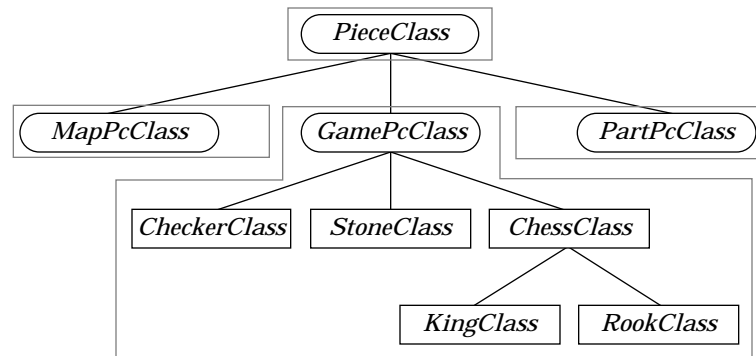


Figure 5-5 Master Classes and Master Groups

Each master class heads its own master group. The outlined classes are all in the same master group.

The functionality of master classes is required to implement GEOS variant classes (see section 5.3.2.4 on page 171). A variant class allows a single class to have a version of “multiple inheritance” in that it can have different superclasses depending on the system context.

5.3.2.3 Class Structure and Class Trees



Advanced Topic

Most programmers will never access the class structures.

For the most part, you won’t ever need or want to know the internal structure of a class as implemented in memory. The class structure is created and partially filled by the Goc preprocessor and Glue linker; the remainder is filled by the kernel when the class is loaded. It’s unlikely you will need to know the actual class structures; you won’t ever have to manually build a class unless your program dynamically creates it (not a common procedure for typical applications).

This section will describe how the class is implemented and how class trees are structured and managed. However, it will not discuss creating new classes during execution.

Classes are implemented with special data structures and code blocks. Each class is defined within and exported by a particular geode; when the geode is loaded the class definition and its code are loaded into the geode’s fixed

memory. All references to the class are then relocated by the kernel into pointers. For example, if a class is defined by a library, that library's "core block" (the special information kept about it by the kernel) contains an absolute pointer to the class' definition in a fixed memory resource owned by the library. Any applications then using that class load the library. The kernel examines the library's core block for the proper pointer and uses it each time the application references that class.

Because of this, each class is loaded into memory just once; all objects that use the class use the same class structure and code. Each object has a pointer in its instance chunk directly to the class structure; each class contains a pointer to its superclass' class structure. Using these pointers, the kernel can travel up an object's class tree to access any appropriate code. See Figure 5-7 for a simplified illustration of how these pointers are followed by the kernel.

5.3

A class is a combination of data structure and code. The data structure (**ClassStruct**) contains information about the class, its superclass, its methods, and the structure and size of its instance data. The code consists of methods (message handlers). A diagram of the data structure is given in Figure 5-8; its components are detailed below.

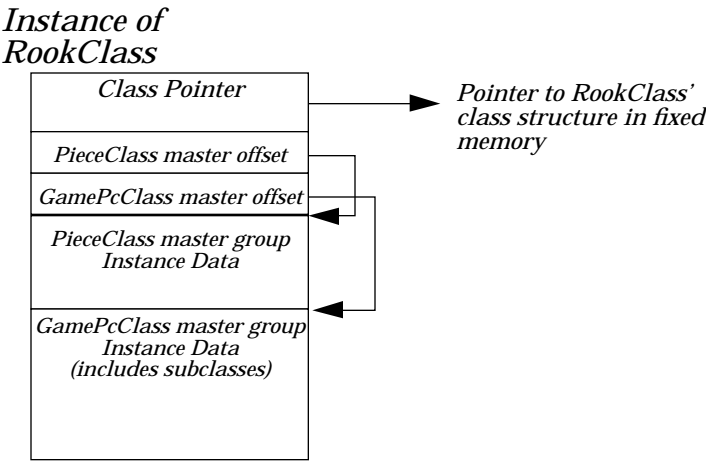


Figure 5-6 *A Sample Instance Chunk*
The RookClass chunk has two master groups, each having an offset stored after the class pointer.

5.3

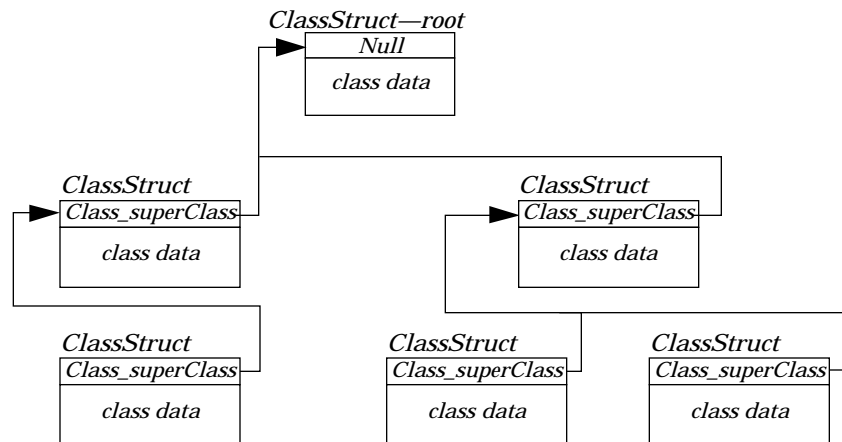


Figure 5-7 A Class Tree

It is simple to find the root of any class tree by following `Class_superClass` links. This is how inheritance is implemented for methods; if a class does not recognize a message, the kernel looks in the superclass.

Class_superClass

Every class has as its first four bytes a pointer to its superclass. This points to the superclass' **ClassStruct** structure in all cases except two: The root of any class tree has a null superclass pointer, indicating that the root has no superclass. Variant classes have the integer 1 (one) always, indicating that the superclass is determined in a special manner. For more information on variant classes, see section 5.3.2.4 on page 171.

Class trees are constructed when classes are defined; a new class is created as the subclass of some existing class, and its `Class_superClass` pointer is automatically set to point to the superclass. There is no need to point down the tree; messages are always passed to superclasses and never to subclasses. An example of the use of `Class_superClass` is shown in Figure 5-7.

Class_masterOffset

Class_masterOffset stores the offset indicating how far into the instance chunk the object's offset to this class' master part is. Figure 5-9 shows how this field allows a class to locate the appropriate master part. Note that use of this offset is entirely internal; individual classes do not have to figure out where their instance data is within the chunk (they may, however, have to know what master level each class is).

The master offset is used primarily because an object can have some of its master parts initialized and others uninitialized. If only one master part of

5.3

ClassStruct

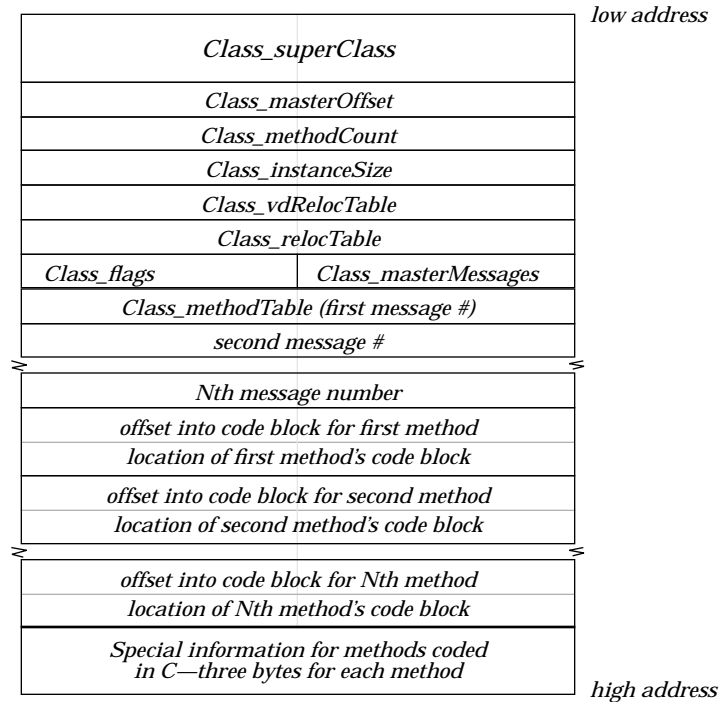


Figure 5-8 The *ClassStruct* Structure

This structure has two basic parts: The class information and the method table. The class information has eight fields in eight words; the method table varies in size and consists of a message number table followed by a table of pointers to the actual method code.



the object had been initialized, the location of the instance data in the chunk may be different than if all master parts had been initialized.

Class_methodCount

Class_methodCount stores the total number of methods referenced in the class' method table. This is the total number of methods defined for this class only; other methods defined in other classes (even in the same master group) are stored in the method tables of those classes.

5.3

Class_instanceSize

Class_instanceSize holds the number of bytes to be allocated whenever an object of this class is instantiated. If the class is a master class or a subclass of a master class, this is the size of the master part. If the class has no master class above it, this is the number of bytes to allocate for the entire object (including superclass pointer).

Class_vdRelocTable

Class_vdRelocTable is a near pointer (16 bit offset) to the variable-data relocation information. The relocation information contains the type of relocation to be done for each data type. There is one entry in the variable

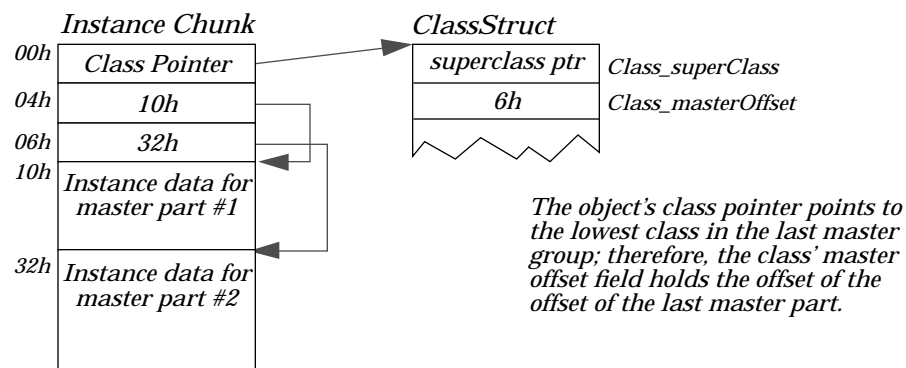


Figure 5-9 *Class_masterOffset*

The instance chunk holds an offset to the master group's instance data; this offset is referenced by the class' *Class_masterOffset* field.

data relocation table for each relocatable field in each particular variable-data type. Variable data (also called *vardata*) is described in full in “Defining and Working With Variable Data Fields” on page 195.

Class_relocTable

Class_relocTable is a near pointer (16 bit offset) to the relocation information for the non-variable data instance fields of the class. The relocation information contains the type of relocation done for each relocatable instance field (other than variable-data entries). A relocatable instance field is one which must be updated when the object is loaded—pointers, offsets, etc. The entry in the relocation table is defined with the **@reloc** keyword, described on page 202.

5.3

Class_flags

Class_flags contains seven flags (shown below) that determine the characteristics of the class. Declarers for these flags are used in the **@classdecl** declaration (see section 5.4.1 on page 184).

CLASSF_HAS_DEFAULT

This flag indicates that the class has a special default method to handle unrecognized messages (typically, this handler simply passes the unrecognized message on to the superclass). This flag is not implemented in C. This flag is set by declaring the class as *hasDefault*.

CLASSF_MASTER_CLASS

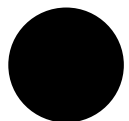
This flag is set if the class is a master class. This flag is set by declaring the class as *master*.

CLASSF_VARIANT_CLASS

This flag is set if the class is a variant class. This flag is set by declaring the class as *variant* (all variants must also be declared as masters).

CLASSF_DISCARD_ON_SAVE

This flag indicates the instance data for the class can be discarded when the object is saved. This flag applies only to master classes and will be heeded only when the master group is immediately above a variant-master group. This flag is set by declaring the class *discardOnSave*.



CLASSF_NEVER_SAVED

This flag indicates objects of this class are never saved to a state file or loaded in from a resource. Goc will not build a relocation table for a class with this flag set. This flag is set by declaring the class *neverSaved*.

CLASSF_HAS_RELOC

This flag indicates that the class has a special routine for relocating and unrelocating objects of the class when they are loaded from or written to disk. This flag is set by declaring the class *hasReloc*.

CLASSF_C_HANDLERS

This flag indicates the class' methods are written in C rather than in assembly. This flag is set by the compiler and should not be set by applications.

Class_masterMessages

Class_masterMessages contains a number of flags set by Goc indicating whether this class has methods for messages defined within a given master level. It is used to optimize the search for a method to handle a message. These flags are internal and should not be used by programmers.

The Method Table

Every class has a method table, a table that indicates the location of the code to be executed when a certain message is received. The method table is in three parts and begins at the byte labelled *Class_methodTable* (this is simply a label, not a specific data field).

The first part of the method table is a list of messages the class can handle. Each entry in this list is two bytes and contains the message number of a message handled by the class.

The second part of the method table is a list of pointers to methods. Each entry in this list is a pointer (four bytes) which points to a specific routine in a code block. If the code is in a fixed block, the pointer will be a far pointer; if the code is in a moveable or discardable block, the pointer will be a far pointer containing a *virtual segment*. (A virtual segment, something you do not need to know about, is a handle shifted right four bits with the top four bits set. Since this represents an illegal segment address, GEOS recognizes it as a

virtual segment and will take the necessary actions to lock the block into memory before access and unlock it after access. Manipulation of the bits in the virtual segment is completely internal to GEOS.)

The kernel searches the message list until it comes across the appropriate message number and notes the message's position in the table. It then looks at the corresponding position in the pointer list. If the pointer there is a virtual segment and offset, it will load the appropriate code block, lock it, and execute the code. If the pointer points to fixed memory, the code will be executed immediately. (If the message number is not found in the table, the kernel will either execute the class' default handler or pass the message on to the class' superclass.)

5.3

5.3.2.4 Variant Classes

A variant class is one which has no set superclass. The variant's superclass is determined at run-time based on context and other criteria. Note that *objects* may not be variant—only classes may be variant. An object always has a specific class to which it belongs, and its class pointer *always* points to that class' **ClassStruct** structure. In addition, every variant class *must* also be a master class.

A variant class, however, may have different superclasses at different times. This functionality provides a form of “multiple inheritance”: the class may inherit the instance data and functions of different classes depending on its attributes and desired features. Note, however, that a variant class may have only one superclass at any given moment.

The most visible example of a variant class is **GenClass** and how a generic object is resolved into its specific UI's appropriate representation. Each generic object (for example, a GenTrigger), is a subclass of the master class **GenClass**. All the instance data belonging to **GenTriggerClass** and **GenClass**, therefore, is stored in the Gen master part of the instance chunk.

GenClass, however, is a variant class, meaning that it does not know its superclass when the object is instantiated. Each generic object's class will be linked directly to another class provided by the specific UI in use: the specific UI's class provides the visual representation while the generic UI class provides the object's functionality. In this way, the object can actually



perform many of its generic functions without having a visual representation.

The resolution of the superclass comes when the generic object is displayed on the screen: the kernel sees that the object has no superclass and looks into its instance data and class structure. The kernel then determines what the appropriate specific UI class will be for the object's class and provides the superclass link necessary. It also then initializes the superclass' master part of the object (in this case, the master part belonging to **VisClass**), updating all the master part offsets in the instance chunk's master offset fields.

You can see from the above discussion that **GenClass** must know at least something about its potential superclasses. In fact, all variant classes must know at least the topmost class of all its potential superclasses. The definition of **GenClass** is

```
@class GenClass, VisClass, master, variant;
```

The **@class** keyword declares the new class, **GenClass**. **GenClass** is to be a variant class and therefore must also be a master class. All the superclasses of **GenClass** will be related to **VisClass**; this means that all specific UI classes which may act as Gen's superclass must be subclassed from **VisClass**. (Another way of looking at the definition is that **GenClass** is an *eventual* subclass of **VisClass**—you have no way of knowing beforehand how many class layers may be between the two, however.)

The variant must specify an eventual superclass so the kernel knows how many master offset fields it must allocate when an instance of the variant is created. For example, a **GenTrigger** has two master groups: that of **GenClass**, and that of **VisClass**. Because the **GenClass** master group is necessarily below the **VisClass** master group in the class hierarchy (after the superclass link has been resolved), the **GenClass** master offset in the instance chunk must be after the **VisClass** master offset. If the definition did not specify **VisClass** as an eventual superclass, no master offset field would be allocated for it, and the *Class_masterOffset* field of **GenClass**' Class structure would not be able to hold any particular value.

As stated at the beginning of this section, there are no “variant objects.” Every object belongs to a specific class, and the object's class can never change. All instances of a variant class, however, can be resolved to different superclasses due to the way the superclass of each variant is resolved. One

example of this is the generic-to-specific mapping of the **GenInteraction** object.

All **GenInteractions** are of class **GenInteractionClass**; this never changes. **GenInteractionClass**, however, is a subclass of **GenClass**, a variant class. This means that the class tree of the **GenInteraction** object is only partially completed; before the **GenInteraction** is resolved, it looks like the simplified diagram in Figure 5-10.

5.3

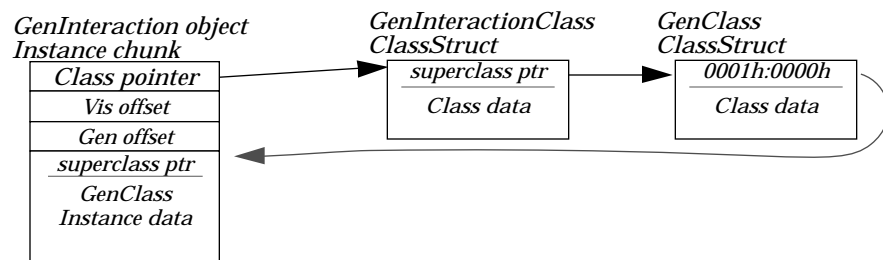


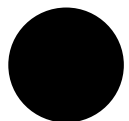
Figure 5-10 A Variant Class Object

A variant class always has the constant 0001h:0000h instead of a superclass pointer. The class pointer of the variant class is stored in the first four bytes of the variant class' master part in the instance chunk. Until the variant class is resolved, this pointer is null and the master part of the superclass (in this case **VisClass**) is not initialized.

The **GenInteraction** object may be resolved into one of several different specific UI classes. For example, the **motif.geos** library has several classes for **GenInteractions**; among them are the classes that implement menus, dialog boxes, and grouping interactions. These classes are all specialized subclasses of **VisClass**, the eventual superclass of **GenClass**.

Notice from Figure 5-10 that the class tree of the **GenInteraction** is not complete. A class tree must have links all the way back to **MetaClass** for it to be complete; this only goes to **GenClass**. **GenClass** has a special value in its *Class_superClass* field, 0001h:0000h. This represents a reserved "pointer" that indicates to the kernel that the class is a master class.

The superclass of the variant can be different for every instance because the superclass pointer is actually stored in the object's instance chunk rather



5.3

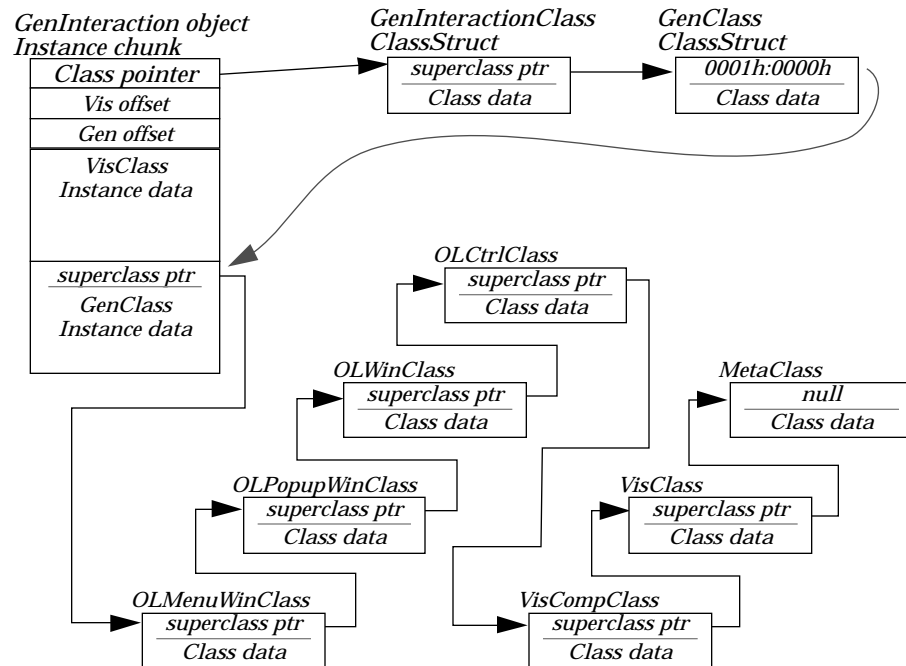


Figure 5-11 A Resolved Variant Object

The *GenInteraction* menu object has been resolved to belong to *OLMenuWinClass*, thus completing the object's class tree as shown.

than in the class' **ClassStruct** structure. This also allows a class tree to have more than one variant class in its hierarchy; for example, one variant could be resolved to be the subclass of another variant. The tree must always be headed by **MetaClass**.

As shown in Figure 5-10, the superclass pointer for the variant is stored in the variant's master group instance data. Not all master groups have superclass pointers; only those for variant classes. After the *GenInteraction* is resolved, the pointer (the first four bytes of the *Gen* master part) points to the proper superclass for this object (in this case, **OLMenuWinClass**). The object, with its full class tree, is shown in Figure 5-11.

5.3.2.5 An In-Depth Example

This section gives an example of a GenTrigger object after its variant part has been resolved. This example provides in-depth diagrams of the class and instance structures for those programmers who wish to understand them. There is no need to know them, however; you will not likely ever need to access the internals of either a class structure or an instance structure.

The GenTrigger taken as an example is the “New Game” trigger of the TicTac sample application. This trigger is the only child of the Game menu GenInteraction; it is shown in Figure 5-12. The code defining the trigger is given in Code Display 5-4.

5.3

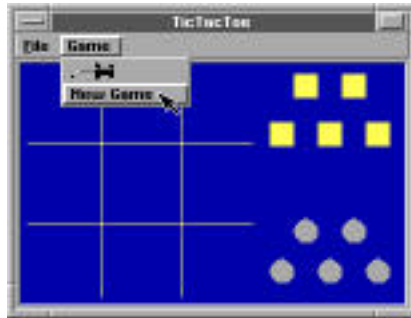
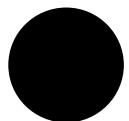


Figure 5-12 *TicTac's New Game Trigger*

This trigger is used as the in-depth example object. The mouse pointer in the diagram points to the trigger.

Code Display 5-4 TicTac's New Game Trigger

```
/* The TicTacNewTrigger has a moniker and an output. All its other instance data
 * is set to the GenClass defaults. The content of these fields is not important
 * for this example, however. */
```



```
@object GenTriggerClass TicTacNewTrigger = {
    GI_visMoniker = "New Game";
    GTI_destination = TicTacBoard; /* Send the action message to the
                                   * TicTac game board object. */
    GTI_actionMsg = MSG_TICTAC_NEW_GAME; /* The action message. */
}
```

5.3

The GenTrigger's Instance Chunk

The GenTrigger object has two master parts, just like the GenInteraction object shown in "Variant Classes" on page 171: the Gen master part holds the instance data for **GenClass** and **GenTriggerClass**. The Vis master part

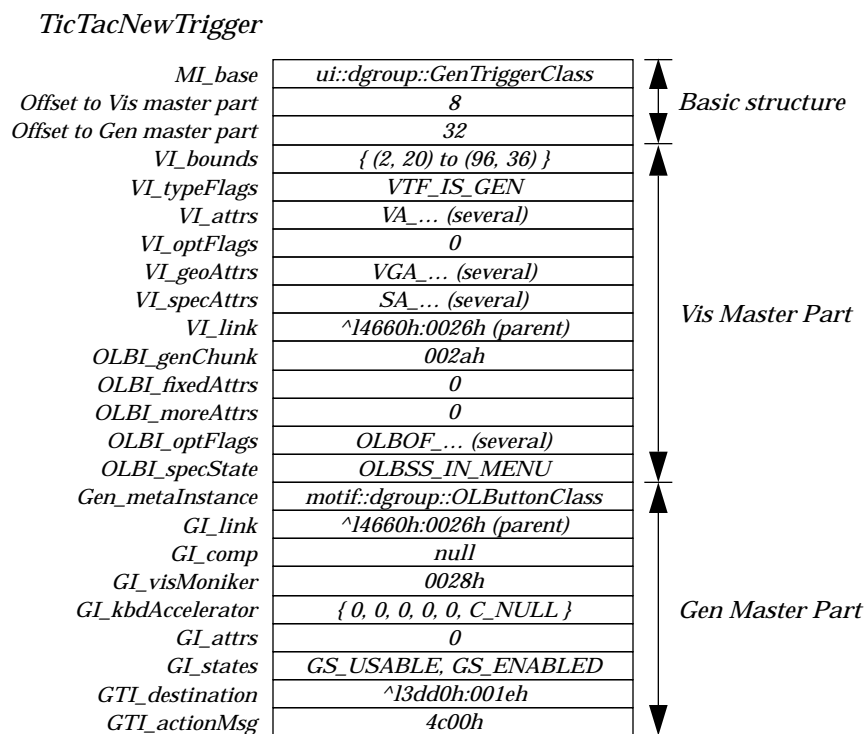


Figure 5-13 *GenTrigger's Instance Chunk*

The *TicTacNewTrigger* trigger has a Gen and a Vis master part and no vardata entries.

holds the instance data for **VisClass** and **OLButtonClass**. The **MetaClass** instance data consists only of the object's class pointer and has no master part.

Figure 5-13 shows the structure of **TicTacNewTrigger**'s instance chunk. The chunk's basic structure consists of the class pointer (four bytes) followed by two words of offset into the chunk. The first offset gives the location of the **Vis** master part, and the second gives the location of the **Gen** master part. After the offsets are the master parts themselves, and if the trigger had any variable data, it would appear at the end of the chunk.

5.3

Each master part has the master class' instance fields first, followed by those of its subclasses. All the fields that belong to **VisClass** begin **VI_...**, all those that belong to **OLButtonClass** begin **OLBI_...**, etc.

Notice also the first four bytes of the **Gen** master part: they contain a pointer to the "superclass" of **GenClass** for the trigger. Although the trigger typically does not have different forms in any given specific UI (as the **GenInteraction** does), it will have a different class for each specific UI it encounters. For example, the **OSF/Motif** class is **OLButtonClass**; another specific UI will use a different class for **GenTriggers**.

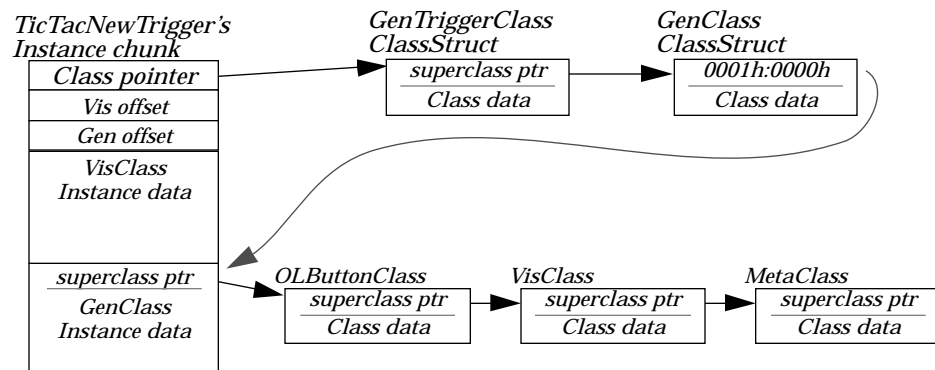
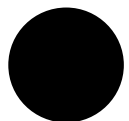


Figure 5-14 *TicTacNewTrigger's Class Tree*

TicTacNewTrigger is of **GenTriggerClass**, a subclass of **GenClass**. **GenClass** is a variant and is resolved to **OLButtonClass** at run-time. **OLButtonClass** is subclassed off **VisClass**, which is subclassed off **MetaClass**.



The GenTrigger's Full Class Tree

Figure 5-14 shows TicTacNewTrigger's full class tree in a simplified diagram. Since **GenClass** is a variant, it has a superclass pointer of 0001h:0000h. This special value (with an illegal segment address) indicates to the kernel that this object's **GenClass** superclass is stored in the instance chunk itself. The superclass is stored in the first four bytes of the Gen master part, as shown in the previous section.

5.3

GenTriggerClass' ClassStruct Structure

Because all classes have the same class structure, only **GenTriggerClass** will be examined here. The class structure and the instance chunk structure are closely linked in several ways, as shown in Figure 5-15.

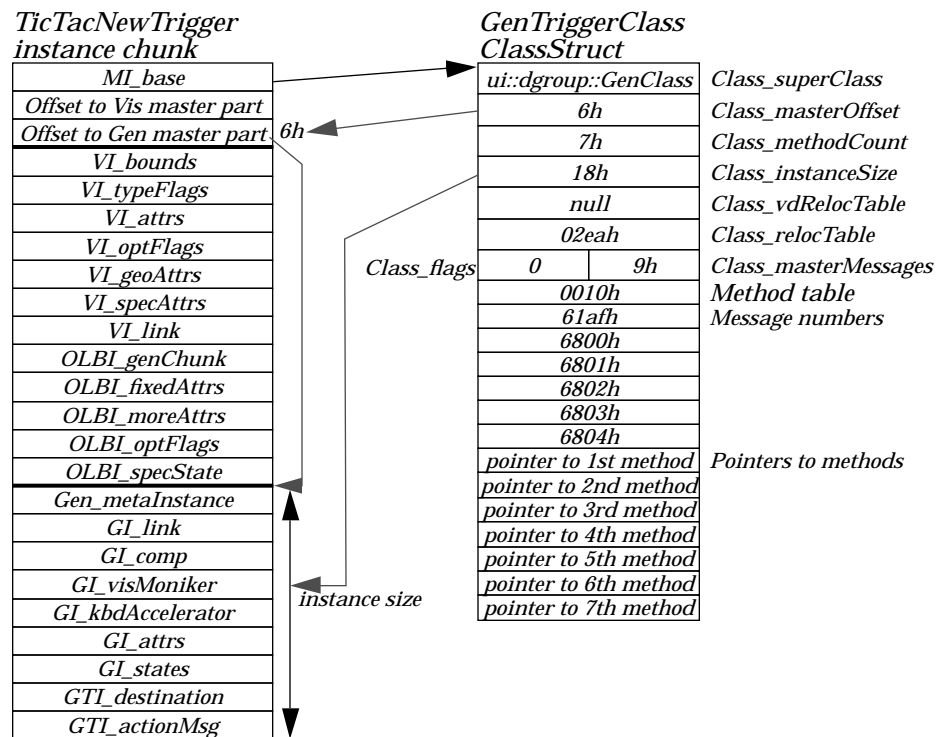


Figure 5-15 GenTriggerClass Subclassed

The class structure and instance structure are linked in several ways.

As shown in the diagram, the instance chunk points directly to the class. The class points to its superclass, thereby providing inheritance of all the methods and structures of classes higher in the class tree such as **GenClass**.

The class structure contains some information about the instance chunk's format, specifically *Class_masterOffset* and *Class_instanceSize*.

Class_masterOffset gives the offset into the instance chunk where the offset to the master part is stored. *Class_instanceSize* contains the size of the master part so the kernel can quickly allocate the needed space when the master part is initialized.

5.3

The method table resides at the end of the class, and it has entries for each message handled by the class. **GenTriggerClass** handles seven messages (stored in *Class_methodCount*); any message received by this trigger and not recognized by **GenTriggerClass** is passed up the class tree for handling. Thus, a MSG_GEN_SET_NOT_ENABLED sent to the trigger will be passed on to **GenClass** and will be handled there.

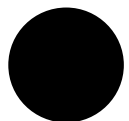
The method table has two parts: The first part is a listing of message numbers, and the second part is a listing of pointers to the method code. When the object receives a message, the kernel scans the first part to see if the class handles the message. If it does, the kernel then scans the second part of the table to get the pointer to the code. It then executes the code there as if that code were a function called by the object's code.

How a Message Is Handled

Most aspects of messages and messaging are described in the following section. This section, however, describes how the kernel finds and executes the code when a message is sent to the GenTrigger. The message is MSG_GEN_SET_USABLE (handled by **GenClass**).

Messages are sent directly to an object using its optr. That is, when you send a message to this particular GenTrigger, you send it directly to TicTacNewTrigger, not to some monolithic "case" statement run by your application. Since the object's optr uniquely identifies the location of the object's instance chunk in memory, the kernel can quickly access the code for the handler.

When MSG_GEN_SET_USABLE is sent to the TicTacNewTrigger, for example, the kernel looks in the object's instance chunk for its class pointer. It follows



this pointer and then looks in **GenTriggerClass'** **ClassStruct** structure. It scans the first part of the class' method table for MSG_GEN_SET_USABLE. If the message is not there (and it isn't), the kernel will then follow the class' *Class_superClass* pointer and look in **GenClass'** **ClassStruct** structure. It then scans the first part of **GenClass'** method table for the message. **GenClass** has an entry for MSG_GEN_SET_USABLE, and therefore the kernel checks the second part of the method table for the code pointer. It follows this pointer to the method's entry point and begins executing the code there.

5.3.3 The GEOS Message System

Because objects are independent entities, they must have some means of communicating with other objects in the system. As shown in the example of the calculator and requestor objects in "System Architecture," Chapter 3, communication is implemented through the use of messages and methods.

5.3.3.1 The Messaging Process

When an object needs to notify another object of some event, retrieve data from another object, or send data to another object, it sends a message to that object. Sending a message is similar to calling a function in C in that the message can take parameters (including pointers) and give return values. However, messages are also quite different from function calls in the multithreaded environment of GEOS.

Each object block in the system is associated with a single thread of execution. Each thread that runs objects (some run only procedural code) has an *event queue*, a queue in which messages are stored until they can be handled. Every message sent to an object from another thread gets put in the object's thread's event queue. (Messages sent between objects within the same thread are not handled via the queue unless forced that way.) Thus, a single thread's event queue can have messages for many different objects. For most single-thread applications, the programmer will not have to worry about synchronization issues.

The sender of a message has to be aware of synchronization issues raised by having multiple threads in the system. Essentially, you can send a message

two ways: The first, “calling” the message, allows the use of return values and acts almost exactly like a function call in C. This places the message in the recipient’s event queue and then halts the sender until the return values are received. The sender “goes to sleep” until the message has been processed and is then awoken by the kernel, thus ensuring the message is handled before the sender executes another line of code. The call option should also be used when passing pointers; otherwise, the item pointed to may move before the message can be handled, invalidating the pointer.

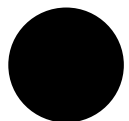
5.3

The second, “sending” the message, is used primarily when synchronization is not an issue. For example, if the message merely notifies the recipient of some condition or event, or if it sends data with no expectation of a return value, the send option can be used. Essentially, the sender will send off the message and then forget about it, continuing on with its own business. (The exception to this is an object sending a message to another object in the same thread; then the message is handled immediately, as if it had been called.)

When an object sends a message, the message actually first gets passed to the kernel (this is all automatic). The kernel will put the message into the proper thread’s event queue and, if necessary, put the sender’s thread temporarily to sleep. When the recipient’s thread is ready to handle the message, the kernel pulls it from the event queue.

The kernel then locates and loads the recipient object into memory (if necessary). The recipient’s object block will be loaded and locked, and the object will be locked while processing the event. (Note, however, that it is possible for the object to move if the recipient makes a call to **LMemAlloc()** or does something else that can cause shuffling of the object chunks.) The kernel will follow the object’s class pointer to its class and will scan the method table. If the class can handle the message, the proper method code will be executed. If the class can not handle the message, the kernel will follow the superclass pointer and check the method table there. The message will continue up the class tree like this until either it is handled or it reaches the root and returns unprocessed.

After the method code has been executed, the kernel collects any return values and wakes up the caller thread again if necessary. To the caller, it’s as if the message were handled instantaneously (with the call option). Senders are never blocked; only messages called (with the call option) may block the caller’s thread. If a message is sent to an object in the same thread, however,



it will be executed as a call and will be handled immediately, unless the sender explicitly states that it should go through the message queue.

Be careful, though, if you are writing code in multiple threads (for example, if you subclass UI objects and write new method code for them). You have to make sure that two threads never call each other; this can lead to deadlock if the calls happen to overlap. The easiest way to deal with this is to have one thread always send a message requesting a return message with any needed return values. The other thread will then send off a return message with the data. For example, a UI object may require information from an application's object. The UI object sends `MSG_REQUEST_INFORMATION` (or something similar). The application's object then receives that message and responds with a `MSG_RETURNING_REQUESTED_INFORMATION` (or something similar). With this scheme, the application's object is free to use call whenever it wants, but the UI object must always use send.

5.3.3.2 Message Structures and Conventions

A message is simply a 16-bit number determined at compile time. Specifically, it is an enumerated type—this ensures that no two messages in the same class can have the same number.

An event is an opaque structure containing the message number and information about the recipient, the sender, parameters, and return values. When an object sends a message, the kernel automatically builds out the event structure (generally stored in the handle table for speed and efficiency). You will never have to know the structure of an event.

5.4 Using Classes and Objects

The previous sections dealt with the internals of the GEOS object system. This section describes how you can create classes and objects and manage them during execution using Goc keywords and kernel routines. Almost all Goc keywords begin with “@” (one notable exception is **gcnList**).

All the most useful keywords available in Goc are shown in Code Display 5-5. This display is for initial reference; all the keywords are detailed in depth in the following sections of this chapter and in the Routines Book.

Code Display 5-5 Goc Keywords

```

/* Including .goh files */
#include <fname>;

/* Defining New Classes and Subclasses */
@class <cname>, <super> [, master [, variant]];
@endc

/* Declaring a class */
@classdecl <cname> [, <cflags>];

/* Defining messages for a class */
@message <retType> <mname>([@stack] <param>*);
@reserveMessages <num>;
@exportMessages <expName>, <num>;
@importMessage <expName>, <messageDef>;
@alias(<protoMsg>) <messageDef>;
@prototype <messageDef>;

/* Defining instance data fields for a class */
@instance <insType> <iname> [ = <default>];
gcList(<manufID>, <ltype>) = <oname> [, <oname>]*;
@instance @composite <iname> [ = <linkName>];
@instance @link <iname> [ = <default>];
@instance @visMoniker <iname> [ = <default>];
@instance @kbdAccelerator <iname> [ = <default>];
@reloc <iname>, [ (<count>, <struct>), ] <ptrType>;
@noreloc <iname>;
@default <iname> = <default>;

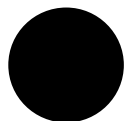
/* Defining vardata fields for a class */
@vardata <type> <vname>;
@vardataAlias (<origName>) <newType> <newName>;
@reloc <vname>, <fn>, [ (<count>, <struct>), ] <ptrType>;

/* Defining methods (message handlers) */
@method [ <hname>, ] <cname> [, <mname>]+;
@method [ <hname>, ] <cname> _reloc;

/* Defining library code and resources */
@optimize
@deflib <libname>
@endlib
@start <segname> [ , <flags> ];
@header <type> [ = <init> ];
@end <segname>
@chunk <type> <name> [ = <init> ];

```

5.4



```

@chunkArray <stype> <aname> [ = {<init>} ];
@elementArray <stype> <aname> [ = {<init>} ];
@extern <type> <name>;
@gstring;

/* Declaring an object */
@object <class> <name> <flags>* = {
    [<fieldName> = <init>];*
    [<varName> [ = <init> ]]*;
}

5.4

/* Sending and calling messages */
@send [<flags>,+<obj>::[<cast>]] <msg>(<params>*);
<ret> = @call [<flags>,+<obj>::[<cast>]] [<cast2>]]<msg>(<params>*);
@callsuper();
@callsuper <obj>::<class>::<msg>(<params>*) [<flags>,];
<event> = @record <obj>::<msg>(<params>*);
@dispatch [noFree] <nObj>::<nMsg>::<event>;
<ret> = @dispatchCall [noFree] [<cast>]] <nobj>::<nMsg>::<event>;

/* Using conditional code */
@if <cond>
@if defined(<item>)
#ifdef <cond>
#endif <cond>

/* Creating Goc macros */
#define <mname> <macro>

```

5.4.1 Defining a New Class or Subclass

@class, @classdecl, @endc, @default, @uses

You can create new classes in GEOS by using the Goc keywords **@class** and **@endc**. These frame the class definition as shown in Code Display 5-6; the **@endc** keyword takes no parameters, but **@class** takes the following parameters:

```
@class <cname>, <super> [, master [, variant]];
```

cname This is the name of the new class.

super This is the class name of the superclass.

- master** When included, this word makes the new class a master class.
- variant** When included, this word makes the new class a variant class. All variant classes must also be declared master classes.

Every class must have a class structure (**ClassStruct**) in memory. This is created and filled automatically by Goc and the kernel; however, you must use the **@classdecl** keyword to make sure the structure gets built. Only one **@classdecl** statement may be used for each class, however—Goc will give an error if the class is declared twice. This is also shown in Code Display 5-6, and its parameters are as follows:

5.4

```
@classdecl <cname> [, <cflags>];
```

cname This is the name of the class being declared.

cflags These are optional flags, described below.

The optional flags that can be used with a class declaration determine how objects of the class get shut down (see “Class_flags” on page 169). The flags you can use with **@classdecl** are

neverSaved This flag indicates that objects of this class will neither be written to a state file nor be loaded in from a resource. This flag should only be used for classes whose objects will only be created at run-time (not declared in the **.goc** file) and for process classes.

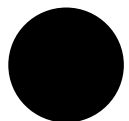
discardOnSave

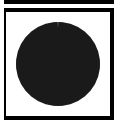
This flag applies only to master classes. Objects of this class will never be saved to a state file and must be re-initialized each time they are loaded. If you want an object simply to revert to its default configuration each time it is loaded, use the flag *ignoreDirty* instead in the object's declaration (see **@object**, below).

In addition, any variant class can have a default superclass. The variant is resolved at compile-time to have the default superclass as its superclass. To set a default superclass for a variant class, add a line with the following structure in the class definition:

```
@default <varRoot> = <super>;
```

varRoot The name of the variant class with “Class” removed. (For example, GenClass would be specified as “Gen.”)





@uses directive

If you do not create variant classes, you will never need to use the @uses directive.

5.4

super The name of the superclass to set as the default.

Sometimes a variant class will know that it will be the subclass of a specific class, though it doesn't know (at compile time) just how that ancestry will be traced. You can use the **@uses** directive to let the compiler know this; that way, the variant class can define handlers for the "used" class. For example, if you know that variant class **MyVariantClass** will always be resolved as a descendant of **MyAncestorClass**, you can put the directive

```
@uses MyAncestorClass;
```

in the definition of **MyVariantClass**. The general format is

```
@uses <class>;
```

class The class which will always be an ancestor to this class.

Code Display 5-6 Defining Classes

```
/* The @class keyword defines a new class or subclass. @endc ends the class
 * definition, and @classdecl must be put somewhere in the code to make sure Glue
 * will link the class structure into the compiled geode.*/

@class MyNewClass, VisClass;
    /* Message declarations would go here. See @message.
     * Instance data field declarations would go here. See @instance
     * and @vardata. */
@endc
@classdecl MyNewClass, neverSaved;

@class MyTriggerClass, GenTriggerClass;
    /* New messages for this subclass are defined here. */
    /* New instance data fields for this subclass are defined here. */
@endc
@classdecl MyTriggerClass;

/* When defining a variant class (which must also be a master class), you can
 * set a superclass for the variant at compile-time using @default. */

@class MyNewVariantClass, MetaClass, master, variant;
    @default MyNewVariant = VisClass;
@endc MyNewVariantClass
@classdecl MyNewVariantClass;
```

5.4.1.1 Defining New Messages for a Class

```
@message, @stack, @reserveMessages, @exportMessages,  
@importMessage, @alias, @prototype
```

As discussed in section 5.3.3 on page 180, messages are simply 16-bit numbers allocated as an enumerated type. When a new class is defined, a constant is automatically created representing the first message number for the class. This constant is then used as the first number in the enumeration of messages.

5.4

The constant is built off the class' superclass. **MetaClass** has the first 16384 messages reserved for its use. Each master level gets 8192, and the first master class of a level gets 2048 of these. All other classes are allocated 512 message spots. Thus, a master class subclassed directly off **MetaClass** would have 2048 messages beginning with number #16384 (since the numbering is zero-based). A subclass of this would have 512 messages beginning with number #18432.

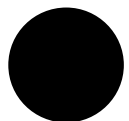
This numbering scheme ensures that no two classes at different levels in the class hierarchy will have the same message number. Specifically, a class will never have the same message number as one of its sub- or superclasses.

New messages must be defined in the class definition between the **@class** and **@endc** keywords (see above). They are defined much like normal function calls and follow the normal C calling conventions (see Code Display 5-7 for examples). If your class uses messages from its superclass, you do *not* have to declare these messages in your class definition—they are inherited automatically. This is true even if you are subclassing the method to alter its functionality.

To define a new message, use the **@message** keyword. This keyword takes the following parameters:

```
@message    <retType> <mname>(<param>*) ;
```

retType This is the data type of the return value of the message. It can be any standard C or GEOS data type (excluding structures), or a pointer to a structure. If this message has multiple return values, you must do as in C function calls and pass pointers to buffers for the return information.



mname	This is the name of the message. By convention, it will be MSG_ followed by a shortened version of the name of the class and then some useful name (e.g., MSG_META_INITIALIZE).
param	This represents one or more parameters. Messages may have no parameters, one parameter, or several parameters. Parameter definition is essentially the same as definition of function parameters; see Code Display 5-7 for examples.

5.4

NOTE: When defining a function with no parameters, it is best to declare it with “void” between the parentheses. This will make sure Goc gives an error if the function is called with arguments.

Messages for Use with Assembly Code

The **@stack** keyword indicates that parameters are passed on the stack; it is important to note that because of calling conventions, parameters passed on the stack must be listed in the message definition *in reverse order* from the way the handler pops them from the stack. This keyword is used only when the message may be handled by an assembly language method; its format is shown below:

```
@message <retType> <mname>(@stack <param>*);
```

All the parameters shown in the formats are the same as in the normal **@message** format.

Code Display 5-7 Defining Messages

```
/* Each message is defined for a class within the class definition. */
@class MyTriggerClass, GenTriggerClass;
/* All the new messages MyTriggerClass can handle are defined here. */
@message void MSG_MYTRIG_SET_COLOR(colors colorIndex);
@message optr MSG_MYTRIG_RETURN_OPTR( void );
@message void MSG_MYTRIG_COLLECT_PARAMS(byte bParam, word wParam, char * string);
/* Instance data fields would be defined here. */

@endc
@classdecl MyTriggerClass;
```

Exporting, Importing, and Reserving Message Ranges

As discussed above, message numbers are assigned based on the class' location in the class tree. No message number will ever conflict with messages defined in the class' superclasses. However, the assignment scheme opens up the possibility that classes on the same level in the class tree could have conflicting message numbers (see Figure 5-7).

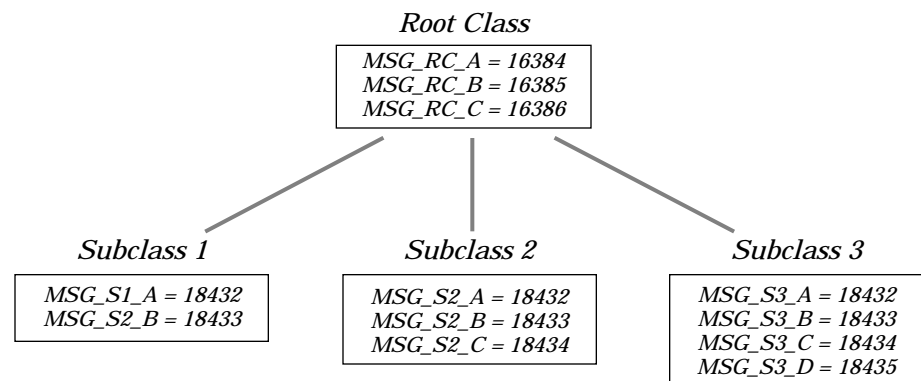
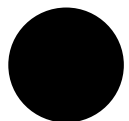


Figure 5-16 *Conflicting Message Numbers*

Although no class' message numbers will ever conflict with its superclasses', it is possible, and even likely, for classes of the same level to have conflicting message numbers. For example, MSG_S2_C and MSG_S3_C both have the message number:

Normally, this is not a problem. If subclasses are to receive the same message and handle it differently, the message can typically be defined in the superclass and simply be intercepted by the subclasses. Sometimes, however, different subclasses will need to have different definitions for the same messages. For example, a class supplied by a library may be used by several applications; if the applications each create a subclass, these subclasses can import particular messages that will be the same for all the subclasses in all the applications.

Goc therefore allows a class to export a range of message numbers which subclasses can import and create specific definitions for. This allows you greater control over what aspects of the class you can define.



To export a range of messages, use the **@exportMessages** keyword. This will set aside several message numbers which can then be imported by subclasses using the **@importMessage** keyword.

Another potential problem is upgrading your program from release to release. If you create classes that may grow in the future, you may want to reserve a number of message spots to ensure that those spots can be filled in later. Nothing is done with the spots; they are simply place holders for future upgrades. You can use the **@reserveMessages** keyword to reserve a range of any size. The parameters of these three keywords are shown below:

```
@reserveMessages    <num>;  
@exportMessages    <expName>, <num>;  
@importMessage     <expName>, <messageDef>;
```

num This is the number of messages in the exported range.

expName This is the name of the exported range. This is used when importing messages to ensure that the proper numbers are used.

messageDef This is a standard message definition line, the same as would be found with the **@message** keyword (though **@message** is left out).

Note that you do not *need* to reserve messages for upgrades; any class can always have messages tacked on to the end of its class definition. If you want to group the messages logically, however, you should reserve ranges where you expect additions to be made.

Aliasing Messages

The **@alias** keyword allows a single message to have more than one pass/return format. The **@prototype** keyword allows quick, clean, and convenient repetition of a single format for many different messages; it also allows a class to create a prototype so users of a message can have their own messages with the same format.

@alias is used when a single method takes conditional parameters. For example, a method may take a word value in a certain case and a dword value in another (dependent upon a passed flag). Each condition must be accounted for in its own message format. Rather than create a message and a method

for each case, you can create a single assembly-language method for all the different pass/return formats; then, you can use **@alias** to make several messages refer to the same method, each using a different format.

```
@alias(<protoMsg>) <msgDef>;
```

protoMsg The name of the original message. The new message may have different pass/return values but will invoke the same method code and will have the same message number.

msgDef The new message definition. It follows the same format as messages defined with the **@message** keyword (with **@message** left off).

5.4

In addition, if you have a single pass/return format for many messages, you can use the **@prototype** keyword as coding shorthand. For example, if an object has ten messages that all take two parameters and return a single value, you can set up the format with the **@prototype** keyword and then use a simpler format for definition of your messages. An example is shown in Code Display 5-8, and the parameters of this keyword are shown below.

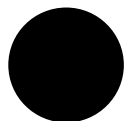
```
@prototype <msgDef>;
```

msgDef This is the standard message definition. For the message name, use something like “MY_PROTOTYPE” that you can insert later into the definitions of your real messages. All other parts of the message definition are the same as would be declared with **@message** (with **@message** left off).

Code Display 5-8 Aliasing Messages

```
/* MyClass in this example uses both prototype and aliased messages. */
@class MyClass, MetaClass;
/* The following is a normal message declaration. The register that this
 * parameter uses is specified because the handler (method) is written in
 * assembly language. */
@message void MSG_MESSAGE_WITH_WORD(byte flag = cl, word value = dx);

/* The following message invokes the same method as the alias above.
 * It has the same message number but passes a different sized parameter
 * in different registers. */
@alias(MSG_MESSAGE_WITH_WORD) void MSG_MESSAGE_WITH_DWORD(byte flag = cl,
                                                             dword value = dx:bp);
```



```
/* The following message is not used. Its pass and return values can
 * be used elsewhere, however, to ensure that all handlers of this message
 * type are given the same format. */
@prototype int MSG_MYCLASS_PROTO(int a, int b);

/* The following have the same return values and parameters
 * as the prototype above. */
@message(MSG_MYCLASS_PROTO) MSG_MY_CLASS_ADD;
@message(MSG_MYCLASS_PROTO) MSG_MY_CLASS_SUBTRACT;
@message(MSG_MYCLASS_PROTO) MSG_MY_CLASS_MULTIPLY;

5.4 @endc
@classdecl MyClass;
```

5.4.1.2 Defining Instance Data Fields

@instance, @composite, @link, @visMoniker, @kbdAccelerator, @activeList

Instance data fields are all defined with the **@instance** keyword. Other keywords may be included in the **@instance** declaration for special types of data. All instance data definitions must appear between the class' **@class** and **@endc** keywords (see above under class definition).

The **@instance** keyword is used to define normal instance data. If you have data that must be added or removed dynamically (such as hints), use the **@vardata** keyword, described in section 5.4.1.4 on page 195. Also, if you have data that requires relocation (such as pointers to fixed data) when the object is loaded, use the **@reloc** keyword.

The format of the **@instance** keyword is as follows:

```
@instance    <insType>    <iname> = <default>;
```

insType	A standard C or GEOS data or structure type representing the data type of the instance field.
iname	The name of the instance field.
default	The default value of the instance field if it is not filled in when an object of this class is instantiated. The value must, of course, be appropriate for the data type.

Goc has several special types of instance data fields that you can declare along with **@instance** to make object definition easier. The format for using one of the special types is shown below (with examples in Code Display 5-9). Each of the types is also described below.

```
@instance    <specType> <iname>;
```

specType This is the keyword (one of those shown in the list below) that defines the special type of this field.

iname This is the name of the instance field.

5.4

The special types are given here:

@composite This field is used when objects of the class being defined are allowed to have children. The **@composite** field will actually contain an **optr** to the first child object in an object tree. Since most objects in object trees are subclassed from **VisClass** or **GenClass**, you will most likely never use the **@composite** keyword. Both **VisCompClass** and **GenClass** have **@composite** fields predefined. The **@composite** type has a special format, shown below:

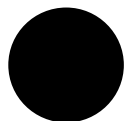
```
@instance  @composite <iname> = <linkName>;
```

where *iname* is the name of the instance field and *linkName* is the name of the field designated as **@link** (below). Note that there must be a **@link** field in every class that has a **@composite** field. See section 5.4.6.4 on page 232 for more information on object trees and the composite and link fields.

@link This field is used by objects that can be children in an object tree. Where the **@composite** field points to the first child, the **@link** field points to the next sibling. If there is no next sibling, this field will point back to the parent object. Since most objects in object trees are subclassed from **VisClass** or **GenClass**, you will most likely never use the **@link** keyword. Both **VisClass** and **GenClass** have **@link** fields predefined.

@visMoniker

This field is designated as holding a pointer to a visual moniker chunk for the object. It is used in **GenClass**—see “GenClass,” Chapter 2 of the Objects Book for information on the *GL_visMoniker* field. The moniker or moniker list must be in the same resource as the generic object using that moniker since



only the chunk's handle is stored. A moniker list can store full object pointers to its monikers, so monikers referenced by a list need not be in the same resource as that list; thus if an object's moniker is specified via a list, then while the list must be in the same resource as the object, the monikers themselves need not be.

@kbdAccelerator

5.4

This field contains a character sequence that, when typed by the user, causes the object to execute its default operation. For example, a keyboard accelerator could invoke a trigger implemented as a menu item. It is used in **GenClass** only.

Code Display 5-9 Declaring Instance Data Fields

```
/* GenClass is a good example of many of the different types of fields. */
@class GenClass, VisClass, master, variant;

    /* The GenClass messages are defined here. */

    @instance @link GI_link;
    @instance @composite GI_comp = GI_link;
    @instance @visMoniker GI_visMoniker;
    @instance @kbdAccelerator GI_kbdAccelerator;
    @instance byte GI_attrs = 0;
    @instance byte GI_states = (GS_USABLE|GS_ENABLED);

    /* Hints and other variable data fields are defined with @vardata. */

@endc
```

5.4.1.3 New Defaults for Subclassed Instance Data Fields

@default

Recall that when defining an instance data field you can set up a default value for that field. When creating a subclass, you may wish to specify that the subclass should have a different default value for a given field than the superclass does. Use the **@default** keyword to do this:

```
@default <iname> = <default>;
```

iname	The name of the instance field.
default	The new default value of the instance field if it is not filled in when an object of this class is instantiated. The value must, of course, be appropriate for the data type. You may use @default as part of this value; this @default will be treated as the value of the superclass. (If this seems confusing, try looking at the example.)

For example, a subclass of GenInteraction could set GIV_DIALOG as its default value for the GenInteraction instance field *GII_visibility*:

5.4

```
@default GII_visibility = GIV_DIALOG;
```

A generic class might want to have the same value for its *GI_states* field as its superclass, except with the GS_USABLE flag turned off:

```
@default GI_states = @default & ~GS_USABLE;
```

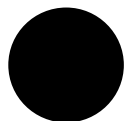
5.4.1.4 Defining and Working With Variable Data Fields

```
@vardata, @vardataAlias, ObjVarAddData(),
ObjVarDeleteData(), ObjVarDeleteDataAt(), ObjVarScanData(),
ObjVarFindData(), ObjVarDerefData(),
ObjVarDeleteDataRange(), ObjVarCopyDataRange(),
MSG_META_ADD_VAR_DATA, MSG_META_DELETE_VAR_DATA,
MSG_META_INITIALIZE_VAR_DATA, MSG_META_GET_VAR_DATA
```

Most classes will have well-defined instance data fields; each object in the class will have the same data structures, and all the instance chunks will look relatively similar and will be the same size.

Many classes, however, will also use “variable data,” or instance fields that may be added or removed dynamically. This allows objects within the same class to have more or less instance data than other objects in the class. One example of variable data is the use of hints in generic UI objects. Because each object in a given class may or may not have hints, the objects can actually have different instance sizes. Variable data instance fields are defined with the use of the **@vardata** keyword.

Using variable data, however, is somewhat more complex than using standard instance data. You must use special kernel routines or messages to get a pointer to the data; then you can use the pointer to access the field.



Variable data is stored together at the end of the instance chunk in “data entries.” Each entry consists of a primary word and optional extra data. The primary word represents a data type defined by the keyword **@vardata**. This type is created automatically by Goc when the **@vardata** keyword is used.

5.4

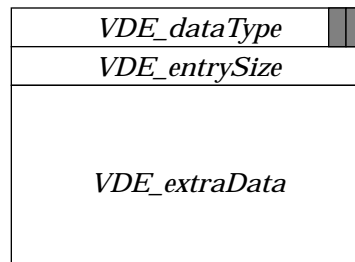


Figure 5-17 Variable Data Storage

The VarDataEntry structure contains a single variable data entry. If it has no “extra data,” the VDE_entrySize and VDE_extraData fields will not exist in the entry. Otherwise, VDE_entrySize will contain the entire size of the entry, and VDE_extraData will contain the actual data of the type found in VDE_dataType. The two shaded boxes in VDE_dataType represent the two least significant bits of that field; they are used to store two flags. Note that this structure is shown for illustration only; application programmers will never have to access it directly.

Each data entry is associated with a master class level and is considered part of the instance data for that master level (despite being stored at the end of the instance chunk). Thus, when a master part of an object is destroyed, the variable data entries associated with that master class will also be destroyed. For example, when a UI object is set not usable (taken off the screen), its Vis master part is removed from the instance chunk; any variable data entries associated with **VisClass** will also be destroyed.

Variable data may also be tagged for saving to the state file. That is, you can set up individual data entries to be saved to a state file and to be reinstated when the object is loaded from the state file. For more information about state saving, see section 5.4.6.6 on page 237.

Variable data may be defined in an object’s declaration in your **.goc** file or may be added and removed during execution. This gives the potential for

using variable data as temporary storage in an object's instance chunk; however, temporary data used in this manner should be kept small to avoid slowing down the system—constantly resizing instance chunks to add and remove vardata fields makes more work for the memory manager.

To define a variable data type in a given class, use the **@vardata** keyword as follows (an example is given in Code Display 5-11):

```
@vardata    <type> <vname>;
```

5.4

type This is the data type of any extra data associated with the variable data. It must be a standard C or GEOS data type. If the type *void* is specified, no extra data will be added to the data entry when it is created. (An instance data field may be declared as an array, just as in standard C.)

vname This is the name of the variable data type. This name is used whenever referring to the vardata entry. Note that no two variable data types should have the same name, even if they're in different classes. Doing so will cause a compilation error. It's a good practice to put the class name within the data type name.

Code Display 5-10 Examples of Instance Data Declarations

```
/* These are some data fields for MyDataClass.
 */

@instance    ChunkHandle    MDI_aChunk;
@instance    HelloInfoFlags MDI_flags;
@instance    byte           MDI_lotsOfNumbers[32];
```

Some vardata types may have varying amounts of extra data. For example, one type may have either a word or a dword of extra data. To allow this, you can set up an alias with the new type attached using the keyword

@vardataAlias:

```
@vardataAlias (<origName>) <newType> <newName>;
```

origName This is the name of the original variable data field already defined with **@vardata**.

- newType** This is the data type of the new variable data field, a standard C or GEOS data type.
- newName** This is the name of the new variable data field. In essence, the original and new fields will have the same data type word but will have different extra data size.

As noted earlier and as shown in Figure 5-17 on page ● 196, the data type field in the data entry has two flags associated with it. These flags are each one bit:

5.4

VDF_EXTRA_DATA

This flag indicates that this data type carries extra data.

VDF_SAVE_TO_STATE

This flag indicates that this particular data entry should be saved along with all the other object's instance data when the state is saved. It should likewise be restored when the object is restored from the state file. Unless set *off* explicitly, this flag will be set for every data type defined in a **.goc** or **.goh** file.

The bitmask VDF_TYPE is a bitwise OR of VDF_EXTRA_DATA and VDF_SAVE_TO_STATE. You can use it to mask out all but those bits.

Code Display 5-11 Defining Variable Data

```
/* Hints are defined with the @vardata command, as is shown in GenClass. Only a
 * small portion of the hints for GenClass are shown here. Those with structures
 * or data types (not "void") have extra data fields associated with them. */

@class GenClass, VisClass, master, variant;
    /* Messages are defined here. */
    /* Followed by instance data defined with @instance. */
    @vardata void HINT_CENTER_MONIKER;
    @vardata SpecSizeSpec HINT_CUSTOM_CHILD_SPACING;
    @vardata char[] ATTR_GEN_INIT_FILE_KEY;

    /* Relocatable instance fields (see the next section) are defined with
     * @reloc. This field contains an object pointer that must be resolved
     * when the GenClass object is loaded. */
    @instance @link GI_link;
```

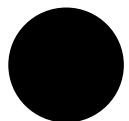
```
@reloc GI_link, optr;  
@vardata DestinationClassArgs ATTR_GEN_DESTINATION_CLASS;  
@reloc ATTR_GEN_DESTINATION_CLASS, 0, optr;  
@endc
```

The kernel provides a number of routines an object may use to add, remove, and change its own vardata entries. Note that all these routines must be called from within the object containing the variable data entries; since variable data is instance data, it is against OOP doctrine for one object to alter another object's variable data directly.

5.4

Instead, **MetaClass** provides vardata messages that can be sent by one object to another to add, remove, change, or retrieve vardata entries of another object. The kernel routines and **MetaClass** messages are outlined below:

- ◆ **ObjVarAddData()**
This routine adds an new entry for the passed data type or replaces the extra data associated with a particular data type entry.
- ◆ **ObjVarDeleteData()**
This routine deletes the entry for a particular data type when passed the data type.
- ◆ **ObjVarDeleteDataAt()**
This routine deletes a particular data entry when passed the entry's pointer as returned by **ObjVarAddData()**, **ObjVarFindData()**, or **ObjVarDerefData()**.
- ◆ **ObjVarScanData()**
This routine causes the kernel to scan all data entries in an object's variable data and call any "handler routines" listed for them. This process is described below.
- ◆ **ObjVarFindData()**
This routine searches for and returns (if possible) a pointer to a data entry of the passed data type.
- ◆ **ObjVarDerefData()**
This routine returns a pointer to a data entry when passed the object's optr and the data type. If the entry does not exist, this routine will call on the object to create and initialize the entry. Such variable data then behaves much like instance data. The object containing the vardata is



responsible for creating the entry and then initializing it upon receipt of a `MSG_META_INITIALIZE_VAR_DATA`, described below.

◆ **ObjVarDeleteDataRange()**

This routine deletes all data entries with types in the passed range.

◆ **ObjVarCopyDataRange()**

This routine copies all data entries within the passed range from one object's instance chunk to another's. If any entries are copied, the destination object will be marked dirty for saving. This routine must be called by the destination object; it is bad policy for one object to alter another object's instance data. This routine is primarily for copying hints from one UI object to another and is not commonly used by applications.

5.4

The four messages (in **MetaClass**) that can be used to add, delete, and alter variable data entries remotely are listed below. Classes will never need to intercept and subclass these messages because the proper functionality is implemented in **MetaClass**.

`MSG_META_ADD_VAR_DATA`

Adds a new vardata type to the recipient object. If the type already exists, the passed type replaces the old one.

`MSG_META_DELETE_VAR_DATA`

Deletes a vardata type from the recipient's instance data. If the type does not exist, nothing is done.

`MSG_META_INITIALIZE_VAR_DATA`

Used when something is trying to access an object's vardata field remotely but the field has not yet been added to the object or initialized. The object must create and/or initialize the vardata field at this point.

`MSG_META_GET_VAR_DATA`

Returns the extra data set for the passed data type.

In addition to supporting variable data structures, GEOS allows you to set up "handlers" for different variable data types. Handlers are routines that process a given data entry; for example, each generic UI object stores a number of hints. Specific UI classes, when attached to the generic object, have a specific routine to handle each hint supported. Some specific UIs do nothing with certain hints; these specific UIs do not have handlers for those hints.

Handlers are associated with data types through the use of a **VarDataCHandler** table. This is a table that you set up in your **.goc** file that contains pairings of routine names with **@vardata** field names. An example of the **VarDataCHandler** table is shown in Code Display 5-12.

A handler is simply a normal C routine or function and is defined as such. The handler should be declared as an **_pascal** routine. The table pairs the handler with the **@vardata** data type, and when **ObjVarScanData()** is called, all handlers for all data types are called in order. This is true for the object's class and all its superclasses since variable data is inherited just as normal instance data is. The handler can do almost anything appropriate with the exception of destroying the object or adding or deleting variable data from the object.

5.4

Code Display 5-12 Variable Data Handlers

```

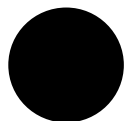
/* This example is taken from the C VarData sample application. */

/* This is a VarDataCHandler. It is called by the ObjVarScanData() routine when
 * the data type corresponding to this routine in the VarDataCHandlerTable
 * is encountered. The parameters indicated are passed.
 *
 * This particular handler is actually used for several different data types
 * (see VarDataCHandlerTable below). The data type can be distinguished by
 * the 'dataType' parameter.
 *
 * NOTE: VarDataInteractionHintHandler, like any handler used in a
 * VarDataCHandler structure, must be declared _pascal. */

void _pascal VarDataInteractionHintHandler(MemHandle mh, ChunkHandle chnk,
void *data, word dataType, HandlerData *handlerData) {

    if (dataType == HINT_ORIENT_CHILDREN_HORIZONTALLY) {
        handlerData->HD_flags.has_horiz = 1;
    } else if (dataType == HINT_ORIENT_CHILDREN_VERTICALLY) {
        handlerData->HD_flags.has_vert = 1;
    } else if (dataType == HINT_ALLOW_CHILDREN_TO_WRAP) {
        handlerData->HD_flags.has_allow_wrap = 1;
    } else if ((dataType == HINT_WRAP_AFTER_CHILD_COUNT) &&
        (((WrapAfterChildCountData *) data)->WACCE_childCount == 2)) {
        handlerData->HD_flags.has_wrap_after = 1;
    }
}

```



```

/* This is the VarDataCHandler Table. It consists of data type/VarDataCHandler
 * pairs. The VarDataCHandlers are far routines. */

static VarDataCHandler varDataInteractionHandlerTable[] = {
    {HINT_ORIENT_CHILDREN_HORIZONTALLY, VarDataInteractionHintHandler},
    {HINT_ORIENT_CHILDREN_VERTICALLY, VarDataInteractionHintHandler},
    {HINT_ALLOW_CHILDREN_TO_WRAP, VarDataInteractionHintHandler},
    {HINT_WRAP_AFTER_CHILD_COUNT, VarDataInteractionHintHandler}
};

```

5.4.1.5 Defining Relocatable Data

@reloc

Some objects and classes may have instance data fields that must be resolved when the object is loaded and linked at run-time. For example, if the object contains an *optr* to another object, that *optr* must be updated when the object is loaded and resolved since the global memory handle can't be known at compile-time.

For some special instance fields, this happens automatically. For example, the **@composite** and **@link** fields as well as *optrs* are automatically resolved. However, if you add your own instance fields requiring relocation, you will have to set them up with the **@reloc** keyword. This is true for both static and variable data.

This keyword uses two formats. The first listed here is for normal, static instance data, and the second is used with variable data.

```
@reloc    <iname>, [(<count>, <struct>)] <ptrType>;
```

iname	This is the name of the relocatable instance field.
count	If the instance variable is an array of relocatable data or structures containing relocatable fields, this is the number of elements in the array.
struct	If the relocatable data is an array of structures, this represents the name of the field within each structure that requires relocation.
ptrType	This is the type of relocatable data contained in the field. It may be one of <i>optr</i> , <i>ptr</i> , or <i>handle</i> .

```
@reloc <vname>, <fn>, [( <count>, <struct> )] ptrType;
```

vname	This is the name of the variable data type.
fn	This is the name of the field within the variable data's extra data. If there is no extra data with this data type, put a zero rather than a name.
count	If the instance variable is an array of relocatable data or structures containing relocatable fields, this is the number of elements in the array.
struct	If the relocatable data is an array of structures, this represents the name of the field within each structure that requires relocation.
ptrType	This is the type of relocatable data contained in the field. It may be one of <i>optr</i> , <i>ptr</i> , or <i>handle</i> .

5.4

5.4.2 Non-relocatable Data

```
@noreloc
```

To force an instance data field which would normally be relocatable (e.g., an *optr*) to not be relocatable, use the **@noreloc** keyword. Use this keyword together with the name of the field to be marked non-relocatable directly after defining the instance field itself as shown in Code Display 5-13.

Code Display 5-13 Use of the @noreloc Keyword

```
@instance optr MCI_ruler;          /* Normally MCI_ruler would be reloc... */
@noreloc MCI_ruler;               /* ...but now it isn't. */
```



5.4.3 Defining Methods

@method, @extern

Methods are the routines executed when an object receives a message. Each class understands a certain set of messages; each of these has a place in the class' method table and corresponds to one method.

5.4

Although methods are class-specific, they are not defined between the **@class** and **@endc** of class definition. Instead, their declaration line links them to a single class and to a specific message. Goc, Glue, and GEOS build each class' method table automatically; you do not have to create the table yourself.

To define a method, use the **@method** keyword. This has the following structure:

```
@method    [ <hname>, ] <cname>, <mname>;
```

hname The handler name, if any. If you wish to use the method as a function, it must have a handler name. If you do not provide a handler name, Goc will create one for you. This name is useful for setting breakpoints when debugging. If you do not provide a name, Goc constructs the name by concatenating the class name (with the -Class suffix, if any, removed) with the message name (with the MSG_ - prefix, if any, removed). For example, Goc would call **MyClass'** handler for MSG_DO_SOMETHING "MyDO_SOMETHING".

cname The name of the class to which this method belongs. Each method may belong to only one class.

mname The name of the message that invokes this method. The plus symbol indicates that one method may be invoked by more than one message as long as they all have the *same* parameters and return values. At least one message must be specified.

Note that the name of the method (the handler name) is optional. Parameters and return values are not put in the method declaration—these are defined with **@message** as discussed in section 5.3.3 on page 180.

If you will wish to call the method as a routine occasionally, your compiler will probably require that you provide a prototype for the routine. If your **@message** declaration looks like

```
@message word MSG_MC_DO_SOMETHING(word thing);
```

and your **@method** declaration looks like

```
@method DoSomething, MyClass, MSG_MC_DO_SOMETHING {
    /* Code Here */ }
```

Then your prototype should look like

```
extern word _pascal DoSomething(optr oself,
                                MyMessages message,
                                word thing);
```

5.4

The name of the type **MyMessages** is constructed automatically by taking the name of the class, removing the “Class” suffix, and replacing said suffix with “Messages”.

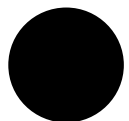
Normally, all of a class’ methods will be coded in the same code file in which the class is declared with **@classdecl**. If, however, you find you need to declare a class’ methods in a different file, you can use the **@extern** keyword to indicate a method is defined and/or used in a different object file. Goc will give no error if **@extern** is used and the method exists nowhere; Glue, however, will give a linking error in such a case. There is no such requirement, however, if you are putting only the class definition (the definitions between **@class** and **@endc**) in a different file. In this case, you can put the class definition in a **.goh** header file and the method code in the same **.goc** file as the **@classdecl** statement; you must **@include** the **.goh** file, but you won’t need to use the **@extern** directive (as long as the method code is in the same file as the **@classdecl** directive).

The format for using **@extern** is as follows:

```
/* In the file in which the class is declared with
 * @classdecl:
 */
    @extern method <cname>, <mname>;

/* In the file containing the method code: */
    @extern method <cname>, <mname>+ {
        ...method code goes here...
    }
```

cname The name of the class for which the method is defined.



mname The name of the message which invokes the method. Note that external method definitions, like normal method definitions, can handle more than one message.

Three parameters are passed automatically with messages and do not have to be declared in the **@message** definition. They are important to know when writing methods, however, because they can greatly simplify your code. These are standard parameters for all classes except **ProcessClass** and its subclasses below:

5.4

pself A far pointer to the object's instance data. *pself* points to the master group for the class for which the handler is defined. Note that this pointer may be invalidated by message calls, variable data operations, or LMem allocations.

oself An optr to the object's instance data. It contains the global memory handle and chunk handle of the instance chunk. This can be used for routines that act on the object's instance data.

message The message number of the message being handled.

As mentioned, **ProcessClass** is a special type of class. It has no true instance data because it uses the standard PC structure of an application (idata, udata, etc.). It only has one standard parameter to each of its methods: the message that was sent to it. This is because the "instance data" of **ProcessClass** includes all the global variables of your program. Because they are accessed automatically, no *oself* or *pself* is required.

Code Display 5-14 A Class Definition

```
/* The class ValClass defines four messages that invoke four different methods. The
 * entire class is shown in this example; it will function properly if coded this
 * way. Note that the methods have the class name in their declaration line
 * and thus do not appear within the class definition. */

@class ValClass, MetaClass;

@instance int value;      /* instance data value: an uninitialized integer */
```

```

/* message declarations
 * All four messages will be handled by this class. They return
 * the types shown and take the parameters defined. */
@interface ValClass {
@message int MSG_VAL_GET_VALUE();
@message void MSG_VAL_SET_VALUE(int newValue);
@message void MSG_VAL_NEGATE_VALUE();
@message Boolean MSG_VAL_IS_VALUE_BIGGER_THAN(int newValue);
@endc
@classdecl ValClass; /* the class structure must be put in memory */

/* Method Declarations
 * Each of the four methods is a single line of code. Note that the
 * parameters are automatically defined in the message definition and do
 * not need to be restated in the method definition. The same is true of
 * the return type. Note also that the class and message names appear in
 * the @method line. */

@implementation ValClass {
@method ValGetValue, MyClass, MSG_VAL_GET_VALUE {
    return(pself->value);
}

@method ValSetValue, MyClass, MSG_VAL_SET_VALUE {
    pself->value = newValue;
}

@method ValNegateVal, MyClass, MSG_VAL_NEGATE_VALUE {
    pself->value *= -1;
}

@method ValClass, MSG_VAL_IS_VALUE_BIGGER_THAN {
    /* This handler's name will automatically be created to be
     * ValVAL_IS_VALUE_BIGGER_THAN. You can use this name as a
     * C function call from within the same thread. */
    return(pself->value > newValue);
}
}

```

5.4

You may sometimes wish to call a method with normal C call-and-return conventions, rather than by sending a message. To do so, you will have to declare the method as a routine as well as a method. The declaration should have the following format:



```
extern <type> _pascal <MethodName>(
    optr                                oself,
    <TruncatedClassName>Messages message,
    <type1>                             <arg1>,
    <type2>                             <arg2>)
```

type This is the type returned by the method. It may be any data type.

5.4

MethodName

This is the name of the method. If you will be calling a method as a routine, you must give the method a name when you declare it (see page 204). Use the same name here.

TruncatedClassName

This is the name of the class, without the word “Class”. The type of this argument is the truncated class name followed (with no space) by the word “Messages”. Thus, for “HelloCounterClass”, the truncated class name would be “HelloCounter”, and the type of this field would be “HelloCounterMessages”.

typen, argn Use these fields to declare each of the arguments passed to the message. Be sure to use exactly the same arguments, and in the same order, as in the message declaration.

Code Display 5-15 Declaring a Method As a Routine

```
@message int MSG_HELLO_COUNTER_RECALCULATE_VALUE( \
    HelloPriority priority, \
    word randomDatum, \
    char aLetter);

extern int _pascal HelloCounterRecalculateValue(
    optr oself,
    HelloCounterMessages message,
    HelloPriority priority,
    word randomDatum,
    char aLetter);

@method HelloCounterRecalculate, HelloCounterClass, \
    MSG_HELLO_COUNTER_RECALCULATE_VALUE {

    /* method code goes here... */
```

}

5.4.4 Declaring Objects

In GEOS programs, you can instantiate objects in two ways: You can declare them in your source code with the **@object** keyword, or you can instantiate them and fill in their instance data during execution. In most cases, you will probably do the former, especially with generic UI objects.

5.4

Additionally, you can create resources and chunks with the **@start**, **@end**, **@header**, and **@chunk** keywords. GEOS libraries also need an additional set of declarations in their definition (**.goh**) files; these declarations (**@deflib** and **@endlib**) indicate that the code contained between them is part of the specified library.

5.4.4.1 Defining Library Code

```
@deflib, @endlib
```

If your geode is a library, it will likely have a number of **.goh** files. Each of these files contains some portion of the library's code and is included by applications that use the library. The library code must be delimited by the two keywords **@deflib** and **@endlib**, which have the following formats:

```
@deflib <libname>
@endlib
```

libname The permanent name of the library with the extender stripped off. For example, the UI library's name is **ui.lib**, and the format would then be

```
@deflib ui
/* library code here */
@endlib
```

Note that these two keywords are only necessary in files that define classes in the library. Files that have just code or data used in the library do not require them (though they are allowed).

5.4.4.2 Declaring Segment Resources and Chunks

```
@start, @end, @header, @chunk, @chunkArray, @elementArray,
@extern
```

5.4

There are essentially three types of resources in GEOS: code resources containing routines and methods, object blocks containing object instance chunks (and often data chunks), and data resources containing only data chunks. Code resources are created automatically, and no declaration is required for them (unless you require special segments; then you should use the *pragmas* appropriate for your C compiler).

Object blocks or other LMem resources are declared with **@start** and **@end**. You can set a special header on a local memory resource with the **@header** keyword. These are described below, and an example of declaring the resource block is given in Code Display 5-16. Note that the **@header** keyword must come between the **@start** and **@end** delimiters.

```
@start    <segname> [, <flags>];
@header   <type> [= <init>];
@end      <segname>
```

- segname** This is the name of the resource segment.
- flags** These are optional flags that determine two characteristics of the resource. If the flag *data* is set, the block will be set to a data resource—the default is an object block. If the flag *notDetachable* is set, the resource block will never be saved to a state file.
- type** This is the name of a structure type that will act as the header structure for the resource block. It must be some permutation of either **LMemBlockHeader** (for non object blocks) or **ObjLMemBlockHeader** (for object blocks).
- init** This is an initializer for the new header type. Typically, some data fields will be added on to the end of one of the standard LMem block headers. These fields may be filled in with initializer data with this optional argument.

The resource elements (objects or chunks, for example) are also declared within the **@start** and **@end** delimiters. The **@chunk** keyword declares a

data chunk and is shown below. For the **@object** keyword, see the next section.

```
@chunk    <type> <name> [= <init>];
```

type	This is the data type that will be held in the chunk.	
name	This is the name of the chunk. You may use this name as you would a variable name to reference the chunk.	
init	This is initializer data in the standard C format. If initializing a structure, make sure you put the data within curly braces.	5.4

Two other types of resource elements may also be defined, both of which are array types. The **@chunkArray** keyword defines a chunk array structure, and the **@elementArray** keyword defines an element array structure. See “Local Memory,” Chapter 16, for information on the structure and usage of chunk and element arrays. The formats for the keywords are described below:

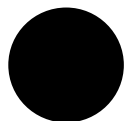
```
@chunkArray <stype> <aname> [= {<init>}];
@elementArray <stype> <aname> [ = {<init>} ];
```

stype	This is the type of structure stored in the array.
aname	This is the name of the array.
init	This is any initializer data. Initializer data is optional.

One other keyword, **@extern**, allows you to use chunks from a different compilation session. The **@extern** keyword can be used to reference remote (external) chunks, objects, and visual monikers. If the item referenced by **@extern** is not actually defined anywhere, Glue will return an error when it tries to link the item. (Note also that Glue can not link through an external item; only one layer of external linkage is allowed. Thus, one object could not have an **@extern** to a chunk that had an **@extern** to a visMoniker.) Children in object trees may not be defined with **@extern**; Goc will produce an error in this case. The format of this keyword is as follows:

```
@extern chunk <cname>;
@extern object <oname>;
@extern visMoniker <vmname>;
```

cname	This is the name of a chunk.
--------------	------------------------------



oname This is the name of an object.
vmname This is the name of a visMoniker chunk.

Code Display 5-16 Declaring Data Resources

```
/* This example declares a data block with three different chunks in it and
 * its own header type. */

5.4 typedef struct {
    LMemBlockHeader    MLMBH_meta;        /* basic header structure */
    int                MLMBH_numEntries; /* the number of entries in the block
 */
} StudentBlockHeader;

@start StudentBlock, data;                /* data flag indicates LMem block */
@header StudentBlockHeader = 1;           /* initialize new header fields */

/* The three chunks are defined below. Each represents a single field associated
 * with a single student; that is why the header's MLMBH_numEntries field contains
 * one (rather than three). */

@chunk  char    firstName[] = "John";
@chunk  char    lastName[] = "Student";
@chunk  int     grade = 6;

@end StudentBlock                        /* end of resource block */
```

5.4.4.3 Declaring an Object

@object, @default, @specificUI, gcList

The **@object** keyword is much like the **@chunk** keyword in its use. It allocates a chunk in a resource segment and fills it with initializer data. However, it also does much more in that it creates an object of a specified class and fills it with default data if no initializer data is specified.

The **@object** keyword, like **@chunk**, must be used between the delimiters **@start** and **@end** (see above). However, it must be used in an object block (it can not be used in a data resource segment). Its format is below, and several examples are shown in Code Display 5-17.

```
@object    <class> <name> = <flags>* {
    <fieldName> = <init>;*
    <varName> [= <init>];*
};
```

class The class of the object. It must be a valid class defined either by GEOS or by your own code.

name The name of the object. The object's name will be used when sending messages to it.

5.4

flags Currently only one flag is allowed for objects: *ignoreDirty*. When set, it will cause the object to ignore when changes have been made to its instance data; this means it will always be restored from a state file in the same way as it is defined in the code file. This flag should *not* be set for any generic objects.

fieldName The name of an instance data field defined for the class.

varName The name of a variable data instance field. If the variable data field has extra data, you can initialize it with the optional argument.

init This is initializer data for the instance data field. If you want default values in the field, you can either leave the field out of the **@object** declaration or use the keyword **@default** for the initializer. When declaring variable data, though, be aware that if the extra data is a structure, the initializer must be enclosed in curly braces.

GEOS supports special instance data fields as described in section 5.4.1.2 on page 192. These are declared as normal fields and are described below. Also, generic hints are implemented as variable data, so they are added to an object in the same way as other variable data types.

Object Trees

All objects declared in a static tree (e.g., your application's generic tree) should be in the same source file. If they are in different files, then they may be joined into a single tree only by dynamically adding objects from one file as children to objects of the other.



Note that if one file contains a tree of objects, then you may incorporate the whole tree by simply dynamically adding the top object in the file to the main tree. You won't have to add each object individually.

If an object declared in one source file will send a message to an object in another source file, you must include an **@extern** line in the source file containing the sending object:

```
@extern object ReceivingObjectName;
```

5.4

The message itself should be sent in the following manner (with variations possible if you will be using **@call**, passing arguments, or what have you):

```
optr ROOpPtr;  
ROOpPtr = GeodeGetOpPtrNS(@ReceivingObjectName);  
@send ROOpPtr::MSG_DO_SOMETHING(0, 0);
```

Declaring Children

If an object is to be part of an object tree, its class (or one of its superclasses) *must* have at least an **@link** instance data field as does **GenClass**. If the object is to be allowed to have children, it must also have a field of type **@composite**. These allow Goc to automatically and correctly link the tree together.

As described in section 5.4.1.2 on page 192, the **@composite** field points to the first child of an object, and the **@link** field points either to the object's next sibling or back to the parent if there is no next sibling. However, all child declaration is done completely in the composite field when using **@object**. The format is as follows:

```
<fname> = <childName> [, <childname>]* ;
```

fname This is the name of the field defined with **@composite** in the class definition.

childName This is the name of an object to be a child of this object. The star symbol indicates that one or more children may be included in the declaration line—they should be separated by commas, and each child must also be declared with an **@object** declaration.

There are many examples of this in the sample applications. Some simple examples are shown below.

```

GI_comp = @TicTacPrimary;
GI_comp = @TicTacView, @TicTacGameMenu;
VCI_comp = @TTX1, @TTX2, @TTX3, @TTX4, @TTX5, @TTO1,
           @TTO2, @TTO3, @TTO4, @TTO5;

```

Declaring Visual Monikers

For an object to have a visual moniker, it must have an instance data field of type **@visMoniker** as **GenClass** does (see section 5.4.1.2 on page 192). If you are in fact working with GenClass' GI_visMoniker field, you might want to consult its description in "GenClass," Chapter 2 of the Objects Book.

5.4

Visual monikers may take many forms, and the declaration line changes depending on which form you are using. The form ranges from a simple text field to a complex list of possible monikers based on video resolution. Each of these forms and the declaration line for it is given below.

The following form is used for simple text strings (shown with example):

```

<fname> = <string>;
GI_visMoniker = "One";

```

fname	The name of the moniker's instance data field.
string	A string of text enclosed in quotation marks. This string is the visual moniker for the object.

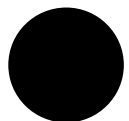
The following form is used for text strings that have a keyboard navigation character. A keyboard navigation character can be a character enclosed in single quotation marks, a numerical constant, or a text string representing some constant defined with **#define**. If it's a text string, the string is passed on for the C compiler to parse. The form is shown below with an example.

```

<fname> = <nav>, <string>;
GI_visMoniker = 'V', "View";

```

fname	The name of the moniker's instance data field.
nav	A keyboard navigation character represented as described above.
string	A text string enclosed in quotation marks.



The following form is used when a list of monikers is required. Most frequently, this form is used when defining an application's icons; one moniker for each potential screen resolution will be included in the list. The form is shown below, along with an example.

```
<fname> = list { <nameList> };  
GI_visMoniker = list {  
    @DocDirButtonSCMoniker,  
    @DocDirButtonSMMoniker,  
    @DocDirButtonSCGAMoniker  
}
```

fname This is the name of the moniker's instance data field.

nameList This is a list of resource data chunk names separated by commas. Each chunk can be defined with the **@chunk** or **@visMoniker** keyword.

It is possible when declaring a list of visual monikers to have each moniker within a chunk or to declare each moniker with the **@visMoniker** keyword. If used on its own line, this keyword takes the form

```
@visMoniker <fname> = <init>;
```

The fields are the name of the moniker (same as that specified in the moniker list) and the moniker data. The visual moniker will be put in its own chunk just as if the moniker had been declared with the **@chunk** keyword, but using **@visMoniker** often is clearer and easier.

Declaring GCN List Assignments

GEOS employs a "General Change Notification" (GCN) system which allows objects to register for notification of certain types of events. For example, some applications may want to notify for changes in the file system (file deletes, moves, copies, etc.); some objects may require notification each time the selection changes (e.g., a `PointSizeControl` object needs to know when the user changes a text selection). For further information, see "General Change Notification," Chapter 9.

While many objects will add themselves to and remove themselves from GCN lists dynamically, others will need to be placed on certain GCN lists in their

definitions. For this, the **gcnList** keyword (the only one not preceded by the marker @) is used. Its format is shown below:

```
gcnList(<manufID>, <ltype>) = <oname>,+;
```

manufID This is the manufacturer ID of the manufacturer responsible for creating the particular list type. It is used to differentiate different manufacturers who may use the same list type enumerator. In many cases, this will be the type MANUFACTURER_ID_GEOWORKS.

5.4

ltype This is the name of the GCN list. Most that you will use are defined by GenApplication and begin GAGCNLT_.... All the list types are defined in the GenApplication chapter in the Objects book.

oname This is a list of objects that should be on the GCN list. The objects are separated by commas.

Declaring Keyboard Accelerators

A keyboard accelerator acts as a “hot key,” invoking the object’s default action when pressed. The accelerator character is defined in an instance field declared with **@kbdAccelerator** as shown in section 5.4.1.2 on page 192. The form of declaration follows.

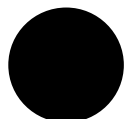
```
<fname> = [@specificUI] <mod>* <key>;
```

fname This is the name of the **@kbdAccelerator** instance data field.

@specificUI This is a Goc keyword that, when used in this declaration, allows your program to use a key combination reserved for the specific UI. This can have undefined consequences and is strongly discouraged.

mod This is a modifier key. You may put any or all of the modifiers here, separated by spaces. The four modifiers are *alt*, *control*, *ctrl*, and *shift*.

key This is either a character enclosed in single quotes (e.g., ‘k’) or a numeric key constant (e.g., C_CTRL_A or 0x41).



Using Default Values

Often an object will be declared with the default values of an instance data field. Other times, especially with generic UI object attributes, an object will have the default values with one or two modifications. In the first case, the instance data field does not need to be addressed at all; the default value will be used if no declaration is made.

5.4

In the second case, however, you must use the **@default** keyword to get the default values. If modifications are made and **@default** is not used, all default values will be lost. This is normally used only with bitfield-type attributes, and modifications are made with bitwise operators. The use of **@default** is shown below.

```
<fname> = @default [<op> [~]<attr>]*;
```

fname	This is the name of the instance data field.
op	This is a bitwise operator. If adding an attribute to the default, use the bitwise OR operator (<code> </code>); if removing an attribute, use the bitwise AND operator (<code>&</code>) with the inverse of the attribute (see below). One operator is required for each attribute added or removed. The priorities of the operators are the same as in the standard C programming language.
~	This is the bitwise operator NOT. If removing an attribute using the bitwise AND operator, you should include the NOT symbol in front of the attribute.
attr	This is the name of the attribute being added or removed.

Code Display 5-17 Declaring Objects with @object

```
/* This example shows the use of @start, @object, @visMoniker, and @end.
 * It is taken from the TicTac sample application. */

@start AppResource;
/* The AppResource resource block contains the TicTacApp
 * object and its visual moniker chunk. */

@object GenApplicationClass TicTacApp = {
    GI_visMoniker = list { @TicTacTextMoniker };
    GI_comp = @TicTacPrimary;
    gcnList(MANUFACTURER_ID_GEOWORKS,GAGCNLT_WINDOWS) = @TicTacPrimary;
}
```

```

@visMoniker TicTacTextMoniker = "TicTacToe";
@end AppResource

@start Interface;
    /* The Interface resource declares TicTac's primary window and other UI
       * gadgetry. Only the GenView from this application is shown. */
@object GenViewClass TicTacView = {
    GVI_content = @TicTacBoard;          /* A relocatable optr field */
    GVI_docBounds = {0, 0, BOARD_WIDTH, BOARD_HEIGHT};
                                          /* A Rectangle structure */
    GVI_color = { C_BLUE, 0, 0, 0 };      /* A ColorQuad structure */
    GVI_horizAttrs = @default | GVDA_NO_LARGER_THAN_CONTENT
                    | GVDA_NO_SMALLER_THAN_CONTENT
                    & ~GVDA_SCROLLABLE;
    /* The NO_LARGER and NO_SMALLER attributes are set in the
       * field, and SCROLLABLE is cleared. The SCROLLABLE attribute
       * is not set by default for the GenView; it is shown here
       * for illustration. */
    GVI_vertAttrs = @default | GVDA_NO_LARGER_THAN_CONTENT
                    | GVDA_NO_SMALLER_THAN_CONTENT;
}

@end Interface

```

5.4

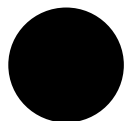
5.4.5 Sending Messages

@send, @call, @callsuper, @record, @dispatch, @dispatchcall

Often you will have to send messages to objects throughout the system. You can send messages in several ways, but the two most basic and most frequently used involve the keywords **@call** and **@send**.

If a message is being sent across threads, you must be aware of synchronization issues. If the message does not cross thread boundaries, the kernel will link the code directly as if it were a function call. (This is an implementation difference only; you do not have to specify anything different in your code.)

The **@send** keyword causes the kernel to put the specified message into the recipient's event queue. Messages sent with **@send** may not give return values and may not take pointers to locked memory as arguments. The



sender then continues executing without ever knowing whether the message was properly handled or not.

The **@call** keyword is used when the message being sent must return information to the sender. It is also used when the message must be handled immediately, before the sender is allowed to continue executing. In essence, the sender is “put to sleep” until the message has been processed, at which time the sender is woken up and may continue executing. If the message sent with **@call** is not handled (passed up the recipient’s class tree and still not handled), it will return as if it had been; no error message will be returned.

The formats for **@send** and **@call** are similar. Use them like function calls. Their format is given below:

```
@send [, <flags>]+ \
                <obj>::[{<cast2>}]<msg>(<params>*) ;
<ret> = @call [, <flags>]+ [{<cast>}] <obj>::\
                [{<cast2>}]<msg>(<params>*) ;
```

flags	This is a list of flags separated by the commas. The allowed flags are shown below.
obj	This is the name of the object to which the message will be sent. It can also be an optr variable.
msg	This is the name of the message being sent.
params	This is a list of parameters, built exactly as it would be for a standard C function call.
ret	This is a variable that will hold the return value, if any. Note that this is valid only with @call because @send does not return anything.
cast	If a message name is put here, Goc will automatically cast the return type to whatever type is returned by cast .
cast2	If a message name is put here, Goc will assume that the message is passed the same arguments as message cast2 .

The flags allowed for these keywords are listed below. They are rarely used but are available.

forceQueue This flag will cause the message to be placed in the recipient's event queue, even if it could have been handled by a direct call. Do not use this flag with **@call**.

checkDuplicate This flag makes the kernel check if a message of the same name is already in the recipient's event queue. For this flag to work, *forceQueue* must also be passed. Events are checked from first (next-to-be-processed) to last.

checkLastOnly This flag works with *checkDuplicate*, causing it to check only the last message in the event queue.

5.4

replace This flag modifies *checkDuplicate* and *checkLastOnly* by superseding the duplicate (old) event with the new one. The new event will be put in the duplicate's position in the event queue. If a duplicate is found but the *replace* flag is not passed, the duplicate will be dropped.

insertAtFront This puts the message at the front of the recipient's event queue.

canDiscardIfDesperate This flag indicates that this event may be discarded if the system is running extremely low on handles and requires more space immediately.

The **@call** command can also be used within an expression in the same way a function call could. For example, the following conditional expression is valid:

```
if (@call MyObj::MSG_MYOBJ_TEST()) {
    /* conditional code */
}
```



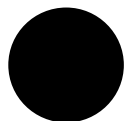
Nested Messages

messages cannot be nested on a single line.

The result of the message call will be evaluated in the if statement. Note that this may not be done with **@send** because it returns nothing.

Nested Message Calls

Because of the way Goc processes message calls, it is impossible to nest messages on a single line. For example, this call is illegal:



```
@send Obj1::MSG_THATS_PASSED_AN_INT(\
    @call Obj2::MSG_THAT_RETURNS_INT());
```

Any such call will generate a compile-time error. Instead, you should use temporary variables to break this up into several lines, e.g.:

```
int i;
i = @call Obj2::MSG_THAT_RETURNS_INT();
@send Obj2::MSG_THATS_PASSED_AN_INT(i);
```

5.4

Sending a Message to an Object's Superclass

Often you may wish to send a message directly to an object's superclass to ensure that default behavior is implemented. Use the **@callsuper** keyword with the following format:

```
@callsuper <obj>::<class>::<msg>(<pars>*) [<flgs>+];
```

- | | |
|--------------|---|
| obj | This is the object to send the message to, as in @call and @send . The object block must already be locked, and must be run by the current thread of execution. (Usually an object uses @callsuper() to send a message to itself.) |
| class | This is the class whose superclass should receive the message. |
| msg | This is the message name. |
| pars | This is the parameter list, same as @call and @send . |
| flgs | This is the flags list, same as @call and @send . |

When used on a line by itself (with no parameters or return values), **@callsuper()** passes a received message on to the superclass. This is used quite often when a subclass wants to alter existing behavior rather than replace it.

Encapsulating a Message

By encapsulating messages, you can set up events to be sent out at a later time. An encapsulated message can include the message to be sent, the object it should be sent to, and the parameters that should be passed. Using encapsulated messages can sometimes simplify coding.

Messages can be encapsulated with the **@record** keyword and later dispatched with **@dispatch** and **@dispatchcall**. (Though the use of **@record** does not necessitate a later **@dispatch**—instead, the recorded event can be passed as a parameter to another class for dispatching.) In addition, when the event is dispatched, you can override the values set in the encapsulated event to change the destination or the message. You can also cast the return value to another type if necessary. The formats of these three keywords are as follows:

```
<event> = @record <obj>::[{<cast>}]<msg>(<params>*);
```

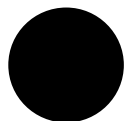
5.4

event	The handle of the recorded event, of type EventHandle .
obj	The object set to receive the message when it is dispatched. This field may be set to <i>null</i> if the destination is determined when the message is dispatched.
cast	The name of a message; if this is present, the event will have the same argument types as the specified message.
msg	The message set to be sent when the event is dispatched. This field may be set to <i>null</i> if the message is determined when it is dispatched.
params	The parameter list (same as in @call and @send) that will be sent with the dispatched message.

The **@dispatch** keyword is used to dispatch an encapsulated event to its destination. This is similar to **@send** in that it can have no return values. If the event has return values, use **@dispatchcall** (below).

```
@dispatch [noFree] \
    <nObj>::[{<cast>}]<nMsg>::<event>;
```

noFree	This flag indicates that the event's handle should not be freed after the event has been dispatched. This is useful if the same encapsulated event is to be used more than once.
nObj	This is an override destination. If the destination in the encapsulated event is null, then an object must be set here. Any object set will override that in the encapsulated message. If no override object is desired, set this to <i>null</i> .
nMsg	This is an override message. If set, this message will be sent rather than that in the encapsulated event. If no override



message is desired, set this to *null*. Any override will be sent with the same parameters as set in the encapsulated event.

event This is the handle of the encapsulated event.

The **@dispatchcall** keyword works exactly like the **@dispatch** keyword above except that it allows the use of return values. The sender will be “put to sleep” if necessary while the recipient processes the message and will be “woken up” when the message returns.

5.4

```
<ret> = @dispatchcall [noFree] [{<cast>}] <nObj>::\
                                     <nMsg>::<event>;
```

ret This is a variable that will contain the return value of the message.

other parameters

All the other parameters are the same as those in **@dispatch**.

Using Expressions with Messages

All message-sending keywords described in the previous sections—**@call**, **@send**, **@record**, **@dispatch**, and **@dispatchcall**—can take expressions in place of a destination object’s name. Additionally, the **@dispatch** and **@dispatchcall** keywords can take expressions in place of the message name. However, if an expression is used for the message, you must use a cast type to make sure Goc knows the return and parameter types. Note, however, that casts in this case use curly braces rather than parentheses.

Casting Message Call and Return Types

Goc allows you to cast a message’s pass and return values. This is best explained by example:

```
{
    int swapInt;
    char c;

    c = @call {MSG_1} object:: {MSG_2} MSG_X(swapInt);
}
```

In this case, MSG_2 takes an integer argument and MSG_1 returns a char. The casts tell Goc how MSG_X will receive parameters and return results. Goc needs the casts in those cases where MSG_X doesn’t appear explicitly

(perhaps it has been stored as an integer), and thus Goc would not be able to parse the parameters or return values.

When Goc tries to determine proper parameters and returns, it will look to the following sources when available. When trying to figure out parameters, it will look first for MSG_2, then MSG_X, and MSG_1 last. The first one Goc finds will determine the parameters.

@send and **@record** don't support return values, but on a **@call**, Goc will figure out return values by looking at MSG_1, MSG_X, and finally MSG_2.

5.4

In this case, Goc will pass to fn's method like MSG_CAST_2 but will return values as MSG_CAST_1 does:

```
Message fn = GetMessageToCall();
c = @call {MSG_CAST_1} myObj:: {MSG_CAST_2} fn(x);
```

Now we pass to MSG_B like MSG_CAST_2, but return like MSG_B:

```
c = @call myObj:: {MSG_CAST_2} MSG_B(swapInt);
```

Message Shortcuts

All messages, when received, contain three basic parameters: the message number (*message*), the optr of the recipient (*oself*), and a far pointer to the recipient's locked instance chunk (*pself*). This allows several shortcuts and shorthand formats for use within methods:

```
@callsuper;
```

When used in a method as above, the **@callsuper** keyword passes the received message up to the object's superclass. Use this whenever subclassing a message when the default functionality must be preserved.

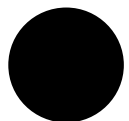
```
<ret> = @call self::<msg>(<params>*);
```

Any object can send a message to itself using **@call** and "self" as the destination. The remainder of the command is the same as a normal **@call**.

```
<ret> = @call process::<msg>(<params>*);
```

Any object can send a message to its Process object with **@call** and "process" as the destination. (The Process object is the object of class **ProcessClass**.) The remainder of the command is the same as a normal **@call**.

```
<ret> = @call application::<msg>(<params>*);
```



Any object can send a message to its Application object (of class **GenApplicationClass**) with **@call** and “application” as the destination. The remainder of the command is the same as a normal **@call**.

```
<ret> = @call @visParent::<msg>(<params>*) ;
```

Any object in a visible tree can use **@visParent** as the destination of an **@call** command. The message will be sent to the object’s parent in the visible object tree. The remainder of the command is the same as a normal **@call**.

5.4

```
<ret> = @call @genParent::<msg>(<params>*) ;
```

Any object in a generic tree can use **@genParent** as the destination of an **@call** command. The message will be sent to the object’s parent in the generic object tree. The remainder of the command is the same as a normal **@call**.

```
@send @visChildren::<msg>(<params>*) ;
```

Any composite object in a visible object tree (therefore a subclass of **VisCompClass**) can send a message that will be dispatched at once to all of its children. Any message sent with **@visChildren** as the destination must be dispatched with the **@send** keyword and therefore can have no return value.

```
@send @genChildren::<msg>(<params>*) ;
```

Any composite object in a generic object tree (therefore a subclass of **GenClass**) can send a message that will be dispatched at once to all of its children. Any message sent with **@genChildren** as the destination must be dispatched with the **@send** keyword and therefore can have no return value.

In addition to the above shortcuts, you may also pass the *optr* of an object using **@<obj>**, where *<obj>* represents the name of the object. This syntax gets translated by Goc into *(optr)&<obj>*; this is similar to using the ampersand (&) to pass a pointer.

5.4.6 Managing Objects

In addition to knowing how to declare objects and classes, you need to know how to manage objects during execution. This includes instantiating new objects, deleting objects, saving object state, and moving objects around object trees.

Both the kernel and **MetaClass** (the topmost class in any class hierarchy) have routines and methods to create, manage, and destroy objects. You will probably not have to or want to use all these routines and methods, but understanding what they do and how they work can help you understand the object system as a whole.

5.4.6.1 Creating New Objects

5.4

```
ObjDuplicateResource(), ObjInstantiate(),  
MSG_META_INITIALIZE, MSG_GEN_COPY_TREE
```

You can introduce objects to the system in four basic ways. Each of these has benefits and drawbacks, and each has an appropriate place and time for use. It is unlikely, however, that you will use all four different means.

Storing Objects in a Resource Block

This is a common and simple way to introduce objects to the system. The Hello World sample application uses this method of creating and loading objects. Resource blocks are contained in your geode's executable file (the **.geo** file) and are automatically loaded when accessed. These resources may also be tagged for saving to state files automatically.

Setting up an object resource is simply a matter of defining the resource and using the **@object** keyword to define each object in the resource. The object resource block is automatically created and put in your executable file. Each object defined with **@object** is allocated a chunk and a chunk handle within the resource block. Because both the chunk handle and the handle of the resource block are known, accessing individual objects in the resource is simple. In essence, when you set up an object resource, you don't need to worry about bringing the objects into the system or shutting them down.

Using a resource for objects has limitations. Objects loaded from resources are always loaded with the same characteristics. This can be a problem if you need to have several different copies of a certain set of objects, and each copy can be changed. In this case, you would duplicate the resource (see below) before accessing the objects within it.

For an example of objects defined by means of declaring them with the **@object** keyword within an object resource, see Code Display 5-17.

To define an object resource, you must know what objects you'll require before your geode is launched. Some complex programs will dynamically instantiate individual objects or entire trees without knowing previously what objects will be required. To do this, you'll need to use **ObjInstantiate()** (see below) for instantiating individual objects.

Duplicating an Object Block Resource

5.4

This is another common method employed by both the User Interface and applications. It provides the simplicity of the object resource model (above) while allowing you to have several different copies of the resource. Thus, it's perfect if you want to use templates for your object blocks (this is what the Document Control object does, as shown in the Tutorial application (see `APPL\TUTORIAL\MCHRT4\MCHRT.GOC`)).

First, you must set up an object resource in your code file with **@start**, **@end**, and **@object**. The objects in such a "template" resource should not be linked to any object outside the block. Generic object branches created in this manner should have their topmost object marked as not usable (`~GS_USABLE`); this is because it is illegal for a generic object to be usable without having a generic parent. Instead of accessing these objects directly, you should duplicate the resource block. (A resource can not be both duplicated and used directly.) This is done with **ObjDuplicateResource()**, which allocates a new block on the heap, sets it up properly, and copies the resource directly into it.

You are returned a handle to the new object block, which you can then modify any way you like. Because all the chunk handles of all the objects will be the same as in the source block, you can easily access any object in the duplicate. Once copied, the duplicate objects may be added to your generic tree and then set `GS_USABLE`. And, by using **ObjDuplicateResource()** more than once on the same resource, you can have several different, possibly modified versions of the resource at once.

As with using resource blocks, however, you must know the configuration of all your template objects beforehand. You may still need to add new objects to the resource or dynamically create other objects. This is the primary drawback of this method.

Additionally, if you duplicate resource blocks, you should also free them when they're not needed any more. Generic objects in the block should be set not

usable and then removed from the tree before the resource is freed. Freeing should be done by sending `MSG_META_BLOCK_FREE` to any object in the block or by calling **ObjFreeObjBlock()**. Use of the kernel routine **ObjFreeDuplicate()** is not recommended as it requires all objects in the block to be thoroughly clean of any ties to the system. (`MSG_META_BLOCK_FREE` and **ObjFreeObjBlock()** ensure that the objects have had adequate time to relinquish these ties first.)

Instantiating an Individual Object

5.4

The most complex of these three options, this can provide the flexibility needed for all cases. The actual act of instantiating an object is not difficult or very complex. However, it is time and labor intensive and requires several steps that are not necessary when using object resources. In addition, cleaning up after objects created in this manner is more complex.

To create a new object on the fly, you first must set up a place to put it. To do this, allocate a memory block on the global heap (you can instead use an existing object block, of course) and set it up with the proper flags and header to be an object block. Then, lock the chosen block on the heap with **ObjLockObjBlock()**. The block is now set up to receive the new object.

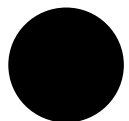
To actually create the new object, call the kernel routine **ObjInstantiate()**. This will create a chunk in the object block and zero the instance chunk. If the object is of a class with master parts, the instance chunk will remain uninitialized until the first time a class in the master group receives any message. If the class is a direct descendant of **MetaClass**, the instance chunk will be immediately initialized to default values with `MSG_META_INITIALIZE`. If you want to change or add to the default data of this type of object, subclass `MSG_META_INITIALIZE`; be sure to call the superclass first. To initialize any master group of an object, send it a classed event that will be handled by a class in that master level.

After calling **ObjInstantiate()**, unlock the object block with **MemUnlock()**. An example of instantiating a new object is shown in Code Display 5-18. Generic objects created this way may then be added to a generic tree and set usable. They may be destroyed using `MSG_GEN_DESTROY`.



MSG_META_INITIALIZE

Always call the superclass prior to handling the message.



Code Display 5-18 Instantiating an Object

```
/* This sample of code belongs to a sample GenInteraction object (the object would
 * have to be a subclass of GenInteractionClass). It does several things:
 *
 * 1. It instantiates a new GenTrigger object. The new
 *    GenTrigger will be added to the same object block
 *    containing the GenInteraction handling the message.
 *
 * 2. It adds the new GenTrigger as a child of the
 *    SampInteraction (the handling object).
5.4 *
 * 3. It sets the GenTrigger usable and enabled. */

@method SampInteractionClass, MSG_SAMP_INTERACTION_CREATE_TRIGGER {
    optr          newTrig;

    newTrig = ObjInstantiate(ObjPtrToHandle(oself),
                            (ClassStruct *)&GenTriggerClass);
    /* The two parameters are the handle of an object block and the
     * pointer to a class definition. The object block is the same
     * one containing the GenInteraction, whose optr is contained in
     * the standard oself parameter. The class structure pointer points
     * to the class definition of GenTriggerClass. */

    /* Now it is necessary to dereference our pself parameter. Because
     * the ObjInstantiate() call could move this object block (it must
     * allocate new space in the block, and this can cause the block to
     * move), we have to reset our pointer based on our optr. This is
     * done with one of the dereference routines. */
    pself = ObjDerefGen(oself);

    /* Now set the trigger as the last child of the GenInteraction. */
    @call self::MSG_GEN_ADD_CHILD(newTrig, (CCO_MARK_DIRTY | CCF_LAST);

    /* Now set the trigger usable and enabled. */
    @call newTrig::MSG_GEN_SET_USABLE(VUM_DELAYED_VIA_UI_QUEUE);
    @call newTrig::MSG_GEN_SET_ENABLED(VUM_NOW);
}
```

Copying a Generic Tree

The fourth way to create new objects is by using the message `MSG_GEN_COPY_TREE`. This, when sent to a generic object in a generic tree, copies an entire generic tree below and including the object into another, pre-existing object block.

This is an easy way to copy generic trees, one of the more common purposes of creating new objects. However, it only works with the generic objects (with a superclass **GenClass**). Trees created using MSG_GEN_COPY_TREE can be destroyed with MSG_GEN_DESTROY.

For an example of MSG_GEN_COPY_TREE use, see the SDK_C\GENTREE sample application.

5.4.6.2 Working With Object Blocks

5.4

```
ObjIncInUseCount(), ObjDecInUseCount(), ObjLockObjBlock(),  
ObjFreeObjBlock(), ObjFreeDuplicate(),  
ObjTestIfObjBlockRunByCurThread(), ObjBlockSetOutput(),  
ObjBlockGetOutput()
```

Once you have an object block created, either with **ObjDuplicateResource()** or with the memory routines, there are several things you can do with it. It may be treated as a normal memory block, but there are also several routines for use specifically with object blocks:

ObjIncInUseCount() and **ObjDecInUseCount()** increment and decrement an object block's in-use count (used to ensure the block can't be freed while an object is still receiving messages). **ObjLockObjBlock()** locks the object block on the global heap. **ObjFreeObjBlock()** frees any object block. **ObjFreeDuplicate()** is the low-level routine which frees an object block created with **ObjDuplicateResource()**.

ObjTestIfObjBlockRunByCurThread() returns a Boolean value indicating whether the calling thread runs a given object block.

ObjBlockSetOutput() and **ObjBlockGetOutput()** set and return the optr of the object set to receive output messages (i.e., messages sent with travel option TO_OBJ_BLOCK_OUTPUT) from all the objects within the object block.

5.4.6.3 Working With Individual Objects

```
ObjIsObjectInClass(), ObjGetFlags(), ObjSetFlags(),  
ObjDoRelocation(), ObjDoUnRelocation(), ObjResizeMaster(),  
ObjInitializeMaster(), ObjInitializePart()
```

The kernel supplies several routines for working with and modifying individual object chunks and object data. These are all described fully in the Routine Reference Book; most are not commonly used by applications.

ObjIsObjectInClass() takes a class and an optr and returns whether the object is a member of the class. **ObjGetFlags()** returns the object flags for a given object instance chunk; **ObjSetFlags()** sets the flags to passed values. **ObjDoRelocation()** processes any passed instance data fields in the object declared as relocatable; **ObjDoUnRelocation()** returns the passed relocatable fields to their index values.

ObjInitializeMaster() causes the system to build out a particular master group's instance data for an object. **ObjInitializePart()** causes the system to build all master groups above and including the passed level. (This will also resolve variant classes.) **ObjResizeMaster()** resizes a given master part of the instance chunk, causing the chunk to be resized.

5.4.6.4 Managing Object Trees

```
ObjLinkFindParent(), ObjCompAddChild(),  
ObjCompRemoveChild(), ObjCompMoveChild(),  
ObjCompFindChildByOptr(), ObjCompFindChildByNumber(),  
ObjCompProcessChildren()
```

Many objects will be part of object trees. Nearly all generic UI and visible objects exist as members of trees for organizational purposes. Object trees can be useful, powerful, and convenient mechanisms for organizing your objects.

An object tree is made up of “composite” objects—objects which may or may not have children. The distinguishing characteristic of a composite object is that it has one instance data field declared with the **@composite** keyword and another declared with the **@link** keyword. The **@composite** field contains a pointer to the object's first child in the tree, and the **@link** field

contains a pointer to the object's next sibling. This representation is shown in Figure 5-18.

If you set up an object resource block containing composite objects, it's very easy to set up an object tree. Your generic UI objects are declared in a tree with the GenApplication object at its head. Additionally, it's easy to alter an object tree once it's been created. The kernel provides several routines, and **MetaClass** uses several messages for adding, removing, and moving objects to, from, and within trees.

5.4

◆ **ObjLinkFindParent()**

This routine finds the optr of the calling object's direct parent. The kernel traverses the link fields until it returns to the parent object.

◆ **ObjCompFindChildByOptr()**

This routine returns the number of the child (first, second, third, etc.) whose optr is passed. The child must exist and must be a child of the calling object.

◆ **ObjCompFindChildByNumber()**

This routine returns the optr of the child whose number (first, second, etc.) is passed.

◆ **ObjCompAddChild()**

This routine takes an object's optr and adds it to the caller's list of children. Depending on the flags passed, the child may be inserted in any child position (first, second, etc.).

◆ **ObjCompMoveChild()**

This routine takes a child object and moves it to a new position. However, it will still remain a child of the calling object. If you want to move the child to be a child of a different object, you must first remove it from the tree altogether and then add it to the other parent.

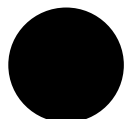
◆ **ObjCompRemoveChild()**

This routine removes a child object from the object tree.

◆ **ObjCompProcessChildren()**

This routine calls a callback routine for each child of the calling object in turn. The callback routine may do virtually anything (except destroy the object or free its chunk or something similar).

By using the above routines, you can fully manipulate any object tree and the objects within it.



5.4

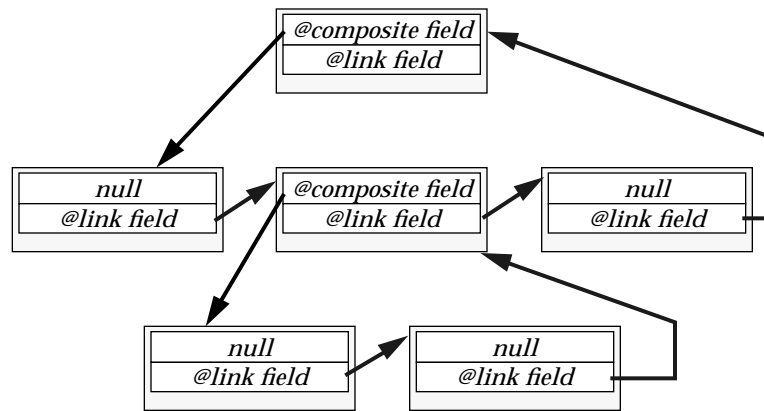


Figure 5-18 *Structure of an Object Tree*

An object's composite field points to its first child, and its link field points either to its next sibling or back to the parent. You can see that, by following these links, any object is accessible from any other object in the tree.

5.4.6.5 Detaching and Destroying Objects

```
MSG_META_DETACH, MSG_META_DETACH_COMPLETE, MSG_META_ACK,
MSG_META_OBJ_FLUSH_INPUT_QUEUE, MSG_META_OBJ_FREE,
MSG_META_FINAL_OBJ_FREE, ObjInitDetach(), ObjIncDetach(),
ObjEnableDetach(), ObjFreeChunk()
```



Advanced Topic

MetaClass handles detaching and destruction.

While creating objects is actually quite simple, detaching and destroying them can be quite involved. For this reason, GEOS does most of the work for you, and in most cases you won't have to worry about what happens when your application is shut down or saved to a state file. However, if you instantiate individual objects of your own classes, you should be very careful about how your objects are detached from the system and destroyed.

Detaching objects involves severing all of the object's ties with the rest of the system. Destruction of an object entails actually removing the object's instance chunk and related structures, making sure that it has handled all its waiting messages.

Throughout its life, an object is likely to become involved with a number of other things—other objects, drivers, files, streams, the memory manager, the kernel—and each of these things may send the object messages from time to time. The task, when detaching an object from the system, is to sever all the object's ties with the outside world, to make sure that no other entity will ever try to contact the object again.

To those unfamiliar with these problems, they can be overwhelming. However, GEOS takes care of them for you in most situations. All generic and visible objects, all objects in object trees, and all objects that maintain an active list will automatically (in nearly all cases) have the detach functionality built in by **MetaClass**.

5.4

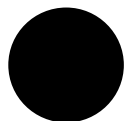
The Detach Sequence

When an object receives MSG_META_DETACH, it begins the detach sequence to sever all its ties with other entities in the system. It must first notify all its children and all the objects on its various notification lists that it will be going away (most often, all its children, by association, will also be detached). It then must clear its message queues. Finally, it must acknowledge its detachment to the object that originally sent MSG_META_DETACH. Each of these phases is described in detail below and is implemented by **MetaClass**. You have to do none of this work unless your object is truly a special case.

Detaching in conjunction with destruction is somewhat intricate because not only must the object notify all other interested parties, but it must also receive acknowledgment of the notice. Otherwise, the notification and some other message could be sent at the same time, and the object could be destroyed before the notification can be handled. (Destruction is discussed in the following section.)

Because any object may be put in the position of being detached and then immediately destroyed, it must send out notification and then wait until all the notices have been acknowledged before continuing with other tasks. The kernel and **MetaClass** implement a mechanism for this using four kernel routines. Again, you do not need to do this since all classes have **MetaClass** as their root.

First the object being detached (in its MSG_META_DETACH handler) calls the routine **ObjInitDetach()**. This tells the kernel that the object is initiating a detach sequence and that the acknowledgment mechanism must be set up.



5.4

The kernel will allocate a variable data entry to hold a count of the number of notices sent and acknowledgments received.

After this, the object must send a `MSG_META_DETACH` or its equivalent to each of its children and each of the objects on its active list. With each notice sent, the object *must* call **`ObjIncDetach()`**, which increments the notice count.

After sending all the notices, the object then calls the kernel routine **`ObjEnableDetach()`**. This notifies the kernel that all the notices have been sent and that the object is waiting for the acknowledgments.

Acknowledgment comes in the form of `MSG_META_ACK` and is received by the object being detached. `MSG_META_ACK` is handled by **`MetaClass`** and will decrement the notice count, essentially saying there are one fewer notices left to be received. When the final `MSG_META_ACK` is received (setting the notice count to zero) and **`ObjEnableDetach()`** has *also* been called, the kernel will automatically send a `MSG_META_DETACH_COMPLETE` to the object. This assures the object that it will never receive another message from another entity in the system.

The final step in the detach sequence is acknowledging that the object has been detached. In its `MSG_META_DETACH_COMPLETE` handler, the object should send a `MSG_META_ACK` to the object that originated the detach sequence. This will allow that object to continue with its detach sequence if it was involved in one; without this step, only leaves of object trees could ever be detached. This final step is provided in default handlers in **`MetaClass`** and is inherited by all objects.

The Destruction Sequence

The destruction sequence must be initiated from outside and will begin when the object receives a `MSG_META_OBJ_FREE`. Often, the `MSG_META_OBJ_FREE` will be sent by the object to itself.

The destruction sequence consists of three steps: First, the object must clear out its message queues; even though it is detached and can not receive new messages, there may be some left over in the queue (an error if it occurs). Second, it must finish executing its code and working with its instance data. Third, it must free its instance chunk. Each of these steps is described below.

Even though the object has notified the rest of the system that it is going away, it still must flush its message queues of any unhandled messages. These messages could have been received between the time the original MSG_META_OBJ_FREE was received and notification was sent out (due to interrupts or context switching). To clear its message queues, the object must send itself a MSG_META_OBJ_FLUSH_INPUT_QUEUE, which will ensure that any messages in the queues are handled appropriately before the object shuts down. This step is handled automatically by the MSG_META_OBJ_FREE handler in **MetaClass**. You should never have to send this message, and indeed its use is discouraged.

5.4

To the outside world, the second and third steps seem like a single step. However, MSG_META_OBJ_FREE can not simply free the instance chunk after the queues are cleared; it must be able to access the instance chunk until all the method code has been executed. So, MSG_META_OBJ_FREE sends the final message to the object, MSG_META_FINAL_OBJ_FREE, and then exits. MSG_META_FINAL_OBJ_FREE waits a short while and then frees the object's chunk. This ensures that MSG_META_OBJ_FREE has finished and the chunk is not being used by any code.

Possible Pitfalls

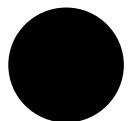
If you're not careful, you can cause the detach mechanism to fail by instantiating an object on the fly and saving that object's optr. If the object is then detached and you don't account for the saved optr, you could later send a message to a nonexistent object. This has undefined results and can be nearly impossible to track down.

Note that objects created within resources and by **ObjDuplicateResource()** will almost always automatically be taken care of by the detach mechanism. Objects you create with **ObjInstantiate()** are the ones to be careful with.

5.4.6.6 Saving Object State

```
ObjSaveBlock(), ObjMarkDirty(), ObjMapSavedToState(),  
ObjMapStateToSaved()
```

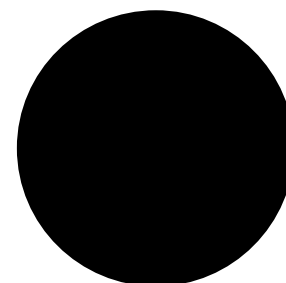
Object state saving is almost entirely contained within the system. For the most part, only UI objects are saved to state files; however, you can mark



other object blocks for saving. State saving is described in full in section 6.1.4 of chapter 6.

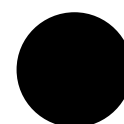
5.4

Applications and Geodes

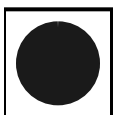


6

6.1	Geodes	242
6.1.1	Geode Components and Structures	244
6.1.1.1	Geode Attributes	245
6.1.1.2	Geode Token	247
6.1.2	Launching an Application.....	247
6.1.3	Shutting Down an Application.....	249
6.1.4	Saving and Restoring State.....	250
6.1.5	Using Other Geodes	253
6.1.5.1	Using Libraries	253
6.1.5.2	Using Drivers	254
6.1.6	Writing Your Own Libraries	257
6.1.7	Working with Geodes.....	259
6.1.7.1	Accessing the Application Object	259
6.1.7.2	General Geode Information.....	259
6.1.7.3	Managing Geode Event Queues	260
6.1.8	Geode Protocols and Release Levels.....	261
6.1.8.1	Release Numbers	261
6.1.8.2	Protocol Numbers.....	262
6.1.9	Temporary Geode Memory	264
6.2	Creating Icons	265
6.2.1	The Token Database	265
6.2.2	Managing the Token Database File	267
6.3	Saving User Options	269
6.3.1	Saving Generic Object Options	269
6.3.2	The GEOS.INI File	271
6.3.2.1	Configuration of the INI File.....	271
6.3.2.2	Managing the INI File	273
6.3.2.3	Writing Data to the INI File.....	274
6.3.2.4	Getting Data from the INI File	275
6.3.2.5	Deleting Items from the INI File	276



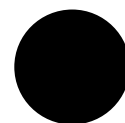
6.4	General System Utilities	277
6.4.1	Changing the System Clock	277
6.4.2	Using Timers	277
6.4.3	System Statistics and Utilities	278
6.4.4	Shutting the System Down.....	279
6.5	The Error-Checking Version.....	280
6.5.1	Adding EC Code to Your Program.....	281
6.5.2	Special EC Routines.....	282
6.6	Inter-Application Communication.....	283
6.6.1	IACP Overview	284
6.6.2	GenApplicationClass Behavior	286
6.6.3	Messages Across an IACP Link	287
6.6.4	Being a Client.....	288
6.6.4.1	Registering as a Client	289
6.6.4.2	Unregistering as a Client	292
6.6.5	Being a Server	292
6.6.5.1	Registering and Unregistering a Server.....	293
6.6.5.2	Non-Application Servers and Clients	294



This chapter discusses the life of an application as well as several topics most application programmers will want to cover at one time or another. Before reading this chapter, you should have read each of the previous chapters; this chapter builds on the knowledge you gained in those chapters.

This chapter details the components and features of a *geode*, a GEOS executable. The chapter is presented in the following five sections:

- ◆ **Geodes**
The *Geodes* section describes how libraries, drivers, and applications are launched, used, and shut down. It also discusses the general sections of a geode and the geode's executable file.
- ◆ **Creating Icons**
The *Creating Icons* section describes the structure of the Token Database, how icons are stored, managed, and created.
- ◆ **Saving User Options**
The *Saving Options* section describes how to work with the GEOS.INI file.
- ◆ **General System Utilities**
The *General System Utilities* section describes several different mechanisms provided by GEOS that you may find useful in your application or other geode.
- ◆ **The Error-Checking System Software**
The *Error-Checking System Software* describes the differences between the debugging and standard versions of the system software. There are two versions of nearly every part of the system software including the kernel, the UI, and most libraries and drivers.
- ◆ **IACP: Inter-Application Communication Protocol**
The *IACP* section describes how applications can communicate with each other and pass data back and forth. This protocol works for applications that are not necessarily loaded and running; thus, an application can change another application's data dynamically—the recipient application is automatically started if it is not already running. (This is an advanced topic; most programmers will not need to read this section.)



6.1 Geodes

Geode is the term used to describe a GEOS executable. Just as DOS has executables (programs) that reside in files on a disk, so too does GEOS. GEOS executables normally have the filename extension .GEO.

Each geode may have up to three different aspects:

6.1

◆ *process*

Most of your geodes will have this aspect. A geode that is a process has an initial event-driven thread started for it by the kernel. All applications and some libraries will have this aspect.

◆ *library*

This aspect indicates that the geode exports *entry points* for other geodes to use. Typically, these entry points describe where either object classes or code for routines is within the geode. A library has a special routine (the library's entry point) that is called by the system when the library or one of its clients is loaded or in the process of being unloaded.

◆ *driver*

This aspect indicates that the geode hides the details of some device or similarly changeable thing from the system. A driver has a special structure that defines its type, and it has a single entry point called a *strategy routine*. All calls to the driver pass through this strategy routine.

A geode can have any combination of these aspects. For example, the print spooler is a process-library (and therefore provides routines for other geodes while also having a thread of its own), but the sound library is actually a library-driver since it manipulates the machine's sound hardware.

Library and driver geodes that do not have the process aspect do not initially have event-driven threads. Therefore, typically they will not contain objects, just procedural code. They can contain objects, however, as any geode is free to create an event-driven thread for itself at any time. In fact, the parallel port driver does just that when it is printing to a port through DOS or BIOS.

Geodes are loaded either with a call to the kernel routine **GeodeLoad()** or as a side effect of a library's client being loaded (in that case, the library geode will be loaded as well). The generic UI supplies the special routine **UserLoadApplication()**, which you may use to load an application—a

geode which has both a process aspect and its process class subclassed off of **GenProcessClass** (and therefore can put generic UI objects on the screen).

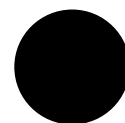
Once a geode has been loaded, it is identified by its *geode handle*, which is the memory handle of the block that holds all the geode's system-administrative data. This block is called the *core block* and should not be accessed by anything other than the kernel. The geode handle is also used to determine the owner of a particular block in memory; when queried for the owner of a particular block, the kernel will return the geode handle of the geode that owns that block. A geode is the only entity that may own a system resource. If the geode is a process, the geode handle may also be known as a process handle.

6.1

When a geode is loaded, its core block is connected to a linked list of the core blocks of other geodes running in the system. This linked list is chronological, with the first entry belonging to the first geode loaded and the last entry belonging to the most recent geode loaded. Each core block contains an entry for the handle of the next core block in the list; the kernel can follow these links to locate any geode in the system. (Only the kernel may do this.)

After the core block is appended to the list, GEOS scans the list for other instances of the same core block. If the geode has been loaded more than once, it will have multiple instances in the list (one instance of the core block for each time the geode is loaded; each core block references the same copy of the geode, however). GEOS then copies the shared-resource handles from an existing core block (if found) into the new core block, thus reducing the amount of work required to load a particular geode multiple times (the shared resources do not need to be reloaded or recreated). Non-shared resource handles are not copied; the resources are loaded or constructed as necessary.

Each geode's core block contains a reference count for that particular geode. When the geode is first loaded, the reference count is set to one. If the geode is a process, the act of initializing the process thread increments the reference count. Each time the geode is loaded again, the new core block will get its own reference count. If the geode is loaded implicitly (as a library, with **GeodeUseLibrary()**, or with **GeodeUseDriver()**), or if it spawns a new thread, it will receive yet another reference count.



The reference count is decremented when a thread owned by the geode exits. If a client of a library geode exits, the library's reference count goes down by one.

When a geode's reference count reaches zero, all the geode's non-sharable resources are freed along with all the file, event, and timer handles owned by the geode. If a sharable resource is co-owned by another instance of the geode, ownership is transferred to the geode's next-oldest instance. (Shared resources are always owned by the oldest instance of their geode.) Once the resources have been freed or transferred, the core block is removed from the linked list and is freed.

To make sure no synchronization problems occur while updating the core block list (e.g. a geode is being loaded while it has just been freed), GEOS maintains an internal semaphore. The geode loading and freeing routines automatically maintain this semaphore.

6.1.1 Geode Components and Structures

A geode is simply a special type of GEOS file. It has a special file header that gets loaded in as the geode's core block. This file header contains the geode's type, attributes, release and protocol levels, and many other pieces of information necessary for GEOS to work with the geode. You never will have to know the exact structure of this header as the kernel provides routines necessary to access important portions of it.

Several important items contained in the header are listed below.

- ◆ **Core Block Handle**
The core block contains its own memory handle, filled in when the geode is loaded into memory.
- ◆ **Geode Attributes**
Each geode has a record of type **GeodeAttrs**. The geode attributes are described below in section 6.1.1.1 on page 245.
- ◆ **Release and Protocol Levels**
Each geode can have release and protocol levels associated with it to ensure compatibility between different versions. Release and protocol levels are discussed in section 6.1.8.2 on page 262.

- ◆ **Geode Name**
Each geode has a geode name and extension specified in the geode parameters (**.gp**) file.
- ◆ **Geode Token**
Every geode has a token associated with it in the token database. The token describes the geode's icon and is a structure of type **GeodeToken**. Tokens and icons are discussed in section 6.2 on page 265.
- ◆ **Geode Reference Count**
The geode's reference count is stored in the core block. See above for a discussion of the reference count and how it's managed. 6.1
- ◆ **Geode File Handle**
The file handle of the geode identifies the open GEO file so the kernel can locate, load, and access its various resources.
- ◆ **Geode Process Handle**
Each geode that has a parent process will have the handle of the process in its core block. This is the handle of the parent process' core block. (The parent process is the owner of the thread that loaded the geode. It is notified when the geode exits, if the exiting geode is a process.)
- ◆ **Handle of the Next Geode**
Because geode core blocks are stored in a linked list, each core block must contain a reference to the next geode in the list.
- ◆ **Handle of Private Data Space**
Each geode that is not the kernel can have a "private data space." This private data space is discussed in section 6.1.9 on page 264.
- ◆ **Resource, Library, and Driver Information**
Each geode that imports or exports library or driver information must keep the import and export specifics available. Additionally, each geode must keep track of the resources it owns. All this information is stored in tables referenced from within the core block.

6.1.1.1 Geode Attributes

Each geode has in its core block a record of type **GeodeAttrs**. This record defines several things about the geode, including which aspects it uses and which of its aspects have been initialized. The **GeodeAttrs** record contains one bit for each of the following attributes.



6.1

GA_PROCESS

This geode has a process aspect and therefore an initial event-driven thread.

GA_LIBRARY This geode has a library aspect and therefore exports routines (and possibly object classes).

GA_DRIVER This geode has a driver aspect and therefore has a driver table, in which the strategy routine is specified.

GA_KEEP_FILE_OPEN

This geode must have its .GEO file kept open because its resources may be discardable or are initially discarded.

GA_SYSTEM This geode is a privileged geode and is almost certainly a system-used driver. These geodes have special exit requirements.

GA_MULTI_LAUNCHABLE

This geode may be loaded more than once and therefore may have more than one instance of its core block in memory.

GA_APPLICATION

This geode is a user-launchable application.

GA_DRIVER_INITIALIZED

This flag is set if the geode has had its driver aspect initialized (if the driver's strategy routine has been initialized). This flag will be set dynamically by the kernel.

GA_LIBRARY_INITIALIZED

This flag is set if the geode has had its library aspect initialized (if the library's entry routine has been called). This flag will be set dynamically by the kernel.

GA_GEODE_INITIALIZED

This flag is set if all aspects of the geode have been initialized.

GA_USES_COPROC

This geode uses a math coprocessor if one is available.

GA_REQUIRES_COPROC

This geode requires the presence of a math coprocessor or a coprocessor emulator.

GA_HAS_GENERAL_CONSUMER_MODE

This geode may be run in the General Consumer (appliance) Mode.

GA_ENTRY_POINTS_IN_C

This geode has its library entry routine in C rather than assembly language.

6.1.1.2 Geode Token

6.1

As stated above, every geode is associated with a token in the token database. This token is defined by the use of a **GeodeToken** structure. This structure and its uses are discussed in section 6.2 on page 265.

6.1.2 Launching an Application

```
GeodeLoad(), UserLoadApplication(),
MSG_GEN_PROCESS_OPEN_APPLICATION
```

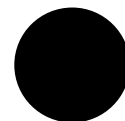
An application is a geode with its GA_APPLICATION attribute set. This type of geode may be launched by the user through GeoManager or some other means provided by the system or another application. For the most part, the system will invoke and carry out the launch; your responsibilities are limited.

An application may be loaded in essentially two ways: It may be launched, or it may be reloaded from a state file. In both cases, the kernel will load the proper resources and build out the UI properly according to the application.

Most of the procedure of launching is handled within **GenProcessClass**, a subclass of **ProcessClass**. An application should define its own subclass of **GenProcessClass** for its Process objects (event-driven threads not acting as Process objects should be subclassed from **ProcessClass**, not **GenProcessClass**). The launch procedure may be invoked by any thread and by any geode in either of the following ways:

◆ **GeodeLoad()**

This loads a geode from a given file and begins executing it based on the geode's type. **GeodeLoad()** takes the name of the geode's file as well as a priority to set for the new geode's process thread, if it has one.



GeodeLoad() first creates the process thread of the application, then sends this thread a message. The process thread (a subclass of **GenProcessClass**) then creates a UI thread for the application.

GeodeLoad() will have to create two threads for the application: one for the process and one for the UI of the geode.

◆ **UserLoadApplication()**

Used by most application launchers, this routine loads a geode from the standard GEOS application directory. (C programmers will generally use `MSG_GEN_PROCESS_OPEN_APPLICATION` instead.) This routine takes some additional parameters and can load a geode either in engine mode or from a state file as well as in the normal open mode (see below). The base functionality of opening and loading the geode is implemented in this routine by a call to **GeodeLoad()**. Note, however, that this routine may only open application geodes—geodes with the `GA_APPLICATION` attribute set.

6.1

Geodes may be launched in three modes:

- ◆ *Application* mode launches the geode, loads all its active resources, builds its UI gadgetry, and sets it usable. The geode must be an application—that is, it must have its `GA_APPLICATION` attribute set.
- ◆ *Engine* mode launches the geode but leaves its UI objects unusable (it never brings them on-screen). Engine mode is useful if you need to launch an application to grab information from it. GeoManager uses engine mode to launch an application, extract its icon, and put the icon in the token database. For efficiency, the application is never set usable, so its UI is never built.
- ◆ *Restore* mode launches the geode from a saved state file, loading in resources and merging them with the default resources. This mode is invoked automatically by the UI if it is restoring the system or the geode from saved state. This mode is handled automatically by **GenProcessClass** (your Process object).

It is possible, however, for one application to launch another in a custom mode. If this is done, the application being launched is responsible for implementing the special mode.

When the launch process has been initiated with the above routines, a thread is created for the application and its Process object is loaded immediately. Also loaded is the application's GenApplication object. The Application and

Process objects interact with the User Interface to load the objects on the application's active list and set them all usable, bringing them up on-screen.

Near the end of this procedure, just before the GenApplication is set usable, the Process object will receive a message based on the mode of launch. If the application must set up any special notification (such as for the quick-transfer mechanism) or must restore special state file data, it should intercept this message. Typically the message received will be MSG_GEN_PROCESS_OPEN_APPLICATION—two others are received (when restoring from state and when opening in engine mode), but they should not be intercepted.

6.1

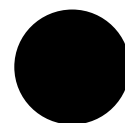
6.1.3 Shutting Down an Application

```
MSG_GEN_PROCESS_CLOSE_APPLICATION,  
MSG_GEN_PROCESS_CLOSE_ENGINE,  
MSG_GEN_PROCESS_CLOSE_CUSTOM, MSG_META_QUIT
```

Just as loading an application is handled almost entirely by the system and GEOS classes, application shutdown is also fairly automatic. If the application intercepted MSG_GEN_PROCESS_OPEN_APPLICATION for its own purposes on startup, it likely has to do a little cleanup; otherwise, it won't have to worry about shutting down. (See "Saving and Restoring State" on page 250 for special information on using this message.)

Any object in the system may cause an application to shut down. Usually, shutdown occurs either when the system is being exited (when a user exits to DOS, for example) or when the user has closed the application. Therefore, the usual source of the shutdown directive is the User Interface.

An application begins shutting down when either its Process object or its Application object receives a MSG_META_DETACH. If you want to cause a shutdown manually, you should send MSG_META_QUIT to the application's GenApplication object; this will execute some default functions and then send the appropriate MSG_META_DETACH. Essentially, the same detach and destruction mechanisms used for any object are used for the entire application. The object receiving MSG_META_DETACH passes the message along to all of its children and to all the objects on its active list. (If a



MSG_META_DETACH is used without MSG_META_QUIT, the application will create a state file.)

When they have all acknowledged the detach, the application acknowledges the detach and sets itself unusable. It automatically flushes its message queues before shutting down to avoid synchronization problems. You should not subclass the MSG_META_DETACH handler unless you have special needs for cleaning up or sending special detach messages to other objects or geodes. If you do subclass it, you must call the superclass at the end of your handler. Otherwise, the application will not finish detaching (see section 5.4.6.5 of chapter 5).

Instead of intercepting MSG_META_DETACH, though, the application may intercept the mode-specific message it will also receive. Depending on the mode in which it was launched, the application will receive (via the Process object) either MSG_GEN_PROCESS_CLOSE_APPLICATION (for application mode) or MSG_GEN_PROCESS_CLOSE_ENGINE (for engine mode). There is no special shutdown message for shutting down to a state file; instead, MSG_GEN_PROCESS_CLOSE_APPLICATION is used.

When the system shuts down or task-switches, a different type of shutdown occurs. Applications (or other objects interested in this event) must register for notification on the notification list GCNSLT_SHUTDOWN_CONTROL (notification lists are described in “General Change Notification,” Chapter 9). When the system shuts down or task-switches, the object will then receive a MSG_META_CONFIRM_SHUTDOWN, at which time the object must call **SysShutdown()**.

6.1.4 Saving and Restoring State

```
ObjMarkDirty(), ObjSaveBlock()
```

Nearly all applications will save and restore their state so the user may shut down and return to precisely the same configuration he or she left. Saving of state is almost entirely contained within the system software; for the most part, only UI objects are saved to state files. You can, however, mark other object blocks and data for saving to a state file.

The state file for an application is a VM file containing object blocks. Only object blocks may be saved to the state file, though you can save LMem data

by setting up object blocks with only data chunks in them. (Create the blocks with **MemAllocLMem()**, passing type `LMEM_TYPE_OBJ_BLOCK`, then simply use **LMemAlloc()** to allocate data chunks.) You can also save an extra data block to the state file using `MSG_GEN_PROCESS_CLOSE_APPLICATION` and `MSG_GEN_PROCESS_OPEN_APPLICATION`. In the close message, you can return the handle of an extra block to be saved to the state file; in the open message, the handle of the extra block is given to you, and you can restore this data as necessary. See the reference information for these messages under **GenProcessClass** for more information.

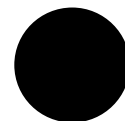
6.1

When a state file is saved, the system recognizes and saves only the dirty (modified) objects and chunks. Later, when state is restored, the system merges the changes in the state file with the original object blocks, resulting in the state that was saved.

For individual objects or entire object blocks to be saved to the state file, they must be marked dirty. Generic objects automatically mark themselves dirty at the appropriate times, so you don't have to worry about them. To mark other objects dirty, use the routine **ObjMarkDirty()**. Each object which has been marked dirty will be saved to a state file when appropriate. If you want to save an entire object block to the state file, you can call **ObjSaveBlock()** on the block; the system will save the entire block, not just the dirty chunks.

State files are dealt with at only two times: First, when the system starts up, it will check for the existence of application state files. If a state file exists, the system will attempt to load the application belonging to it; after loading the application's resources, it will merge the state changes with the default settings to restore the original state.

The second time state files are used is when the system shuts down. A simple shutdown (called a "detach") is invoked only by the UI and is not abortable. When a detach occurs, the system shuts down all geodes as cleanly and quietly as possible, saving them to state files. Only certain geodes will respond in extreme cases, offering the user the option of delaying the detach or cancelling an operation in progress. An example of this is the GEOS spooler; if one or more jobs are actively printing or queued for printing, the spooler will ask the user whether the job should continue and the detach be delayed, or whether the job should be aborted or delayed until the next startup. The spooler can not abort the detach in any case.



Another type of detach is called a “quit.” Any geode may invoke a quit, which is actually a two-step detach. A quit will first notify all other geodes that the system will soon be detaching; other geodes then have the chance to abort the quit if they want. For example, if a terminal program were downloading a file and received a quit notification, it could ask the user whether she wanted to abort the quit or the download. If the user wanted to finish the download, she would abort the quit; if she wanted to quit, she would abort the download. The system would then either shut down via a normal detach or stop the quit sequence.

When a geode is first launched, no state file exists for it. The state file is not created until the geode is actually detached to a state file. If a geode is restored from a state file, the file will exist until the geode is detached again. A geode that gets closed (not detached to state) will remove any state file it may have created during a previous detach. A geode that is detached to state will create or modify its state file as appropriate.

The state of an application (how it was launched) is reflected in the *GAI_states* field of the GenApplication object. To retrieve the application's state, send it MSG_GEN_APPLICATION_GET_STATE. It will return a value of **ApplicationStates**. The most frequent use of this message is by applications that need to know whether a “quit” is underway when their receive the MSG_GEN_PROCESS_CLOSE_APPLICATION message; the process object will query the GenApplication and see if it is in the AS_QUITTING state.

In addition to the above state-saving functionality, the kernel provides two routines that translate handles between the state file and memory.

ObjMapSavedToState() takes the memory handle of an object block and returns its corresponding state file VM block handle.

ObjMapStateToSaved() takes the state file VM block handle and returns the corresponding memory block handle, if any.

If your application's documents are VM files, it is a very simple matter to save document state. In fact, if you use the GenDocument and document control objects, they will take care of document state saving for you. Be sure that the VM file has the VMA_BACKUP flag set in its **VMAttributes**; then you can simply call **VMUpdate()** on the document file. (Note—do not use **VMSave()** instead; it will erase the backup and lock in the user's changes to the

document.) If you are not using GEOS VM files, it is up to you how and if you will save the document's state.

6.1.5 Using Other Geodes

Often, geodes will have to use other geodes. For example, a communications program will use the Serial Driver, and a draw application will use the Graphic Object Library. Normally, this is taken care of by the compiler and the linker when you include a library or driver in your **.goc** and **.gp** files.

6.1

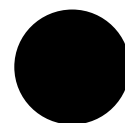
Other times, however, an application will have to load libraries or drivers on the fly and then free them some time later. This section describes how to load, use, and free libraries and drivers.

6.1.5.1 Using Libraries

```
GeodeUseLibrary(), GeodeFreeLibrary()
```

Libraries are always referenced by their file names or by their geode handles. It's easiest, however, to use the file name of the library when loading it—the system will locate the library for you. It's unusual to need to load a library for use with your geode; in almost all cases it's easiest to include the library in your **.goc** and **.gp** files and have the system load and link the library automatically. (To do this, include the library's interface definition file in your code file and list the library's geode name in your geode parameters file.)

If you need to load a library dynamically, though, use **GeodeUseLibrary()**. This routine takes the protocol numbers expected of the library (see section 6.1.8.2 on page 262) and the library geode's filename. It will locate and load the library if not already loaded. If the library is already loaded, it will increment the library's reference count. When you are done using a library loaded with **GeodeUseLibrary()**, you must free the library's instance with **GeodeFreeLibrary()**.



6.1.5.2 Using Drivers

```
GeodeUseDriver(), GeodeInfoDriver(),  
GeodeGetDefaultDriver(), GeodeSetDefaultDriver(),  
GeodeFreeDriver()
```

6.1

Drivers are referenced by either their permanent names or their geode handles. Most drivers used by applications will be loaded automatically by the kernel; the application must have the driver's permanent name specified in its **.gp** file. Should an application need to use a driver not included in its parameters file, however, it can do so with the routines described below.

When you need to use a driver, the **GeodeUseDriver()** routine will locate and load it, adding it to the active geodes list. You must pass the desired driver geode's filename as well as the expected protocol levels of the driver. The routine will return the driver's geode handle. If you load a driver dynamically, you must free it with **GeodeFreeDriver()** when your geode shuts down or otherwise finishes using the driver.

If you know a driver's geode handle, you can easily retrieve information about it with the routine **GeodeInfoDriver()**. This returns a structure of type **DriverInfoStruct**, which contains the driver's type (**DriverType**), the driver's attributes, and a far pointer to the driver's strategy routine. Many driver types have an expanded information structure, of which **DriverInfoStruct** is just the first field. Video driver information structures, for example, also contain dimensions and color capabilities (among other things) of the particular devices they drive. The driver information structure is shown below.

```
typedef struct {  
    void (*DIS_strategy)();  
    DriverAttrs DIS_driverAttributes;  
    DriverType DIS_driverType;  
} DriverInfoStruct;
```

The *DIS_strategy* field of the structure contains a pointer to the driver's strategy routine in fixed memory. After the driver has been loaded, its strategy routine is called directly with a driver function name.

The *DIS_driverAttributes* is an attribute record of type **DriverAttrs**, the flags of which are shown below:

DA_FILE_SYSTEM

This flag indicates that the driver is used for file access.

DA_CHARACTER

This flag indicates that the driver is used for a character-oriented device.

DA_HAS_EXTENDED_INFO

This flag indicates that the driver has a **DriverExtendedInfo** structure.

6.1

The *DIS_driverType* contains the type of driver described by the information structure. The types that may be specified in this field are listed below:

DRIVER_TYPE_VIDEO

This is used for video drivers.

DRIVER_TYPE_INPUT

This is used for input (mouse, keyboard, pen, etc.) drivers.

DRIVER_TYPE_MASS_STORAGE

This is used for storage device drivers.

DRIVER_TYPE_STREAM

This is used for stream and port (parallel, serial) drivers.

DRIVER_TYPE_FONT

This is used for font rasterizing drivers.

DRIVER_TYPE_OUTPUT

This is used for output drivers other than video and printer drivers.

DRIVER_TYPE_LOCALIZATION

This is used for drivers that facilitate internationalization.

DRIVER_TYPE_FILE_SYSTEM

This is used for file system drivers.

DRIVER_TYPE_PRINTER

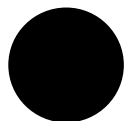
This is used for printer drivers.

DRIVER_TYPE_SWAP

This is used for the system's memory-swapping drivers.

DRIVER_TYPE_POWER_MANAGEMENT

This is used for devices that have power management systems.



DRIVER_TYPE_TASK_SWITCH

This is used for devices or systems that have task switchers.

DRIVER_TYPE_NETWORK

This is used for special networks that require driver functionality.

6.1

When you want a driver to perform one of its functions, you must call its strategy routine. The strategy routine typically takes a number of parameters, one of which is the function the driver should perform. The **DriverInfoStruct** contains a far pointer to the strategy routine; your application should store this far pointer and call it directly any time one of the driver's functions is needed. However, because the driver may be put in a different location each time it's loaded, you should not save the pointer in a state file. Note that this scheme of accessing drivers directly can only be implemented in assembly language. Some drivers may provide library interfaces as well as their standard driver interface; this allows routines to be written in C.

GEOS maintains default drivers for the entire system. The types of default drivers are described by **GeodeDefaultDriverType**; all the types are shown below. They are called default drivers because the default for each category of driver used by the system is stored in the GEOS.INI file. GEOS will, upon startup, load in the default driver of each category.

GDDT_FILE_SYSTEM

GEOS may use several file system drivers during execution. A file system driver allows GEOS to work on a given DOS (or substitute file access system). The primary driver is considered the default driver.

GDDT_KEYBOARD

The system may only use one keyboard driver during execution; typically, keyboards can differ from country to country. Not all systems will have a keyboard.

GDDT_MOUSE

The mouse should be usable when the system comes up on the screen; the user does not have to manually load a mouse driver with each execution of GEOS. Not all systems will have a mouse.

GDDT_VIDEO

GEOS must know what type of video driver to use prior to attempting to display itself.

GDDT_MEMORY_VIDEO

The Vidmem driver is a video driver that draws to memory (i.e. to bitmaps). It is used primarily for printing but also for editing bitmaps.

GDDT_POWER_MANAGEMENT

6.1

A few machines use hardware power management systems (most notably some palmtops and some notebook and laptop machines). In order for GEOS to handle the power management hardware properly, it must load the driver on startup. (Most machines will not use this type of driver.)

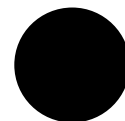
GDDT_TASK Some systems will use a task switcher; in order for GEOS to be a task in one of these, it must have a default task driver.

To retrieve the default that GEOS is using, call the routine **GeodeGetDefaultDriver()** with the appropriate driver type (a member of the type **GeodeDefaultDriverType**). This routine will return the geode handle of the default driver of that type. To set a new default driver for a specified driver type, use **GeodeSetDefaultDriver()**. This routine takes a geode handle and a driver type and sets the system default for that type. Typically, system defaults will be set only by the Preferences Manager application.

6.1.6 Writing Your Own Libraries

Creating a library geode is a simple step beyond creating a normal application. Libraries can have their own process threads or not; most libraries do not, though some will. To create a library geode, you have to do four things:

- ◆ Declare the geode to be a library.
In the geode parameters (**.gp**) file, you must declare the type of the geode to be library.
- ◆ Define an entry point routine.
Create a routine that initializes the library; this routine is called the “entry point routine” and is called each time the library is loaded.



- ◆ Export the entry point routine.
Add another line in the geode parameters (**.gp**) file to export the library's entry point routine.
- ◆ Set the geode's permanent name properly.
Most libraries have the extension of their permanent name be "lib." This helps identify it as a library when using Swat. The permanent name is defined in the geode parameters (**.gp**) file.

6.1

Code Display 6-1 shows an example of the three changes you must make to the geode's parameters file to make it a library.

Code Display 6-1 Defining a Library—the sound.gp File

```
# The Sound Library has the following three lines in its geode parameters file
# (sound.gp) different from how they might appear in an application.

#
# The permanent name of the geode might normally be sound.app if it is an
# application. A library typically has "lib" as its name extension.
#
name sound.lib

#
# Declare the type of the geode to be a library. Applications typically have a
# "process" aspect; many libraries do not. Libraries must also be declared to
# have a "library" aspect. (Note that the Sound Library is exceptional in that
# it has both a driver aspect and a library aspect.)
#
type driver, library, single

#
# Export the geode's entry point routine so the kernel may call it when the
# library is loaded. This routine must be defined somewhere in the geode's code.
# Note that it is exported with the "entry" line rather than the standard
# "export" line; this is to distinguish the exported routine as the entry
# point rather than a typical routine.
#
entry SoundEntry
```

6.1.7 Working with Geodes

The system provides a number of utility routines for getting information and setting attributes of geodes. These are loosely organized throughout the following sections.

6.1.7.1 Accessing the Application Object

6.1

`GeodeGetAppObject()`

GEOS offers a routine for retrieving the optr of an application's GenApplication object. **GeodeGetAppObject()** takes the process handle of the Process object of the application. It returns the optr of the application's GenApplication object.

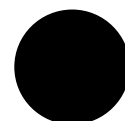
6.1.7.2 General Geode Information

`GeodeFind()`, `GeodeFindResource()`, `GeodeGetInfo()`,
`GeodeGetProcessHandle()`, `GeodeGetCodeProcessHandle()`,
`ProcInfo()`

GeodeFind() returns a geode's handle when given a permanent name and attributes to search on. GEOS will search the active geode list for any geode with the given name and the proper attributes set or clear.

GeodeFindResource() locates a given resource within a geode's file. It must be passed the file handle of the geode, the number of the resource in the file, and an offset within the resource at which the file read/write position should be placed. This routine may only be used on open geode files, and it returns the base offset and size of the resource. You will probably not need to use this routine.

GeodeGetInfo() takes a geode handle, a **GeodeGetInfoType** parameter, and a buffer appropriate for the return information. It returns the appropriate values as specified in the **GeodeGetInfoType** parameter. (This parameter specifies what type of information is sought about the geode; the routine can return the geode's attributes, geode type, release level, protocol, token ID, or permanent name.) The possible values of **GeodeGetInfoType** are shown below:



6.1

GGIT_ATTRIBUTES

This indicates the geode's attributes should be returned.

GGIT_TYPE This indicates the geode's type should be returned (type **GeodeType**).

GGIT_GEODE_RELEASE

This indicates the geode's release numbers should be returned.

GGIT_GEODE_PROTOCOL

This indicates the geode's protocol numbers should be returned.

GGIT_TOKEN_ID

This indicates the geode's token information should be returned.

GGIT_PERM_NAME_AND_EXT

This indicates the geode's permanent name and extender should be returned.

GGIT_PERM_NAME_ONLY

This indicates the eight characters of the geode's permanent name only should be returned, without the extender characters.

GeodeGetProcessHandle() returns the geode handle of the current process (the owner of the current thread). Another routine, **GeodeGetCodeProcessHandle()**, returns the handle of the geode that owns the code block from which it was called.

ProcInfo() returns the thread handle of the first thread of a given process.

6.1.7.3 Managing Geode Event Queues

`GeodeAllocQueue()`, `GeodeFreeQueue()`, `GeodeInfoQueue()`,
`GeodeFlushQueue()`, `ObjDispatchMessage()`, `QueueGetMessage()`,
`QueuePostMessage()`, `GeodeDispatchFromQueue()`

The following routines allocate and manage event queues. These routines are rarely called by applications as event queues are automatically managed for each thread and application.

GeodeAllocQueue() allocates an event queue and returns its handle.
GeodeInfoQueue() returns the number of events in a given event queue.
GeodeFreeQueue() frees an event queue allocated with the routine **GeodeAllocQueue()**. It must be passed the handle of the queue to be freed (unhandled events still in the queue will be discarded).

GeodeFlushQueue() flushes all events from one queue and synchronously places them all in another queue (events may not simply be tossed out).

QueueGetMessage() combined with **ObjDispatchMessage()** removes the first event from the given event queue and handles it via a callback routine. A far pointer to the callback routine in memory must be passed. Typically these will be used only by the assembly **ObjMessage()** routine used by the kernel; some other applications of this routine may be used, though. For example, the sound driver uses a note queue unassociated with objects and messages. The callback routine therefore gets the “event” (note) and pretends it’s handling a message.

6.1

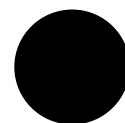
QueuePostMessage() adds an event to the specified queue.

6.1.8 Geode Protocols and Release Levels

Every GEOS geode and VM file has both a release level and a protocol level as extended attributes of the file. These two items help ease the transitions for both programmer and user when changes are made to applications, libraries, drivers, system software, etc. To control release and protocol numbers, use the GREV tool and a REV file as described in “Grev” on page 444 of “Using Tools,” Chapter 10 of the Tools Reference Manual.

6.1.8.1 Release Numbers

The release number is a **ReleaseNumber** structure which consists of four components: The *RN_major* and *RN_minor* numbers are the most significant. The *RN_change* and *RN_engineering* numbers are less significant and are used primarily to indicate non-released or running upgrade types of changes to the geode. The **ReleaseNumber** of a geode or VM file is stored in the file’s FEA_RELEASE extended attribute, and its structure is shown below:



```
typedef struct {  
    word    RN_major;  
    word    RN_minor;  
    word    RN_change;  
    word    RN_engineering;  
} ReleaseNumber;
```

6.1

The contents of the release number are up to the particular geode and are product-specific. Release numbers are not used by GEOS for compatibility checking or any other validation of files, though they are used during installation procedures.

To retrieve the release number of a given geode, use the routine **GeodeGetInfo()**. Release levels should be set at compile time and are not changeable at run-time.

6.1.8.2 Protocol Numbers

The protocol number is a structure of type **ProtocolNumber** stored in the file's FEA_PROTOCOL extended attribute. Each GEOS geode and data file has a protocol level associated with it. The protocol level is used for compatibility checking for both geodes and documents.

The **ProtocolNumber** structure consists of two parts, the major protocol and the minor protocol. This structure is shown below:

```
typedef struct {  
    word    PN_major;  
    word    PN_minor;  
} ProtocolNumber;
```

Differences in protocol levels indicate incompatibilities between two geodes, between a geode and its document format, or between a geode and its state file format. If the major protocols are different, the two items are not compatible at all (unless special provisions are made). If the minor protocol is greater than expected, some incompatibility may exist but should not affect the program. You should increment a geode's or document's protocol whenever a change is made.

If a change to a library is upward-compatible, only the minor protocol needs to be incremented. For example, if a library acquires a new function but the

library's entry points are undisturbed, the minor protocol should be incremented and the major protocol left as is. If the new function causes relocation of the entry point numbers, however, the major protocol must also be incremented.

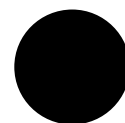
An application's protocol must be increased whenever a change will affect the application's state files. If you make a change to an application, for example, and the user has old state files, either the changes in the application can be replaced with the old information, or the state file will cause an incompatibility with unpredictable results. If the change to the application is simply functional, increment the minor protocol. If the change is to any part of a UI resource or to any other item saved to a state file, increment the major protocol. State files will be loaded if minor protocols are different and will be ignored if major protocols are different.

6.1

If an application's document format changes, you should make sure the new documents are not loaded by old applications or vice versa (unless you take the necessary conversion steps). When opening a document, you can check its protocol by checking the document file's extended attribute `FEA_PROTOCOL`. If the protocol level needs to be changed (after conversions have been done, of course), you can change them by setting `FEA_PROTOCOL`. (See "File System," Chapter 17.)

A few examples of when minor and major protocols should be incremented follow. Keep in mind that this list is by no means exhaustive.

- ◆ If you add a data structure to a document format and older versions of the application will still be able to open the document, increment the document's minor protocol only.
- ◆ If you change the instance data structure of an object that gets saved to the document, increment the major protocol, as the new methods will be using the wrong offsets to access data in the old object.
- ◆ If you add a new class to your application and don't disturb any of the other classes' entry points (i.e. add the class at the end), increment the minor protocol only.
- ◆ If you add a new class, disturbing entry points, increment the major protocol.
- ◆ If you add or delete a resource, increment the major protocol.



- ◆ If you add a chunk or object to a resource that gets saved to state files, increment the major protocol.
- ◆ If you change an object's flags (e.g. mark it ignore-dirty), increment the major protocol. Otherwise, the flags will be restored from the state file and will override the changes you made.

6.1.9 Temporary Geode Memory

6.1

```
GeodePrivAlloc(), GeodePrivFree(), GeodePrivWrite(),  
GeodePrivRead()
```

Every geode in the system has a “private data” area, a space set aside in its core block. This private data is used primarily by library geodes, when each of the library's clients uses its own copy of a particular data structure that gets manipulated by the library. The private data mechanism is used in the GEOS implementation of **malloc()**, for example (though you need not know this to use **malloc()**).

Private memory may be allocated, written to, read from, and freed by the library. The library does not have to allocate a block for each geode and maintain its own handle table; the use of the **GeodePriv...()** routines automatically manages this.

GeodePrivAlloc() reserves a given number of contiguous words for the library within the private data of all geodes in the system. The memory space is reserved but is not actually allocated for a given geode until it is used (written to); this is done for optimization purposes. This routine will return a tag pointing to where the reserved words begin. This tag is used when reading, writing, or freeing the private data. If the memory could not be allocated, the routine will return zero.

GeodePrivWrite() and **GeodePrivRead()** write to and read from the private data space. They take similar parameters: a geode handle, the tag as returned by **GeodePrivAlloc()**, the total number of words to be written or read, and a pointer to a locked or fixed buffer. In **GeodePrivWrite()**, the buffer will be passed containing the words to be written; in **GeodePrivRead()**, the buffer will be passed empty and returned containing the words read.

Typically, the geode handle passed will be zero; this indicates that the current process (which will be the library's current client) will be the owner of the private data affected. Because the library code will be executing in the thread of a given application, the application geode will be the only one having its private data affected. Thus, a library can use the same code to store different data for each geode that uses it; neither the library nor the geode needs to know that other geodes are also using the same routines.

GeodePrivFree() frees a given number of words from all geodes' private data. It needs to be passed only the number of words to be freed and the tag as returned by **GeodePrivAlloc()**.

6.2

6.2 Creating Icons

Every geode can have an icon associated with it. Typically, only applications will have special icons; other geodes (libraries and drivers) generally use one of the system icons or the icon of the application they're primarily used by.

An application's icon is stored in two places: It is defined and stored within the application's extended attributes in its.GEO file. It is also stored in a database file maintained by the UI, called the *token database*.

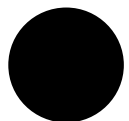
The GEOS development kit includes an icon editor tool so you can easily create icons of various color and resolution characteristics and install them into your applications.

6.2.1 The Token Database

`TokenOpenLocalTokenDB()`, `TokenCloseLocalTokenDB()`

The token database is stored in its own file. Each entry represents a single icon or series of icons that can be used with any number of files. The token database file is managed by the UI and the kernel; you should have no reason to access its internals directly, and most applications will never need to use any of the token routines except to install their document icons.

Some systems may have shared token database files; this is controlled by the INI file key *sharedTokenDatabase* in the *paths* category. Most often a shared



database file exists on a network drive and may be supplemented with a local token database. By default, if a shared database file exists, it will be opened read-only in addition to the read/write local file. You can open and close only the local database file, however, with the routines

TokenOpenLocalTokenDB() and **TokenCloseLocalTokenDB()**.

Every GEOS file has a token. The token is an index into the token database. When GeoManager scans a directory, it grabs the token from each file and searches through the token database file for it. If a match is found, GeoManager selects the proper icon and displays it; if no match is found or if the file's token is invalid, GeoManager will launch the application in engine mode and request that it install its token in the database.

For non-GEOS files, GeoManager uses the three extension characters of the file's name as a pseudo-index. For each extension (e.g. .COM, .EXE, .DOC, .BAT, etc.), GeoManager uses a single icon. Which icon is used can be set in the GEOS.INI file if a user wishes, but GeoManager will normally select the default DOS icon (of which there are two: one for executables and one for non-executables).

The index into the token database consists of two parts and is of type **GeodeToken**. This structure contains four text characters as well as the manufacturer ID number of the geode's manufacturer. The structure's definition is shown below:

```
typedef struct {
    TokenChars          GT_chars;
    ManufacturerID      GT_manufID;
} GeodeToken;

typedef char TokenChars[TOKEN_CHARS_LENGTH];
```

This structure is created and filled automatically in the geode's **.geo** file header by the Glue linker, which takes the values from the geode's geode parameters (**.gp**) file. The two fields used from the **.gp** file are *tokenchars* for the four characters and *tokenid* for the manufacturer ID.

A **GeodeToken** structure in the token database file can also be filled in when GeoManager scans a directory. If the header of a particular application's **.geo** file does not have a recognized token, GeoManager will launch the application in "engine" mode, loading its GenApplication object. It will

request that the application object then install its icon into the token database file (**GenApplicationClass** knows how to do this).

The token database file contains one entry for each token that has been installed. Each time a new token and icon is encountered, a new entry is added by the UI. This happens automatically when GeoManager scans a directory.

6.2.2 Managing the Token Database File

6.2

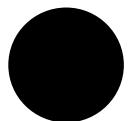
```
TokenDefineToken(), TokenGetTokenInfo(),
TokenLookupMoniker(), TokenLoadMonikerBlock(),
TokenLoadMonikerChunk(), TokenLoadMonikerBuffer(),
TokenRemoveToken(), TokenLoadTokenBlock(),
TokenLoadTokenChunk(), TokenLoadTokenBuffer(),
TokenLockTokenMoniker(), TokenUnlockTokenMoniker(),
TokenGetTokenStats(), TokenListTokens()
```

In nearly all cases, you will create your application's icon with the icon tool and not worry about it again. However, the following routines allow you to add, change, access, and remove entries from the token database.

TokenLoadTokenBlock(), **TokenLoadTokenChunk()**, and **TokenLoadTokenBuffer()** load a **TokenEntry** structure into memory (a newly allocated block, a newly allocated chunk, or a locked buffer). The **TokenEntry** structure contains information about the token, the geode's release number, and the geode's protocol number. This structure does not actually contain the monikers used for the icon.

TokenLookupMoniker() gets the specific moniker of a token entry given the display type (CGA, EGA, VGA, etc.), the entry's **GeodeToken** structure, and search flags. If the moniker is found, the entry identifier (database group and ID numbers) of the moniker are returned. You can use these return values to lock the moniker into memory (see below).

TokenLockTokenMoniker() locks a moniker into memory given its entry identifier. This routine returns a pointer to a locked block and the chunk handle of the chunk containing the locked moniker. A moniker should always be locked before it is drawn; this keeps it from moving in memory while it is being accessed. The routine **TokenUnlockTokenMoniker()** unlocks a



previously locked moniker when given the moniker's segment address. This unlocks the entire block, not just the individual moniker.

TokenLoadMonikerBlock(), **TokenLoadMonikerChunk()**, and **TokenLoadMonikerBuffer()** load a specific moniker into memory from the token database file (into a newly allocated block, a newly allocated chunk, or a locked buffer). It takes the same parameters as **TokenLookupMoniker()** but returns the handle and chunk handle of the loaded moniker. If using this routine, simply lock the memory block rather than using **TokenLockTokenMoniker()**.

TokenGetTokenInfo() finds a token when passed its *tokenchars* and *tokenid* and returns the token's flags. If no token exists with the passed characteristics, it will return an error flag.

TokenDefineToken() adds a new token and its moniker list to the token database. If the given token already exists, the new one will replace the old one. The token identifier (*tokenchars*, *tokenid*), handle and chunk handle of the moniker list, and flags of the new token must be passed.

TokenRemoveToken() removes a given token and its moniker list from the token database file. It returns only a flag indicating whether the token was successfully removed or not.

TokenListTokens() returns a list of the tokens in the token database. It is passed three arguments:

- ◆ A set of **TokenRangeFlags**, which specifies which tokens should be returned. The following flags are available:

TRF_ONLY_GSTRING

Return only those tokens which are defined with a GString.

TRF_ONLY_PASSED_MANUFID

Return only those tokens which match the passed manufacturer's ID.

- ◆ A number of bytes to reserve for a header.
- ◆ A manufacturer ID. This field is ignored if TRF_ONLY_PASSED_MANUFID was not passed.

TokenListTokens() allocates a global memory block and copies all the specified tokens into that block. It leaves a blank space at the beginning of

the block; this space is the size specified by the second argument. The rest of the block is an array of **GeodeToken** structures. **TokenListTokens()** returns a dword. The lower word of the return value is the handle of the global memory block; the upper word is the number of **GeodeToken** structures in that block.

6.3 Saving User Options

6.3

Almost all users enjoy configuring their systems to their own tastes, whether it's setting the background bitmap or choosing a default font. Most applications will, therefore, want to provide a way for the user to choose and save various options for the application.

Generic objects have this capability built in. If you set them up for saving options, they will automatically set and maintain the saved options, making sure the options are set properly whenever the application is launched. However, sometimes you will want to set additional options not managed by UI objects. This is not difficult to do in GEOS: You can have your application save its options directly to the local initialization file, GEOS.INI.

6.3.1 Saving Generic Object Options

All appropriate generic UI objects have the ability to save their options. For example, a properties GenInteraction could save which of its options are on and which are off when the user presses a "save options" trigger.

To save generic object options, you have to do two basic things:

- ◆ Create an Options menu
Typically, you will create a menu for user options. This menu should be declared of type GIGT_OPTIONS_MENU as shown in Code Display 6-2.
- ◆ Use a special GCN list
All objects that want to save options must be put on a special GCN list in the application's GenApplication object. The list type should be either GAGCNLT_STARTUP_LOAD_OPTIONS (if the options should be loaded at the time of application startup) or GAGCNLT_SELF_LOAD_OPTIONS (if

the options should be loaded when the object deems it necessary, typically when the object is first displayed).

Code Display 6-2 shows an example of how objects should be declared for saving their options.

Code Display 6-2 Saving Generic Object Options

6.3

```
/* The GenApplication object must declare a GCN list of the type appropriate for
 * the options being saved (or both types if the application uses both types).
 * This GenApplication declares a list of objects whose options do not have to be
 * loaded at startup. */

@object GenApplicationClass SampleApp = {
    GI_visMoniker = "Sample Application";
    GI_comp = SamplePrimary; /* Primary window object is the only child. */
    gcnList(MANUFACTURER_ID_GEOWORKS, GAGCNLT_WINDOWS) = SamplePrimary;
    /* The above list is to declare windowed objects that must appear
     * when the application is opened and made usable. */
    gcnList(MANUFACTURER_ID_GEOWORKS, GAGCNLT_SELF_LOAD_OPTIONS) =
        SampleController;
    /* The above list is used for generic objects that save their
     * options but do not need their options loaded at startup. */
}

/* Some applications that have generic objects save their own options might not
 * have a special Options menu but may just have a trigger somewhere for saving
 * options. In any case, the "save options" trigger sends MSG_META_SAVE_OPTIONS to
 * its GenApplication object. If you use an Options menu with GIGT_OPTIONS_MENU
 * set, this will automatically be built into the menu.
 * The SampleOptionsMenu is a child of SamplePrimary, not shown. */

@object GenInteractionClass SampleOptionsMenu = {
    GI_comp = SampleToolbox, SampleToolControl, SampleSaveOptsTrigger;
    GII_visibility = GIV_POPUP; /* Make it a menu. */
    ATTR_GEN_INTERACTION_GROUP_TYPE = (GIGT_OPTIONS_MENU);
}

/* The other objects (controllers) are not shown here. Just the "save options"
 * trigger, which sends MSG_META_SAVE_OPTIONS to the GenApplication object. */
```

```
@object GenTriggerClass SampleSaveOptsTrigger = {  
    GI_visMoniker = 'S', "Save Options";  
    GTI_destination = SampleApp;  
    GTI_actionMsg = MSG_META_SAVE_OPTIONS;  
}
```

6.3.2 The GEOS.INI File

6.3

GEOS can use multiple initialization files in network situations, but only the local GEOS.INI is editable. The local GEOS.INI (referred to hereafter simply as GEOS.INI or “the INI file”) is read first into a moveable, swappable buffer, and it contains the path names of any other INI files to be used. The other INI files are subsequently loaded into other buffers. When the kernel searches for a specific entry in one of the INI files, it looks first in the local INI’s buffer and then in any others in the order they were loaded. When it reaches what it wants, it stops searching. Thus, multiple entries are allowed but are not used, and the local INI file has precedence over all others.

6.3.2.1 Configuration of the INI File

The GEOS.INI file has a very specific format and syntax. It is segmented into *categories*, each of which contains several *keys* that determine how GEOS will perform. (For an example of an INI file, see Code Display 6-3.) Note that typically the GEOS.INI file should not be accessed directly because its format is subject to change.

A category is essentially a group of keys. Certain geodes will work with their own categories, and certain categories will be used by several geodes. For example, the *system* category is used by the kernel and the UI to determine several of the default system settings. Each category is set in GEOS.INI by putting its name within square brackets. The text within the brackets is case-insensitive and ignores white space, so [My category], [mycategory], and [MY CATEGORY] are all equivalent.

A key is any setting that the system or an application can recognize and assign a value to. A single key may exist in several different categories; since both category and key define an entry, the entries will be unique if either key

or category is different. Keys may be text strings, integers, or Boolean values, and every key is identified by its name, which is an ASCII string. Data may also be read from and written to the file in binary form; the kernel will automatically convert this into ASCII hexadecimal for storage and will revert it to binary during retrieval.

Both category names and key fields are called “entries.” Each entry may exist on a single line or may cover several lines. An entry that contains carriage returns is technically known as a “blob.” Each blob will automatically be enclosed in curly braces when it is written into the file. Any blob that contains curly braces will automatically have backslashes inserted before the closing braces so GEOS doesn’t mistake them for the blob delimiters.

Comments may be added to the INI file by putting a semicolon at the beginning of the line. The file has several standard categories and keys that can be set within them. These are detailed in “The INI File,” Chapter 9 of the Tools Reference Manual.

Code Display 6-3 Example GEOS.INI File Entries

```
; The category "system" is used by the kernel and the UI to set certain system
; defaults such as the number of handles, the default system font and size, and
; the types of memory drivers to be loaded.

[system]

; The handles key is assigned an integer that determines the number of handle
; spaces allocated for the system's Handle Table.
handles = 2500

; The font key is assigned a character string that represents a file name
; or a list of file names separated by spaces.
; The listed file(s) will be read in as the font driver geode.
font = nimbus.geo

; The memory key is assigned a blob of text containing all the names of the memory
; drivers available to the system.
memory = {
disk.geo
emm.geo
xms.geo
}

; The category "My App's Category" is set for example only. It is used by the MyApp
; application.
```

```
[My App's Category]

; The myappHiScore key is an integer key set by the MyApp application.
myappHiScore = 52

; The myappBoolean key is a Boolean value. Booleans are case-insensitive, so True,
; true, and TRUE are all equated to "true". This is actually a Boolean value and
; is translated by the read and write routines that work with Boolean values.
myappBoolean = true

; The myappHiName is a text string that, in this case, contains carriage returns,
; backslash characters, and curly brace characters. The original text looked like
; this:
;
;         this is a multi-line
;         blob of text with curly
;         brace ({,}) characters }} in it
; It is automatically given backslashes in front of the closing braces, and it
; is automatically surrounded with curly braces.
myappHiName = {this is a multi-line
blob of text with curly
brace ({,\}) characters \\\} in it}
```

6.3

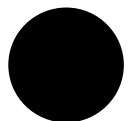
6.3.2.2 Managing the INI File

```
InitFileSave(), InitFileRevert(),
InitFileGetTimeLastModified(), InitFileCommit()
```

Because the INI file is common to all geodes and to all threads in the system, only one thread at a time may access it. This synchronization is handled by the kernel whenever a routine to read from or write to the INI file is used. Additionally, the INI file is loaded into a buffer when the system is first run; all operations on the INI file actually work on this buffer, and the buffer may be flushed to disk only by the kernel.

There are, however, four routines that work directly on the INI file. One saves the file, another reverts it from the last save, the third checks the time the file was last modified, and the fourth commits any pending INI file changes to disk.

To save the local GEOS.INI, use the routine **InitFileSave()**; this saves the changes to the backup file. It requires no parameters and returns an error flag if the file could not be saved. **InitFileRevert()** reverts the GEOS.INI file



to its state the last time it was backed up. This routine takes no parameters, and it returns an error flag if the revert can not be accomplished.

InitFileGetTimeLastModified() returns the system counter's value stored the last time GEOS.INI was modified.

InitFileCommit() takes all the changes made since the file was last modified and flushes them to disk. This commits all the changes and should be used only from the kernel. It should not be used by applications.

6.3

6.3.2.3 Writing Data to the INI File

```
InitFileWriteData(), InitFileWriteString(),  
InitFileWriteStringSection(), InitFileWriteInteger(),  
InitFileWriteBoolean()
```

GEOS provides five routines to write to the INI file, one for each of the allowable data types. Each of these routines will first gain exclusive access to the local INI buffer, then locate the appropriate category for writing. After writing the key and its value, the routine will relinquish exclusive access to the buffer, allowing other threads to write into it. Writing a category or key that does not exist in the local INI file will add it to the file.

Each of these routines takes at least three arguments: The category is specified as a null-terminated character string; a pointer to the string is passed. The key name is also specified as a null-terminated character string; again, a pointer to the string is passed. The third parameter is specific to the routine and contains the value to which the key will be set.

InitFileWriteData() writes a number of bytes into the INI buffer, and it takes four parameters. The additional parameter is the size of the data. The data will be converted into ASCII hexadecimal when the file is saved and will be converted back when the key is read.

InitFileWriteString() takes a pointer to the null-terminated character string to be written. If the character string contains carriage returns or line feeds, it will automatically be converted into a blob.

InitFileWriteStringSection() writes a new string section (a portion of a blob) into the specified entry. The specified entry must be a blob already, and

the string section will be appended to the blob. A string section is a line of a blob delineated by line feeds or carriage returns.

InitFileWriteInteger() takes as its third argument the integer to be written.

InitFileWriteBoolean() takes a Boolean value. A zero value represents false, and any nonzero value represents true. When looking at the INI file with a text editor, the Boolean value will appear as a text string of either “true” or “false”; it will, however, be interpreted as a Boolean rather than a text string.

6.3

6.3.2.4 Getting Data from the INI File

```
InitFileReadDataBuffer(), InitFileReadDataBlock(),
InitFileReadStringBuffer(), InitFileReadStringBlock(),
InitFileEnumStringSection(),
InitFileReadStringSectionBuffer(),
InitFileReadStringSectionBlock(), InitFileReadInteger(),
InitFileReadBoolean()
```

When you want to check what a key is set to in the INI file, you should use one of the **InitFileRead...()** routines. These search the local INI file first and then each of the additional INI files in the order they were loaded. They will return the first occurrence of a given key, so if the key exists in both your local INI file and another INI file, these routines will return only the local value.

All of these routines take at least two parameters. The first is the category of the entry to be retrieved; this is stored as a null-terminated ASCII string, and a pointer to the string is passed. The second is the key of the entry. This, too, is stored as a null-terminated ASCII string, and a pointer to the string is passed.

InitFileReadBoolean() returns the Boolean value of the given key. If the key is set “false,” a value of zero (FALSE) will be returned. If the key is set “true,” a nonzero value will be returned (-1, the constant TRUE).

InitFileReadInteger() returns the integer value of the given key.

InitFileReadDataBuffer() and **InitFileReadDataBlock()** both return the data bytes stored in the given key. The first, however, takes the address

of a buffer already allocated and puts the data into the buffer. The second allocates a new block on the heap and puts the data into it. If you don't know the size of the data, you should use **InitFileReadDataBlock()**.

InitFileReadStringBuffer() and **InitFileReadStringBlock()** both return the null-terminated string stored in the given key. In both cases, curly braces will be stripped off of blobs and backslash characters will be removed if appropriate. The first, however, takes the address of a buffer already allocated and puts the string in the buffer. The second allocates a new block on the heap and returns the string in it. If you don't know the approximate size of the string, use **InitFileReadStringBlock()**.

InitFileReadStringSectionBuffer() and its counterpart **InitFileReadStringSectionBlock()** both return a null-terminated section of the string stored in the given key. A string section is defined as any number of contiguous printable ASCII characters and is delimited by carriage returns or line feeds. These routines take the number of the string section desired and return the section (if it exists). **InitFileReadStringSectionBuffer()** takes the address of a buffer already allocated on the heap and returns the string section in the buffer. **InitFileReadStringSectionBlock()** allocates a new block on the heap and returns the string section in it. You should use this routine if you don't know the approximate size of the string section.

InitFileEnumStringSection() enumerates the specified blob, executing a specified callback routine on each string section within the blob.

6.3.2.5 Deleting Items from the INI File

```
InitFileDeleteEntry(), InitFileDeleteCategory(),  
InitFileDeleteStringSection()
```

Besides reading and writing data, you can delete categories and keys that were previously entered. **InitFileDeleteEntry()** takes pointers to both the null-terminated category name and the null-terminated key name and deletes the entry. **InitFileDeleteCategory()** takes only a pointer to the null-terminated category name and deletes the entire category, including all the keys stored under it.

In addition, you can delete a single string section from a specified blob. **InitFileDeleteStringSection()** takes the category and key names of the blob as well as the index of the string section, and it deletes the string section.

If the string section does not exist or if either the key or category can not be found, the routine will return an error flag.

6.4 General System Utilities

The kernel provides a number of routines that fulfill general needs for system utilities. These range from setting the system's date and time to retrieving the amount of swap memory used to shutting down the system in order to execute a DOS-based program.

6.4

6.4.1 Changing the System Clock

`TimerGetDateAndTime()`, `TimerSetDateAndTime()`

The system clock reflects the current date and time of day.

TimerGetDateAndTime() takes a pointer to a **TimerDateAndTime** structure and fills it with the current year, month, day, day of week, hour, minute, and second. **TimerSetDateAndTime()** sets the current date and time to the values in the passed structure. It is unusual for any application other than the Preferences Manager to change the system clock.

6.4.2 Using Timers

`TimerStart()`, `TimerStop()`, `TimerSleep()`, `TimerGetCount()`

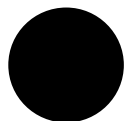
If your geode needs an action to happen on a timed basis, you will want to use a timer. GEOS lets you set up timers that will call a routine or send a message after a given interval has passed. GEOS offers four types of timers:

- ◆ One-shot

A one-shot timer counts a certain number of "ticks." (There are 60 ticks in a second.) It will call a routine or send a message after the specified number of ticks have been counted.

- ◆ Continual

A continual timer counts a certain number of ticks and then resets and



counts again. Each time it reaches the specified number of ticks, it calls a routine or sends a message.

◆ Millisecond

A millisecond timer is a one-shot timer that counts in milliseconds rather than ticks and calls a routine when time is up (it does not send a message).

◆ Sleep

A sleep timer causes the thread that creates it to sleep for a certain number of ticks. The sleep timer is unlike the others in that it does not send a message or call a routine when time is up; it only awakens the sleeping thread.

◆ Real-time

A real-time timer is passed the number of days since 1980, the hour, and the minute of the event. On hardware platforms that support such a thing, GEOS will wake up when the real-time timer goes off. (All other timers are ignored under these conditions.)

All timers except sleep timers are created and started with the routine **TimerStart()**, and continual timers can be destroyed with **TimerStop()**. **TimerStop()** may also be used to prematurely stop a one-shot timer, though if the one-shot sent a message, the message could be in the recipient's queue even after **TimerStop()** was called. Sleep timers are created and started with **TimerSleep()**, and they do not need to be stopped.

An additional routine, **TimerGetCount()**, returns the current system counter. The system counter contains the number of ticks counted since GEOS was started.

6.4.3 System Statistics and Utilities

```
SysStatistics(), SysGetInfo(), SysGetConfig(),  
SysGetPenMode(), SysGetDosEnvironment()
```

Occasionally, a geode will need to query the kernel about the state or configuration of the system. GEOS offers five routines for this:

SysStatistics() returns a structure of type **SysStats**. This structure contains information about how busy the CPU is, how much swap activity is going on, how many context switches occurred in the last second, how many

interrupts occurred in the last second, and how many runnable threads exist. You can not set or otherwise alter this structure.

SysGetInfo() returns a particular statistic about the current system status. It can return a number of different statistics including the CPU speed, the total number of handles in the handle table, the total heap size, the total number of geodes running, and the size of the largest free block on the heap. Most of the information your application can request with this routine can be garnered from the Perf performance meter; it is easiest to use Perf when debugging, though a few geodes will need to know this information.

6.4

SysGetConfig() returns information about the current system's configuration including the processor type and machine type. It also returns flags about the particular session of GEOS including whether this session was started with Swat, whether a coprocessor exists, and other information.

SysGetPenMode() returns TRUE if GEOS is currently running on a pen-based machine.

SysGetDosEnvironment() returns the value of a DOS environment variable. It is passed a string representing the variable name and returns a string representing the value.

6.4.4 Shutting the System Down

`DosExec()`, `SysShutdown()`

There are two ways for a geode to shut down GEOS. The first, **DosExec()**, shuts the system down to run a program under DOS, returning after the DOS program has finished—unless a task-switch driver is in use, in which case the system will create a new task and cause the task-switcher to switch to the new task. The second, **SysShutdown()**, forces the system to shut itself down completely. Neither of these routines is commonly used by anything other than the kernel, GeoManager, special “launcher” programs, or the UI. Their use by other libraries or applications is discouraged unless absolutely necessary.

DosExec() takes several parameters including the pathname of the DOS program to be run, arguments for the program, an optional disk handle of the disk that contains the program to be run, the optional directory and disk

handle in which the program should be executed, and a record of **DosExecFlags**. If the return value is nonzero, an error occurred in loading the DOS program, and you can use **ThreadGetError()** to check what error occurred. Note that **DosExec()** always returns. Applications should *not* rely on **DosExec()** shutting the system down; if a task switcher is present, GEOS will be swapped out rather than shut down.

6.5

SysShutdown() causes GEOS to exit in one of several ways. This routine should be passed a shutdown mode. If the mode is `SST_CLEAN`, `SST_RESTART`, `SST_SUSPEND`, or `SST_CLEAN_FORCED`, the routine will return; otherwise, it will not return and the shutdown will commence. If `SST_CLEAN` is passed, the shutdown may be aborted after **SysShutdown()** returns. You can have **SysShutdown()** cause GEOS to reboot itself after shutting down (as the Preferences Manager application does for certain preferences settings), but this starts GEOS fresh. This routine is very rarely used by anything other than the UI, the kernel, or the Preferences Manager application.

If something else (typically the UI or task switcher) shuts the system down, objects that register for shutdown notification will receive `MSG_META_CONFIRM_SHUTDOWN`. The application should call **SysShutdown()** with the mode `SST_CONFIRM_START`; this allows the object to have exclusive rights for asking the user to confirm the shutdown (when the object is finished with the user interaction, it can call **SysShutdown()** with `SST_CONFIRM_END` to release exclusive access). This is useful if your application or library has an ongoing operation and wants to verify the shutdown with the user.

6.5 The Error-Checking Version

GEOS has two versions of its system software. The normal version as shipped retail is the “non-error-checking” version. The other, used for debugging applications and other geodes, is called the “error-checking” version, or *EC version*. Nearly all components of the system software exist in both versions—the kernel, the UI, libraries, drivers, etc.

Together, Swat and the EC version provide extensive and superb debugging power. The EC version allows you to call special error-checking routines to

check the integrity of handles, resources, memory, files, and other things. These error-checking routines all begin with EC; for more information, see their entries in the Routine Reference Book. In addition to the full error checking provided by the EC versions of the system geodes, you can add your own error-checking code to your programs.

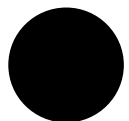
6.5.1 Adding EC Code to Your Program

6.5

When compiling your code, you can include certain extra lines in either the EC version or the non-EC version of your program. For example, during debugging you want your program to check the validity of each memory handle before locking the memory block; for the final version of your program, however, you don't want this validity check because it slows down your program. You can easily add a few instructions that will be compiled only into the EC version.

GEOS provides five macros, listed below, for version-specific instructions. These macros must be treated as statements; they may not be used in expressions.

- | | |
|-------------|--|
| EC | This macro adds the specified line of code to the EC version only. When the non-EC version is compiled from the same code, the line will be left out. |
| EC_ERROR | This macro halts the execution of the EC version of a program by calling FatalError() with a specified error code. The call to FatalError() will not be included in the non-EC version. |
| EC_ERROR_IF | This macro works like EC_ERROR, above, but it also allows you to set a condition for whether FatalError() will be called. If the condition is met, FatalError() will be called. |
| NEC | This macro adds the specified line of code to the non-EC version but leaves it out of the EC version. (It is the converse of the EC macro.) |
| EC_BOUNDS | This macro checks the validity of a specified address (pointer) by calling the ECCheckBounds() routine. If the pointer is out of bounds, ECCheckBounds() will call FatalError() . This macro may only add the bounds check to the EC version. |



An example of use of these macros is shown in Code Display 6-4.

Code Display 6-4 EC Macros

```
/* This code display shows only the usage of these macros; assume that each
 * line shown below exists within a particular function or method. */

/* The EC macro adds a line of code to the EC version. Its format is
 * EC(line) where line is the line of code to be added.
6.5 * Note that the NEC macro is similar. */
    EC( @call MyErrorDialogBox::MSG_MY_ERR_PRINT_ERROR(); )
    NEC( @call MyErrorDialogBox::MSG_MY_ERR_PRINT_NO_ERROR(); )

/* The EC_ERROR macro adds a call to FatalError() to the EC version. Its format is
 * EC_ERROR(code) where code is the error code to be called. */
    EC_ERROR(ERROR_ATTR_NOT_FOUND)

/* The EC_ERROR_If macro is similar to EC_ERROR but is conditional. Its format is
 * EC_ERROR_IF(test, code) where test is a Boolean value and code is the
 * error code to be called. */
    lockVariable = MyAppCheckIfLocked(); /* TRUE if inaccessible. */
    EC_ERROR_IF(lockVariable, ERROR_ACCESS_DENIED)/* Error if inaccessible. */

/* The EC_BOUNDS macro adds a call to ECCheckBounds() to the EC version.
 * Its format is
 * EC_BOUNDS(addr) where addr is the address to be checked. */
    myPointer = MyAppGetMyPointer();
    EC_BOUNDS(myPointer)
```

6.5.2 Special EC Routines

```
SysGetECLevel(), SysSetECLevel(), SysNotify(), EC...(),
CFatalError(), CWarningNotice()
```

While using Swat and the EC version, you can set and retrieve the current level of error checking employed. The level is set with a record of flags, each of which determines whether a certain type of checking is turned on. This record is of type **ErrorCheckingFlags**. You can retrieve it with **SysGetECLevel()** and set it with **SysSetECLevel()**.

SysNotify() puts up the standard error dialog box—white with a black frame. This routine is available in both the EC and the non-EC versions of the kernel. This is the box used by the kernel to present unrecoverable errors to the user. You can also call it up and allow the user any of five options: retry, abort, continue, reboot, or exit. Usually, this dialog box is used only for errors, but it can be used for other user notification as well. (Note that with the “exit” and “reboot” options, this routine will *not* return. Note also that only certain combinations of the five options are supported.)

The **CFatalError()** and **CWarningNotice()** routines provide run-time and compile-time error or warning messages. The **CFatalError()** routine calls the kernel's **FatalError** function, which puts up an error dialog box and, in Swat, causes Swat to hit a breakpoint so you can debug the error.

CWarningNotice() may be put in error-checking code to make the compiler put up a compile-time warning note.

6.6

6.6 Inter-Application Communication



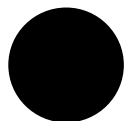
Advanced Topic

Not many applications will need to use inter-application communication.

Applications do not usually need to communicate with each other. Most applications will interact only with the kernel and with libraries; such applications don't even notice if any other applications are running. However, a few applications will need to communicate with other applications.

For example, many desktop-managers provide a “print file” command; the user selects a file's icon, then chooses this command to print out the appropriate file. The desktop manager will have no way of knowing how to print the file; after all, the file could have been created by an application which hadn't even been written when the desktop manager was installed. The desktop manager therefore causes the appropriate application to be started. It then instructs the application to print the file.

Because GEOS is an object-oriented system, there is a natural model for inter-application communication. When one application needs to contact another, it can simply send a message; thus, sending information or instructions to another application is not, in principle, different from sending it from one object to another within an application.



The GEOS Inter-Application Communications Protocol (*IACP*) specifies how to open communications with another application, and how to send messages back and forth.

6.6.1 IACP Overview

6.6

There is a major difference between sending a message within an application, and sending one to a different application. When you send a message from one object to another within an application, you know that the recipient exists, and you know the *optr* of that recipient. This makes it easy to send messages.

When you send a message to another application, however, you do not (at first) know any *optrs* to that application. In fact, you may not even know that the application is running. Often, all you will know is something like, “I want to send a message to **SpiffyWrite**”.

GEOS uses a client-server model of inter-application communication. Every **GeodeToken** corresponds to a server. Whenever an application is launched, GEOS checks to see if there is a *server-list* corresponding to the application's token. If there is, GEOS adds the app's Application object to the server-list; if there is not, GEOS creates a server-list and adds the Application object to that list.

For example, suppose the user launches a single copy of **SpiffyWrite**; this application has a manufacturer-ID of `MANUFACTURER_ID_SPIFFYWARE`, and the token characters “SWRI”. GEOS will check if there's a server-list for that **GeodeToken**. Let us suppose there isn't such a list; GEOS will automatically create one, and add **SpiffyWrite**'s Application object to that list. The Application object is now said to be a *server* for the list.

Now let us suppose another application needs to contact **SpiffyWrite**; for example, perhaps a desktop program needs to print a **SpiffyWrite** file. It tells the kernel that it would like to be a *client* on the list for the token “{`MANUFACTURER_ID_SPIFFYWARE`, “SWRI”}”. GEOS will check to see if a server-list for that token exists. If so, it will add the client to that list; this will cause a notification message (`MSG_META_IACP_NEW_CONNECTION`) to be sent to every server for that list.

Once a client is linked to a server, it can send a message to the server list. It does this by encapsulating a message, then passing the encapsulated message to the server-list. GEOS will dispatch the message to every server for the list; the server objects will receive it just like any ordinary message. (It actually passes the encapsulated message to the server object as an argument to MSG_META_IACP_PROCESS_MESSAGE; the server can then dispatch the message to the final recipient.)

This establishes the link between the applications. The client can pass an optr to the servers by putting it in the encapsulated message; a server object can then send messages straight to a particular object.

6.6

When a client no longer needs to communicate with a server, it unregisters itself from the server list. GEOS then sends a notification message to every server object.

Customarily, whoever allocates a global resource must also free it. For example, if a client might pass information to the server by allocating a global block, writing the data to the block, and passing the block to the server. The server should notify the client when the server is finished with that data; the client can then free the block. Similarly, if a server allocates a block to pass information to a client, the server should free the block.

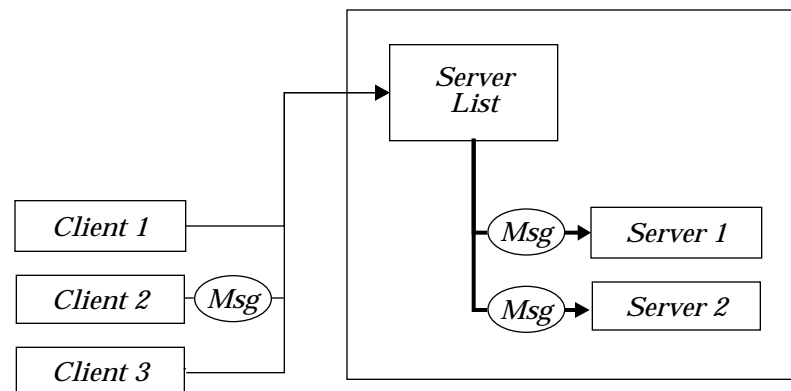
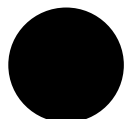


Figure 6-1 *IACP Clients and Servers*

A client can encapsulate a message, then use an IACP routine to instruct the kernel to send the message to the servers for a server list. There may be several servers on any given server list.



There may be several servers for a given server list. For example, if three copies of **SpiffyWrite** were running at once, each of their Application objects would be a server for the same server-list. Furthermore, any object can make itself a server for any list. All servers will receive copies of every message sent to the server list. To distinguish between different servers for a list, every server for a list is assigned a distinct **ServerNumber**. If it chooses to, a client can specify that a message be sent only to the server with a specific number.

6.6

6.6.2 GenApplicationClass Behavior

GenApplicationClass is built to support IACP automatically. If a server or client object is subclassed from **GenApplicationClass**, most of the work of supporting IACP is done transparently to the application writer. The following capabilities are built in:

- ◆ The Application Object automatically registers itself as a server for the appropriate list when it is launched in application mode. If it is launched in engine mode, it registers itself when it receives `MSG_META_APP_STARTUP`. If the last client-connection to the application is closed, and the Application is not currently running in application mode, the Application object will shut down automatically.
- ◆ When the Application object registers itself as a server for its own list, it sends itself `MSG_GEN_APPLICATION_IACP_REGISTER`. An application may subclass this if it wants to take other action at this time (e.g. registering itself for other lists). Similarly, when an Application object unregisters itself from its own server list, it sends itself `MSG_GEN_APPLICATION_IACP_UNREGISTER`.
- ◆ The Application object automatically handles the `MSG_META_IACP...` messages appropriately. In particular, when the kernel passes an encapsulated message to an Application object with `MSG_META_IACP_PROCESS_MESSAGE`, the Application object automatically dispatches the message to the appropriate location.
- ◆ An Application object will refuse to quit as long as any client has an open IACP connection to it. (It can, however, be forcibly detached; this happens when the system is shut down, as noted below.). In such a case, the Application object will automatically call **IACPShutdownAll()** to shut

down all IACP links it has open, whether it is a client or a server on those links.

- ◆ When a link is closed, IACP automatically sends `MSG_META_IACP_LOST_CONNECTION` to all objects on the other side of the link. When an Application object receives this message, it waits until all remaining messages from the link have been handled; it then calls **IACPShutdown()** for that connection. It also forwards this message to all Document objects, so a Document object will know to close itself if the IACP connection was the only reference to it. Again, the Application object does this whether it is a client or a server.
- ◆ If the Application object is forcibly detached, it sends itself `MSG_GEN_APPLICATIONIACP_SHUTDOWN_ALL_CONNECTIONS`. The default handler for this message will call **IACPShutdownAll()** to shut down all IACP links the Application object has open, whether it is a client or a server on those links. You can subclass this message if you need to take some additional action when the IACP connections are severed.

6.6

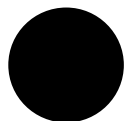
6.6.3 Messages Across an IACP Link

`IACPSendMessage()`, `IACPSendMessageToServer()`

Either a client or a server may send messages over an IACP link. Both clients and servers use the same technique. The message sender encapsulates a message, and passes the encapsulated message to **IACPSendMessage()**. **IACPSendMessage()** dispatches the message to every object on the other side of the link. For example, if a client passes a message to **IACPSendMessage()**, that message will be dispatched to every server object for the specified list.

IACPSendMessage() is passed five arguments:

- ◆ The token for the IACP link.
- ◆ The **EventHandle** of an encapsulated message.
- ◆ The **TravelOption** for that message.
- ◆ An encapsulated completion message; this will be dispatched once each time the first message has been successfully handled.



- ◆ A member of the **IACPSide** enumerated type. This tells whether the message is being sent by a client or a server. If you pass the value **IACPS_CLIENT**, the message will be dispatched to all servers; if you pass **IACPS_SERVER**, the message will be dispatched to all clients.

The message will be dispatched to all geodes on the other side of a link. Note that a client need not send the message to the server object per se. It can use the travel options field to direct the message anywhere within the server object's geode. It can also specify the **optr** of the recipient when it encapsulates the message; in this case, it should pass a **TravelOption** of -1.

6.6

Every time the encapsulated message is successfully handled, the "completion message" will be dispatched. Typically, the completion message is addressed to the object that called **IACPSendMessage()**, instructing it to free global resources that had been allocated for the message.

The routine returns the number of messages that were dispatched. This lets the sender know how many completion messages to expect, and lets it properly initialize all reference counts to global resources.

A client may choose to send a message to a specific server. It can do this by calling **IACPSendMessageToServer()**. This takes almost the same arguments as **IACPSendMessage()**. However, instead of being passed an **IACPSide** value, it is passed a server number. GEOS will dispatch a single copy of the message to the specified server. **IACPSendMessageToServer()** returns the number of times the message was dispatched. This will ordinarily be one; however, if the specified server is no longer registered, it will be zero.

6.6.4 Being a Client

Any object can register as a client for an IACP server list. When an object is a client, it can send messages to the server list, which will pass them along to the servers for that list.

6.6.4.1 Registering as a Client

```
IACPConnect(), IACPCreateDefaultLaunchBlock()
```

To register as a client for a list, call the routine **IACPConnect()**. When you call this routine, you specify which server list you are interested in. If there is no such server list running, you can instruct the kernel to start up the server list, as well as one of the default applications for that list.

IACPConnect() is passed five arguments:

6.6

- ◆ The **GeodeToken** of the list for which you want to register.
- ◆ A set of **IACPConnectFlags**. The following flags are available:

IACPF_OBEY_LAUNCH_MODEL

This indicates that if GEOS should follow the launch model, which will specify whether the user should be presented a dialog box, asking the user whether an existing application should be used as the server, or a new application launched. If you set this flag, you must pass an **AppLaunchBlock()**, with the *ALB_appMode* field set to **MSG_GEN_PROCESS_OPEN_APPLICATION**; you must also set **IACPF_SERVER_MODE** to **IACPSM_USER_INTERACTIBLE**.

IACPF_CLIENT_OD_SPECIFIED

This flag indicates that you will specify what object will be the new client. If you do not set this flag, the sending geode's Application object will be registered as the client.

IACPF_FIRST_ONLY

If you pass this flag, the client will be connected to only the first server on the server list. Any messages sent by the client to the server list will be passed only to that one server.

IACPSM_SERVER_MODE

This field is three bits wide; it holds a member of the **IACPServerMode** enumerated type. This type specifies how the client expects the server to behave. Currently, only two types are supported:

IACPSM_NOT_USER_INTERACTIBLE

This is equal to zero. It indicates that the server object need not interact with the user.



IACPSM_USER_INTERACTIBLE

This is equal to two. It indicates that the server should be able to interact with the user like any normal application.

- ◆ The **MemHandle** of an **AppLaunchBlock**. If the server you specify is not running, GEOS will launch the application specified by the **AppLaunchBlock**. If you pass a null **MemHandle**, GEOS will return an error if no server is running.
- ◆ The **optr** of the client object. This is ignored if **IACPF_CLIENT_OD_SPECIFIED** was not passed.
- ◆ A pointer to a word. **IACPConnect()** will write the number of servers on the list to that word.

6.6

IACPConnect() returns a word-sized **IACPConnection** token. You will need to pass that token when you call another IACP routine to use the connection. It will also return the number of server objects on the list; it returns this value by writing it to the address indicated by the pointer passed.

If the server list you indicate is not currently running, IACP may do one of two different things. If you pass a null handle as the third argument, **IACPConnect()** will fail. It will return the error value **IACP_NO_CONNECTION**, and indicate that there are no servers on the specified list.

If you pass an **AppLaunchBlock**, **IACPConnect()** will examine that launch block to see what application should be launched to act as a server. The **AppLaunchBlock** should specify the location and name of the application to open. If the *ALB_appRef.AIR_diskHandle* field is non-zero, **IACPConnect()** will look in the specified disk or standard path for an application with the right **GeodeToken**; otherwise, it will look in the standard places for an application.

Note that if you pass a launch block to **IACPConnect()**, you may *not* alter or free it afterwards. If the application is created, the block you pass will be its launch block; if not, the kernel will free the block automatically. In any event, the caller no longer has access to the block.

If **IACPConnect()** launches an application, the caller will block until that application has been created and registers for the server list. If the application does not register for that list, the caller will never unblock. You

must therefore make sure that you are launching the right application for the list. Note that every application object will automatically register for the server list which shares its token.

To create a launch block, you should call **IACPCreateDefaultLaunchBlock()**. This routine is passed a single argument, which specifies how the application will be opened. That argument must be `MSG_GEN_PROCESS_OPEN_APPLICATION` (the application will be opened as a standard, user-interactable application); `MSG_GEN_PROCESS_OPEN_ENGINE` (the application will be opened in engine mode, i.e. with no user interface); or `MSG_GEN_PROCESS_OPEN_CUSTOM` (which has an application-specified meaning).

6.6

IACPCreateDefaultLaunchBlock() allocates a launch block and sets up its fields appropriately. As created, the launch block will have the following characteristics:

- ◆ The application's initial working directory (i.e. the launch block's *ALB_diskHandle* field) will be `SP_DOCUMENT`.
- ◆ No application directory will be specified in the launch block (i.e. *ALB_appRef.AIR_diskHandle* will be zero); **IACPConnect()** will attempt to find the application on its own.
- ◆ No initial data file will be specified (i.e. *ALB_dataFile* will be blank).
- ◆ The application will determine its own generic parent (i.e. *ALB_genParent* will be null).
- ◆ No extra data word will be passed to the application (i.e. *ALB_extraData* will be zero).
- ◆ There will be no output descriptor (i.e. *ALB_userLoadAckOutput* and *ALB_userLoadAckMessage* will be null.)

IACPCreateDefaultLaunchBlock() returns the handle of the newly-created launch block. Once the block is created, you can alter any of its fields before passing the launch block to **IACPConnect()**. (Once you pass the launch block to **IACPConnect()**, you may not alter it any more.)

Often a client will want the server to open a specific document. For example, if a desktop-manager is implementing a "print-file" command, it will need to open a server application, instruct it to open the file to be printed, and then



instruct it to print the file. To make the server open a document, pass the document's name in *ALB_dataFile*. The server will open the file when you register, and close it when you unregister.

6.6.4.2 Unregistering as a Client

```
IACPShutdown(), IACPShutdownAll()
```

6.6

If an application no longer needs to interact with a particular server list, it should call **IACPShutdown()**. This routine is also used by servers which wish to remove themselves from a server list. This section describes how the routine is used by clients; section 6.6.5 of chapter 6 describes how it is used by servers.

The routine is passed two arguments:

- ◆ The **IACPConnection** token for the link which is being shut down.
- ◆ An *optr*. This must be null if the routine is being called by a client.

IACPShutdown() sends `MSG_META_IACP_LOST_CONNECTION` to all objects on the other side of the link; that is, if a client calls **IACPShutdown()**, all servers on the list will be sent this message.

IACPShutdownAll() closes all IACP links for the application which calls it. The Application object automatically calls this routine when the application is exiting.

6.6.5 Being a Server

Every time an application is launched, its Application object automatically registers as a server for the server list that shares its **GeodeToken**. The Application class has default handlers for all the notification messages IACP sends to the server objects.

If you wish, you can have another object act as a server. However, if you do this, you will have to do more of the work yourself. While the notification messages are defined for **MetaClass**, and thus can be handled by any class of object, **MetaClass** does not come with handlers for these messages; if the server is not subclassed from **GenApplicationClass**, you will have to write the handlers yourself. This is discussed below.

6.6.5.1 Registering and Unregistering a Server

```
IACPRegisterServer(), IACPUnregisterServer()
```

You will not generally need to register and unregister a server object explicitly. As noted above, when an application is launched, the application object is automatically registered as a server for the list with its **GeodeToken**; when the application exits, the Application object is automatically unregistered from that list.

6.6

However, you may wish to explicitly register an object as a server. For example, you might want your application object to be a server on a list with a different **GeodeToken**; or you might want to register a non-Application object as a server. In this case, you will need to explicitly register and unregister the object.

To register an object as a server, call **IACPRegisterServer()**. This routine is passed the following arguments:

- ◆ The **GeodeToken** of the list for which you are registering as a server.
- ◆ The **optr** of the object which is registering as a server. This object must be able to handle the MSG_META_IACP... messages appropriately (this is built into **GenApplicationClass**).
- ◆ A member of the **IACPServerMode** enumerated type. This type specifies how the client expects the server to behave. Currently, only two types are supported:

IACPSM_NOT_USER_INTERACTIBLE

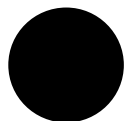
This is equal to zero. It indicates that the server object will not interact directly with users.

IACPSM_USER_INTERACTIBLE

This is equal to two. It indicates that the server will be able to interact with the user like any normal application.

- ◆ A set of **IACPServerFlags**. Currently, only one flag is supported: **IACP_MULTIPLE_INSTANCES**, indicating that multiple copies of the application might be running at once. (Every multi-launchable application should set this flag.)

IACPRegisterServer() registers the object as a server for the appropriate list; it creates the server list if necessary.



To unregister an object as a server, call **IACPUnregisterServer()**. This routine is passed two arguments: the **GeodeToken** of the server list, and the **optr** of the server. The object will be removed immediately from the server list. Note, however, that the server list might have already dispatched some messages to the server being removed; these messages might be waiting on the server object's queue, and thus the server object might get some IACP messages even after it calls **IACPUnregisterServer()**. One way to deal with this is to have the server object send itself a message, via the queue, immediately after it calls **IACPUnregisterServer()**. When the object receives this message, it will know that it has no more IACP messages on its queue.

Every server object on a given server list has a unique *server number*. This server number will not change while the server is attached to the list. A server object can find out its server number by calling **IACPGetServerNumber()**. This routine takes two arguments: the **GeodeToken** of the server list, and the **optr** to the server object. It returns the object's server number.

6.6.5.2 Non-Application Servers and Clients

```
MSG_META_IACP_PROCESS_MESSAGE, IACPProcessMessage( ),  
MSG_META_IACP_NEW_CONNECTION,  
MSG_META_IACP_LOST_CONNECTION
```

Every server and client object must be able to handle certain messages. **GenApplicationClass** comes with handlers for these messages, so you need not write them yourself. However, if you will be using some other kind of object as the server, you must handle the messages yourself. You may also choose to have your application object subclass any of these messages; in that case, you should generally have your handler use **@callsuper**.

When a server or client sends an IACP message, the kernel passes the encapsulated message to each object on the other side of the link. It does this by sending the message **MSG_META_IACP_PROCESS_MESSAGE** to each object. This message comes with three arguments:

msgToSend The **EventHandle** of the encapsulated message.

topt The **TravelOption** for that message.

completionMessage

The **EventHandle** of any message to be sent after *msgToSend* has been dispatched. (This field may be set to zero, indicating that there is no completion message.)

The recipient of MSG_META_IACP_PROCESS_MESSAGE should call **IACPProcessMessage()**. This routine is passed four arguments: the optr of the object calling the routine, and the three arguments passed with MSG_META_IACP_PROCESS_MESSAGE. **IACPProcessMessage()** dispatches both encapsulated messages properly.

6.6

Remember, if the client or server is subclassed from **GenApplicationClass**, all of this is done for you. You need only write a handler for the message if the client or server object is not from a subclass of **GenApplicationClass**.

Whenever a client registers on an IACP list, the kernel sends MSG_META_IACP_NEW_CONNECTION to all servers on that list. This message comes with three arguments:

appLaunchBlock

This is the handle of the **AppLaunchBlock** which the server passed to **IACPConnect()**.

justLaunched

This is a Boolean value. If the server's application has just been launched specifically to be a server, this will be *true* (i.e. non-zero).

IACPConnection

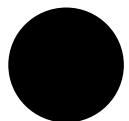
This is the token for the IACP connection.

When an object (either client or server) removes itself from an IACP connection, the kernel sends MSG_META_IACP_LOST_CONNECTION to all objects on the other side of the link. This message has two parameters:

connection The token for the IACP list.

serverNum If a server removed itself from the list (and this message is being sent to clients), this field will have the number of the server that removed itself. If a client removed itself from the list, this field will be zero.

Whenever a new client is attached to a server list, MSG_META_IACP_NEW_CONNECTION is sent to every server object. This message comes with three arguments:



appLaunchBlock

This is the handle of the application's launch block. The default **GenApplicationClass** handler examines the launch block to see if the application should open a document.

justLaunched

If the application was just launched specifically to be a server, this field will be *true* (i.e. non-zero).

6.6

connection This is the token for the IACP connection.

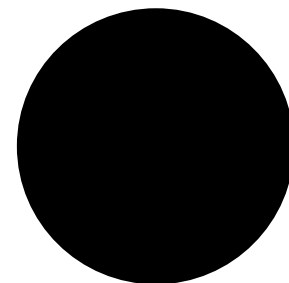
When a client is removed from a server list, every server object is sent MSG_META_IACP_LOST_CONNECTION. Similarly, when a server is removed, every client object is sent MSG_META_IACP_LOST_CONNECTION. The message comes with two arguments:

IACPConnection

This is the token for the IACP connection.

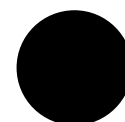
serverNum If a server left the list (and the message is being sent to clients), this argument will hold its *serverNum*. If a client left the list (and the message is being sent to servers), this argument will be set to zero.

The Clipboard



7

7.1	Overview	299
7.1.1	Cut, Copy, and Paste.....	300
7.1.2	Quick-Transfer	301
7.2	Transfer Data Structures.....	302
7.2.1	The Transfer VM File Format	303
7.2.2	ClipboardItemFormatInfo.....	305
7.2.3	Transfer Data Structures.....	307
7.2.4	Clipboard Item Formats	308
7.3	Using The Clipboard.....	309
7.3.1	Registering with the Clipboard.....	311
7.3.2	Managing the Edit Menu	311
7.3.3	The GenEditControl	314
7.3.4	Handling Cut and Copy	316
7.3.5	Handling Paste.....	319
7.3.6	Unregistering with the Clipboard.....	322
7.3.7	Implementing Undo	322
7.3.8	Transfer File Information	322
7.3.9	Undoing a Clipboard Change.....	323
7.4	Using Quick-Transfer	323
7.4.1	Supporting Quick-Transfer	324
7.4.2	Quick-Transfer Procedure.....	325
7.4.3	Quick-Transfer Data Structures.....	326
7.4.4	Source Object Responsibility.....	327
7.4.4.1	Responsibilities of a Potential Destination	328
7.4.4.2	Responsibilities of the Destination Object.....	330
7.4.4.3	Getting More Information	330
7.4.4.4	When the Transfer Is Concluded	330
7.5	Shutdown Issues	331





For an application to survive in the GUI world today, it must support at least the ever-present “Cut, Copy, and Paste” functions to which users have grown so attached. GEOS makes this very simple through the Clipboard and Quick-Transfer mechanisms.

All applications should support and use these mechanisms in order to maintain consistency throughout the system. In addition, these mechanisms provide certain functionality for data transfer that can be easily used.

7.1

To understand this chapter fully, you should have a working knowledge of Virtual Memory management as well as some familiarity with the concepts of mouse input. However, the concepts of the Clipboard and quick-transfer mechanisms are simple and may be understood without having read those sections.



7.1 Overview

The user understands two ways to cut and paste information: The first is to use the Edit menu and the *Cut*, *Copy*, and *Paste* functions. The second is to use the mouse by selecting and dragging text, graphics, or objects with the quick-copy and quick-move functions. For a full description of how these two operations are accomplished, see the user’s guide provided in the Geoworks retail products.

Both these functions use similar data structures; a single flag determines whether the operation is a quick-transfer or a normal Clipboard operation. However, the procedures and rules of usage have significant differences.

You should be familiar with the features and uses of both the Clipboard and the quick-transfer mechanism. If you are, you can skip this section; however, if you are not familiar with the features and rules governing the Edit menu and the quick-transfer operations, you most likely will want to read this section.

7.1.1 Cut, Copy, and Paste

Even the simplest, text-using applications provide an Edit menu with the *Cut*, *Copy*, and *Paste* options. A sample Edit menu is shown in Figure 7-1.

The Edit menu works with a hidden data area called the *Clipboard*. The Clipboard has no visual representation, and it can contain any format of data from text to graphics strings to custom formats defined by applications. The Clipboard's structure and implementation are completely invisible to the user beyond the functions *Cut*, *Copy*, and *Paste*.

The most common use of the Edit menu is within a word processor or draw program. For example, when a user wants to place a graphic from GeoDraw into GeoWrite, or when he wants to rearrange text in his NotePad, he can use the cut, copy, and paste operations.

When a user selects an item such as the word “puddle,” the *Cut* option becomes enabled. This indicates that the user can remove the selection from the document and it will be placed on the Clipboard (anything already on the Clipboard will be replaced with the selection). When nothing is selected, the *Cut* option is disabled.

In order to enable the *Copy* option, the user must select an item, just as with *Cut*. Copying, however, will not remove the selection from the document—it will simply create a copy of the selection and place it on the Clipboard, removing whatever had previously been on the Clipboard.



Figure 7-1 The Edit Menu

The Edit menu of NotePad contains the *Cut*, *Copy*, and *Paste* items.

Pasting is allowed only when something in a compatible data format is already sitting on the Clipboard. The *Paste* operation copies what is on the Clipboard into a document at the current insertion point (in the case of text, this is where the cursor was last placed); if something is already selected, the selection will be replaced with the pasted material. If there is nothing on the Clipboard, then there is nothing to paste; therefore, the option is disabled.

7.1.2 Quick-Transfer

7.1

The user can copy and move selected items without using the Edit menu; he simply drags the item using the quick-transfer button of his mouse, and the item will be either copied or moved.

There are several rules which govern whether a move or copy is taking place. The user can override these rules with a certain set of keypresses to force either a copy or move. Copy is equivalent to using the *Copy* and *Paste* functions in succession, and move is equivalent to using the *Cut* and *Paste* functions in succession.

The operation of a quick transfer (move or copy), however, depends on whether the transfer is across documents or internal to a single document. If the transfer is internal, it should become a move. If the transfer occurs across documents, it should become a copy. (Note that, for the purposes of quick-transfer, the term *document* is used to refer to anything the user will conceptually view as a single data holder—for example, a GeoManager “document” consists of one disk.)

Because not all documents will support a given transfer operation (for example, a text file editor can not receive a GeoDraw graphic), the user is provided with immediate feedback about the type and validity of the transfer.

For example, if a GeoDraw ellipse was selected and a quick-transfer was begun, immediately the mouse pointer would show the transfer as a move operation. As the user moves the mouse outside the GeoDraw document and into another GeoDraw document, the cursor changes to indicate a copy operation. If the user then moves the mouse over a text-only document, a “no operation,” or “invalid,” cursor would be shown, indicating that the ellipse could not be received.

Some applications may implement an additional feature that provides feedback to the user: modifying the source object as the type of transfer changes. For example, a drawing program may wish to alter the shape or color of an object while it is being quick-moved but leave it normal during a quick-copy or no-operation transfer.

7.2

7.2 Transfer Data Structures

Both the Clipboard and quick-transfer mechanisms use similar structures to accomplish data transfer. These data structures are owned and managed by the GEOS User Interface; however, they are accessible to all geodes in the system through a number of routines and messages.

The Clipboard consists of a VM file known as the *Transfer VM File*, shown in Figure 7-2. The quick-transfer mechanism uses the same file for its data transfer. Typically, this is a file designated and managed by the User Interface.

In addition, certain formats of data are supported automatically—GEOS text and graphics strings are special formats that are described below. Geodes can

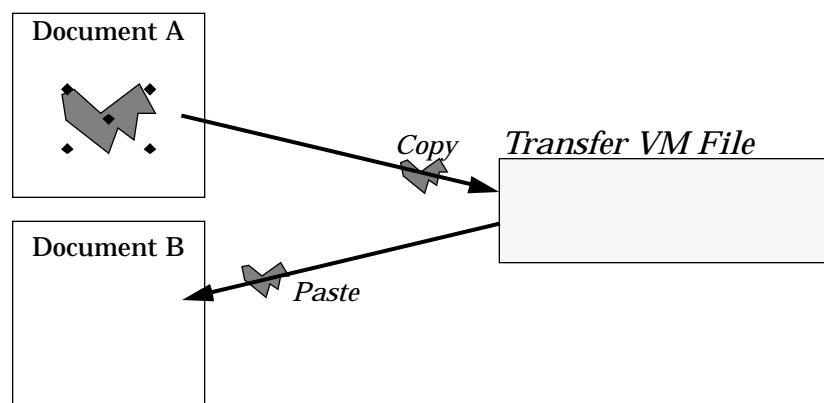


Figure 7-2 *The Transfer VM File*

How the Transfer VM File is used by the Copy and Paste functions; its use for quick-transfer is similar.

create their own specific formats; for example, GeoManager uses a special format for quick-transfer of files between disks and directories.

7.2.1 The Transfer VM File Format

The Transfer VM File is a normal VM file, managed by the UI. It contains several components, each of which is accessed through special routines that take care of nearly all the synchronization issues involved with the Clipboard and quick-transfer mechanisms. Because both the Clipboard and the quick-transfer mechanism use this file and its data structures, these structures are detailed here; the following section (“Using The Clipboard” on page 309) details how to use them for either the Clipboard or the quick-transfer mechanism.

7.2

The Transfer VM File has one header block for the Clipboard transfer item and another for the quick-transfer transfer item. These headers have the same structure, and they contain all the information necessary to work with the file's contents. They contain pointers to and information about each format available for both the Clipboard and the quick-transfer functions. The transfer item's header is shown in Code Display 7-1.

Code Display 7-1 ClipboardItemHeader

```
/* The ClipboardItemHeader structure details what formats, if any, exist in the
 * Transfer VM File for both the Clipboard and quick-transfer mechanisms.*/

typedef struct {

    /* The CIH_owner field is the optr of the object that put
     * the current transfer item into the Transfer VM File. */
    optr          CIH_owner;

    /* The CIH_flags field determines whether the transfer item is used by
     * the quick transfer mechanism or by the normal Clipboard. If this
     * field is equal to the constant CIF_QUICK, then the transfer item
     * belongs to the quick transfer mechanism. If it is any other value,
     * the transfer item belongs to the normal Clipboard. */
    ClipboardItemFlags  CIH_flags;
    /* The ClipboardItemFlags type has one predefined value:
     *      CIF_QUICK          0x4000
     * If this flag does not apply, use TIF_NORMAL, which is 0x0000. */
}
```



The Clipboard

304

7.2

```
/* The CIH_name field is a 33-character buffer that should contain a
 * null-terminated character string representing the name of the given
 * transfer item. */
ClipboardItemNameBuffer CIH_name;

/* The CIH_formatCount field indicates the number of formats of the
 * transfer item currently available. For example, this field would be
 * 2 if CIF_TEXT and CIF_GRAPHICS_STRING formats were supported and
 * available in the Transfer VM File. */
word CIH_formatCount;

/* The CIH_sourceID field contains information about the source
 * document of the transfer. Typically, this will be the optr of the
 * parent GenDocument object. */
dword CIH_sourceID;

/* The CIH_formats field is actually an array of ten
 * ClipboardFormatInfo structures, each of which contains the
 * important details about a specific format of the transfer item. */
FormatArray CIH_formats;

/* This field is reserved and should not be used. */
dword CIH_reserved;
} ClipboardItemHeader;

/* The FormatArray type is an array of ClipboardItemFormatInfo structures. The
 * definition of this field is shown here: */

typedef ClipboardItemFormatInfo FormatArray[CLIPBOARD_MAX_FORMATS];
/* CLIPBOARD_MAX_FORMATS is a constant defining the maximum number of
 * formats allowed for any given transfer item. It is defined as 10. */
```

The rest of the Transfer VM File consists of VM blocks containing the data that is to be transferred. Each format supported will have its own VM block or series of VM blocks, pointed to by the **ClipboardItemFormatInfo** structure in the item's header.

The Transfer VM File actually contains two transfer items: One for the Clipboard and one for the quick-transfer mechanism (see Figure 7-3). When calling **ClipboardQueryItem()**, the requesting geode must specify which item it wants. See section 7.3 on page 309 and section 7.4 on page 323.

7.2.2 ClipboardItemFormatInfo

This structure contains information about a specific format of the transfer item. The Transfer VM File will support up to ten formats of a given item at once (for both the Clipboard and quick-transfer); each of these formats is stored in its own VM block or VM chain and is represented in the header by a **ClipboardItemFormatInfo** structure in the array *CIH_formats*.

The **ClipboardItemFormatInfo** structure contains other information about the specific format as well as space for two extra words of data. The structure is shown in Code Display 7-2.

7.2

Each element in the *CIH_formats* array contains two items: One word represents the manufacturer ID of the geode responsible for the format (useful if a custom format is used within several applications from a single manufacturer), and the other represents the actual format number. To combine these words into a **ClipboardItemFormatID** record, or to extract either word from the record, use the macros shown after Code Display 7-2.

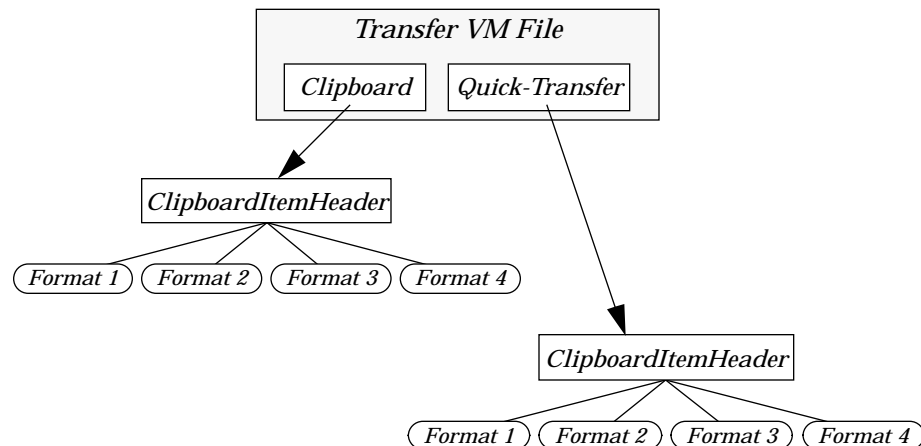


Figure 7-3 Transfer VM File Structure

The Transfer VM File contains two transfer items, one for the clipboard and one for the quick-transfer mechanism. Both can store multiple formats of their transfer items; each format may consist of a VM chain.



Code Display 7-2 ClipboardItemFormatInfo

```
/* The ClipboardItemFormatInfo structure contains information about a given
 * format of the transfer item. */

typedef struct {
    /* CIFI_format contains a format ID as well as the
     * manufacturer ID of the geode responsible for the format. */
    ClipboardItemFormatID    CIFI_format;

    /* CIFI_extra1 and CIFI_extra2 are extra words provided for
     * format-specific use. */
    word                    CIFI_extra1;
    word                    CIFI_extra2;

    /* CIFI_vmChain is a VM handle pointing to the first block in the
     * specific format. */
    VMChain                  CIFI_vmChain;

    /* CIFI_renderer contains the token of the geode that created
     * the format. */
    GeodeToken               CIFI_renderer;
} ClipboardItemFormatInfo;
```

7.2

Below are the macros for use with a **ClipboardItemFormatID** structure.

■ FormatIDFromManufacturerAndType

```
ClipboardItemFormatID FormatIDFromManufacturerAndType(man, typ)
    ManufacturerID      man;
    word                typ;
```

This macro creates a **ClipboardItemFormatID** dword value from the given manufacturer ID and format ID.

■ ManufacturerFromFormatID

```
ManufacturerID ManufacturerFromFormatID(type);
ClipboardItemFormatID type;
```

This macro extracts the manufacturer ID from the given clipboard format ID and manufacturer value.

■ TypeFromFormatID

```
word      TypeFromFormatID(type);
          ClipboardItemFormatID type;
```

This macro extracts the format ID from the given clipboard format ID and manufacturer value.

7.2.3 Transfer Data Structures

7.2

Two structures are used with specific routines when dealing with the transfer mechanisms. The **ClipboardQueryArgs** structure is returned by **ClipboardQueryItem()**, and the **ClipboardRequestArgs** structure is returned by **ClipboardRequestItemFormat()**. Both routines are used during a Paste operation, and both structures are shown in Code Display 7-3.

Note that the *CQA_header* field is of type **TransferBlockID**. This type is a dword made up of two word-sized components: a VM file handle and a VM block handle. The three macros listed after Code Display 7-3 can be used to create the **TransferBlockID** argument and extract either of the components from the whole.

Code Display 7-3 ClipboardQueryArgs and ClipboardRequestArgs

```
/* ClipboardQueryArgs is filled by ClipboardQueryItem(), which is called when
 * determining whether a transfer item exists. */
typedef struct {
    word      CQA_numFormats; /* the total number of formats available */
    optr      CQA_owner;      /* the optr of the originating object */
    TransferBlockID CQA_header; /* The combined VM file handle and VM block
                                * handle of the block containing the
                                * ClipboardItemHeader */
} ClipboardQueryArgs;

/* ClipboardRequestArgs is filled by ClipboardRequestItemFormat(), which is called
 * when the application wants to retrieve the current transfer item. */
typedef struct {
    VMFileHandle CRA_file; /* The VM file handle of the transfer file */
    VMChain      CRA_data; /* The handle of the VM chain containing the
```



The Clipboard

308

```

                                * transfer item */
word        CRA_extra1;      /* an extra word of data */
word        CRA_extra2;      /* another extra word of data */
} ClipboardRequestArgs;
```

Below are the macros for use with the **TransferBlockID** structure.

7.2

■ BlockIDFromFileAndBlock

```
TransferBlockID BlockIDFromFileAndBlock(f, b);
VMFileHandle      f;
VMBlockHandle     b;
```

This macro creates a **TransferBlockID** value from the given file and block handles.

■ FileFromTransferBlockID

```
VMFileHandle FileFromTransferBlockID(id);
TransferBlockID id;
```

This macro extracts the file handle from the given **TransferBlockID** value.

■ BlockFromTransferBlockID

```
VMBlockHandle BlockFromTransferBlockID(id);
TransferBlockID id;
```

This macro extracts the block handle from the given **TransferBlockID** value.

7.2.4 Clipboard Item Formats

There are several built-in transfer formats that many GEOS applications may support; each of these types is an enumeration of **ClipboardItemFormat**. Additionally, custom formats can be defined to allow special data structures to be cut, copied, pasted, or quick-transferred without translation into text or graphics strings. The Transfer VM File may contain up to ten formats of a given transfer item. **ClipboardItemFormat** is shown below.

```
typedef enum /* word */ {
    CIF_TEXT,
    CIF_GRAPHICS_STRING,
    CIF_FILES,
    CIF_SPREADSHEET,
    CIF_INK,
    CIF_GROBJ,
    CIF_GEODEX,
    CIF_BITMAP,
    CIF_SOUND_SYNT,
    CIF_SOUND_SAMPLE
} ClipboardItemFormat;
```

7.3

A transfer item of CIF_TEXT format is headed by a **TextTransferBlockHeader** structure. The text follows this header in the VM chain. A transfer item of CIF_GRAPHICS_STRING format is simply the entire GString stored in the transfer VM file in the VM chain.

Because every format identifier has two components, it is highly unlikely that two different designers will create overlapping custom formats.

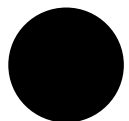
The format is defined as a **ClipboardItemFormatID** type, which is a dword composed of two word-sized pieces. The first piece is a constant representing the format ID number (such as CIF_TEXT or CIF_GRAPHICS_STRING). The second piece is a constant representing the Manufacturer ID number of the manufacturer responsible for creating the format.

To create a custom format, simply define these two items as appropriate (your Manufacturer ID should be set already). Then define your format to fit within the structures used by the Clipboard (shown above).

7.3 Using The Clipboard

```
ClipboardQueryItem(), ClipboardRegisterItem(),
ClipboardDoneWithItem()
```

To use the Clipboard, your application must have an Edit menu (or a GenEditControl object) and an object which can implement the Cut, Copy,



and Paste operations. This object is often the application's Process object or some other coordinating object. This object must be able to do each of the things in the following list:

- ◆ **Register with the Clipboard**
In a multitasking, multithreaded system, another application could change the Clipboard's contents. By registering with the Clipboard on application startup, the object will receive notification whenever the Clipboard's contents are changed (including when your application perpetrates the change). Registration can be either with special routines or with the GCN notification type `GWNT_TRANSFER_NOTIFICATION`.
- ◆ **Maintain the Edit Menu**
In order to provide the user with the *Cut*, *Copy*, and *Paste* features, your application must have an Edit menu. Most programmers will simply include a `GenEditControl` object to create and maintain the Edit menu. Your clipboard-management object must update the triggers properly whenever the contents of the Clipboard are changed.
- ◆ **Handle `MSG_META_CLIPBOARD_COPY`, `MSG_META_CLIPBOARD_CUT`**
When the user issues a Cut or Copy order, the object must put the proper data into the Clipboard Transfer VM File.
- ◆ **Handle `MSG_META_CLIPBOARD_PASTE`**
When the user issues a Paste order, the object must query the Clipboard to ensure a proper format is available and then copy the information from the Clipboard.
- ◆ **Check out when shutting down**
Each object that registers with the Clipboard must unregister when shutting down. Otherwise, the UI might try to send notification to a defunct object or process, resulting in unpredictable behavior.

Because the Clipboard is constantly in use by many different threads, you must always gain exclusive access to the transfer VM file when you want to use it. After you're done with the transfer file, you should relinquish exclusive access so other threads can continue to use it.

For operations that involve changing the transfer item (cut and copy, for example), you must register your new transfer item with

`ClipboardRegisterItem()`, which also allows other threads to use the file.

For operations that involve looking at but not changing the transfer item, you should use **`ClipboardQueryItem()`**. Since you have no changes to register,

you must later use **ClipboardDoneWithItem()** to give up your exclusive access to the transfer VM file.

7.3.1 Registering with the Clipboard

```
ClipboardAddToNotificationList()
```

Because the Clipboard is a system entity available to all geodes, another thread may change it without your application noticing. The Clipboard therefore provides notification for this case. Because the Clipboard does not know which geodes are interested in its contents, however, applications must register when they first start up.

7.3

Calling **ClipboardAddToNotificationList()** allows an application to add an object to the list of those notified of changes to the Clipboard. This routine should be called by whichever object is going to be handling the Cut, Copy, and Paste operations, typically in the object's MSG_META_INITIALIZE handler. If the object handling the Clipboard operations is the application's Process object, however, it may call **ClipboardAddToNotificationList()** in its MSG_GEN_PROCESS_OPEN_APPLICATION handler.

7.3.2 Managing the Edit Menu

```
MSG_META_CLIPBOARD_NOTIFY_NORMAL_TRANSFER_ITEM_CHANGED,  
ClipboardTestItemFormat()
```

The Edit menu is simply a normal menu with several standard triggers. Most applications will simply include a GenEditControl object in their UI, add a menu GenInteraction of type GIGT_EDIT_MENU, and leave the Edit menu construction up to them (see section 7.3.3 on page 314). Some, however, may want to create their own menu and triggers. A sample of this type of setup is shown in Code Display 7-4.

Code Display 7-4 A Sample Edit Menu

```
/* Other objects, including a GenPrimary as the parent of the GenInteraction, are  
 * left out for clarity. */
```



The Clipboard

312

```
/* The GenInteraction is the menu in which the three triggers will appear. */
@object GenInteractionClass EditMenu = {
    GI_visMoniker = 'E', "Edit";
    GI_comp = EditCut, EditCopy, EditPaste;
    GII_visibility = GIV_POPUP;
}

/* The Cut trigger sends a MSG_META_CLIPBOARD_CUT to the Process
 * object when pressed. */
7.3 @object GenTriggerClass EditCut = {
    GI_visMoniker = 't', "Cut";
    GTI_destination = process;
    GTI_actionMsg = MSG_META_CLIPBOARD_CUT;
}

/* The Copy trigger sends a MSG_META_CLIPBOARD_COPY to the
 * Process object when pressed. */
@object GenTriggerClass EditCopy = {
    GI_visMoniker = 'C', "Copy";
    GTI_destination = process;
    GTI_actionMsg = MSG_META_CLIPBOARD_COPY;
}

/* the Paste trigger is set up initially disabled. It sends a
 * MSG_META_CLIPBOARD_PASTE to the Process object when pressed. */
@object GenTriggerClass EditPaste = {
    GI_visMoniker = 'P', "Paste";
    GTI_destination = process;
    GTI_actionMsg = MSG_META_CLIPBOARD_PASTE;
    GI_states = @default & ~GS_ENABLED;
}
```

Some Edit menus may also contain other triggers such as “Delete Event,” or “Remove Item.” These triggers, however, are not standard and must be implemented exclusively by the application.

Two main rules govern the maintenance of the Edit menu:

- ◆ The *Cut* and *Copy* triggers are enabled only when some data is selected that may be cut or copied from the document into the Clipboard.
- ◆ The *Paste* trigger is enabled only when data in a pasteable format exists in the Clipboard’s Transfer VM File.

The first rule must be implemented entirely by the application; the Clipboard will not enable or disable the Copy or Cut triggers. The second rule, however, requires that the application be notified whenever the Clipboard's contents change—it is possible, for example, for the application to copy data to the Clipboard, enable its Paste trigger, and have another application then also copy some custom data to the Clipboard. In this case, the original application must disable its Paste trigger if it can not read the data.

Whenever the Clipboard's contents change, the UI will send notification to all objects that have registered with it. The notification will be in the message `MSG_META_CLIPBOARD_NOTIFY_NORMAL_TRANSFER_ITEM_CHANGED`. Use of this message can be found in the `ClipSamp` sample application.

7.3

Code Display 7-5 Handling Clipboard Changes

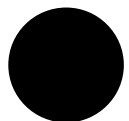
```
/* MSG_META_CLIPBOARD_NOTIFY_NORMAL_TRANSFER_ITEM_CHANGED is sent with no
 * parameters and requires no return value. */

/* The strategy of this message is to first check whether the CIF_TEXT format, the
 * only format supported by this sample application, is available on the Clipboard.
 * If so, the Paste trigger is enabled; if not, the Paste trigger is disabled. */

@method MyClipProcessClass, MSG_META_CLIPBOARD_NOTIFY_NORMAL_TRANSFER_ITEM_CHANGED
{
    ClipboardQueryArgs  query;                /* A structure of arguments */
    Boolean              endisable = FALSE;    /* The trigger is disabled */

    /* Call ClipboardQueryItem() to gain exclusive access to and information about
     * the current transfer item. Pass it zero indicating we're checking the
     * normal transfer item (not the quick-transfer item) and the empty arguments
     * structure. */
    ClipboardQueryItem(TIF_NORMAL, &query);

    /* If there are any formats, then test if CIF_TEXT is one of them. The routine
     * ClipboardTestItemFormat() tests a format against all those available and
     * returns true if it is supported, false if it is not. Use the macro
     * FormatIDFromManufacturerAndType to create the format argument. If CIF_TEXT
     * is not supported, then the endisable argument is set to TRUE. */
    if (query.CQA_numFormats) {
        if (ClipboardTestItemFormat(query.CQA_header,
                                     FormatIDFromManufacturerAndType(MANUFACTURER_ID_ME, CIF_TEXT))) {
            endisable = TRUE;
        }
    }
}
```



The Clipboard

314

```
/* Because we've found out what we need to know, restore the Clipboard with a
 * call to ClipboardDoneWithItem(). This routine takes the transfer item's
 * header and returns nothing; it also relinquishes our exclusive
 * access to the Clipboard and is therefore very important. */
```

```
ClipboardDoneWithItem(query.CQA_header);
```

```
/* Now, if endisable is true, set the Paste trigger enabled. If endisable is
 * false, set it disabled. These operations are accomplished by sending the
 * appropriate message to the trigger object. */
```

7.3

```
if (endisable) {
    @call EditPaste::MSG_GEN_SET_ENABLED(VUM_NOW);
} else {
    @call EditPaste::MSG_GEN_SET_NOT_ENABLED(VUM_NOW);
}
}
```

7.3.3 The GenEditControl

As stated above, most applications will simply let a **GenEditControl** object create and maintain their Edit menu. **GenEditControlClass** is a subclass of **GenControlClass** (see “Generic UI Controllers,” Chapter 12 of the Object Reference Book for usage of controllers in general).

The **GenEditControl** object can provide triggers and/or tools for Undo, Cut, Copy, Paste, Select All, and Delete operations. These operations must all be handled by your application, of course, just as if you did not use a **GenEditControl**; using this controller, however, simplifies your UI programming and allows the Edit tools to be used by the **GenToolControl**.

The features of the **GenEditControl** are listed below (they are flags of the **GECFeatures** record type):

GECF_UNDO This feature adds an “Undo” trigger to the Edit menu. It sends **MSG_META_UNDO** to the application's target.

GECF_CUT This feature adds a “Cut” trigger to the Edit menu. When the user activates this, it sends **MSG_META_CLIPBOARD_CUT** to the application's target.

GECF_COPY This feature adds a “Copy” trigger to the Edit menu. When the user activates this, it sends MSG_META_CLIPBOARD_COPY to the application’s target.

GECF_PASTE This feature adds a “Paste” trigger to the Edit menu. When the user activates this, it sends MSG_META_CLIPBOARD_PASTE to the application’s target.

GECF_SELECT_ALL

This feature adds a “Select All” trigger to the Edit menu. It sends MSG_META_SELECT_ALL to the applications’ target.

7.3

GECF_DELETE

This feature adds a “Delete” trigger to the Edit menu. When the user activates this, it sends MSG_META_DELETE to the application’s target.

The GenEditControl also provides an equivalent set of tools. Each tool executes the exact same functions as the analogous feature; see Code Display 7-6 for the listing of the features and tools as well as the standard settings.

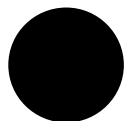
The GenEditControl handles two different notification types:

GWNT_SELECT_STATE_CHANGE, sent when the selection state changes, and GWNT_UNDO_STATE_CHANGE, sent when a state change in the Undo status occurs. In both cases, the GenEditControl will appropriately update the Cut, Copy, Paste, Delete, and Undo triggers (the Select All trigger will always be enabled).

Code Display 7-6 GenEditControl Features and Tools

```
/* This display shows the features and tools records of GenEditControlClass, as
 * well as the default settings and instance data. */
```

```
/* GenEditControlClass features */
typedef WordFlags GECFeatures;
#define GECF_UNDO          0x0020          /* MSG_META_UNDO */
#define GECF_CUT           0x0010          /* MSG_META_CLIPBOARD_CUT */
#define GECF_COPY          0x0008          /* MSG_META_CLIPBOARD_COPY */
#define GECF_PASTE         0x0004          /* MSG_META_CLIPBOARD_PASTE */
#define GECF_SELECT_ALL    0x0002          /* MSG_META_SELECT_ALL */
#define GECF_DELETE        0x0001          /* MSG_META_DELETE */
```



```
#define GEC_DEFAULT_FEATURES      (GECF_UNDO | GECF_CUT | GECF_COPY | \
                                  GECF_PASTE | GECF_SELECT_ALL | GECF_DELETE)

/* GenEditControlClass tools */
typedef WordFlags GECToolboxFeatures;
#define GECTF_UNDO                0x0020          /* MSG_META_UNDO */
#define GECTF_CUT                 0x0010          /* MSG_META_CLIPBOARD_CUT */
#define GECTF_COPY                0x0008          /* MSG_META_CLIPBOARD_COPY */
#define GECTF_PASTE               0x0004          /* MSG_META_CLIPBOARD_PASTE */
7.3 #define GECTF_SELECT_ALL       0x0002          /* MSG_META_SELECT_ALL */
#define GECTF_DELETE              0x0001          /* MSG_META_DELETE */

#define GEC_DEFAULT_TOOLBOX_FEATURES (GECTF_UNDO | GECTF_CUT | GECTF_COPY |
                                     GECTF_PASTE | GECTF_SELECT_ALL |
                                     GECTF_DELETE)

/* GenEditControlClass Instance Data Settings */
@default GCI_output = (TO_APP_TARGET);          /* Send output to the target */
@default GI_states = (@default | GS_ENABLED);
@default GI_attrs = (@default | GA_KBD_SEARCH_PATH);
```

7.3.4 Handling Cut and Copy

MSG_META_CLIPBOARD_CUT, MSG_META_CLIPBOARD_COPY

Cut and Copy are very similar in function; both put data onto the Clipboard. However, Cut causes the data to subsequently be deleted from the document, and Copy does not.

When the user starts either of these operations, the object that handles them must go through a series of specific steps to load the data into the Clipboard's VM file. (For simplicity of example, this chapter will assume that the Process object will handle all Clipboard operations; this may not be the case in complex programs.)

The steps are simple; each is enumerated below, and edited examples from the sample application ClipSamp are provided in Code Display 7-7 and Code Display 7-8. Note that these examples do not use the default text object handlers for copy and paste; they treat the entire text flow as the current selection.

1 Duplicate and attach the data

You must create a duplicate of whatever data is being loaded into the Clipboard. This step includes allocating new VM blocks in the Transfer VM File with **VMAlloc()**. As an alternative, you may pre-duplicate the item in memory with **MemAlloc()** and then simply attach them to the Transfer VM File with **VMAttach()**.

2 Complete the transfer item's header

Fill in all information fields in the transfer item's header block including formats, owner, and flags.

7.3

3 Register the transfer item

Once the data has been attached and the header completed, you must register the transfer with the Clipboard. The UI will then delete any old data in the Clipboard and replace it with your new transfer item. To register the transfer item, use **ClipboardRegisterItem()**.

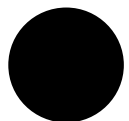
Code Display 7-7 MSG_META_CLIPBOARD_CUT

```
/* This is the same as MSG_META_CLIPBOARD_COPY except that after copying the
 * data to the Clipboard, it deletes the data from the document.
 *
 * MSG_META_CLIPBOARD_CUT has no parameters and no return value. The strategy
 * is as follows: First, copy the subject data into the Clipboard with
 * MSG_META_CLIPBOARD_COPY. Then, delete the data (which, in this case, is a single
 * memory block containing all the subject text). */

@interface ClipSampProcessClass, MSG_META_CLIPBOARD_CUT {
    /* Use MSG_META_CLIPBOARD_COPY to copy the data to the clipboard. */
    @call self::MSG_META_CLIPBOARD_COPY();

    /* Delete the data. The data is contained entirely within a single memory
     * block and is just text. The block is referenced by the memory handle
     * textHandle. If textHandle is not null, then the block may be freed. */
    if (textHandle) {
        MemFree(textHandle);          /* If textHandle is valid, */
        textHandle = 0;                /* free the memory block */
    }                                /* and zero the handle. */

    /* Redraw the view area to reflect the deleted text. */
    ResetViewArea();                  /* Custom routine to redraw the view. */
}
```



Code Display 7-8 MSG_META_CLIPBOARD_COPY

```
/* This message handler goes through all the steps necessary for a Copy operation
 * that works with text data only.
 * MSG_META_CLIPBOARD_COPY has no parameters and requires no return.
 *
 * The strategy employed by this handler is as follows:
 * First, allocate memory for and create the duplicate data block, filling in all
 * the appropriate fields.
7.3 * Next, retrieve the Transfer VM File and attach the data block to the file.
 * Next, allocate and construct the transfer item header VM block.
 * Finally, register and lock in the transfer item to the Clipboard.
 *
 * A single global variable named textHandle refers to the block of text owned and
 * used by the sample application. All other data structures are defined within the
 * message handler. */

@method ClipSampProcessClass, MSG_META_CLIPBOARD_COPY {
    char          *textText;      /* temporary string for the text */
    int           textLength;     /* length of string including null */
    MemHandle     headerMemHandle; /* handle of ClipboardItemHeader block */
    VMFileHandle  transferVMFile; /* VM file handle of Transfer VM File */
    VMBlockHandle dataVMBlock;    /* VM handle of attached data block */
    VMBlockHandle headerVMBlock;  /* VM handle of attached header block */
    ClipboardItemHeader *headerMem; /* ClipboardItemHeader for the VM file */
    optr          textObj;        /* temporary text object for transfer */

    /* First, lock the text string into memory and get its length, adding one for
     * the null character at the end. Then unlock the text string's block. */

    textText = (char *) MemLock(textHandle);
    textLength = (strlen(textText) + 1);
    MemUnlock(textHandle);

    /* Next, build the transfer item block by creating a temporary text object and
     * copying our text into it. Other formats may simply copy the text directly
     * into a VM block. */

    textObj = TextAllocClipboardObject(ClipboardGetClipboardFile(), 0, 0);
    @call textObj::MSG_VIS_TEXT_REPLACE_ALL_PTR((char *)MemLock(textHandle), 0);
    MemUnlock(textHandle);
    dataVMBlock = TextFinishedWithClipboardObject(
                                textObj, TCO_RETURN_TRANSFER_FORMAT);

    /* Now get the transfer VM file. */
    transferVMFile = ClipboardGetClipboardFile();

    /* Now, allocate and fill in the transfer item header block. */
```

```

headerVMBlock = VMalloc(      transferVMFile,
                             sizeof(ClipboardItemHeader),
                             MY_TRANSFER_ID);
headerMem = (ClipboardItemHeader *)VLock(transferVMFile, headerVMBlock,
                                         &headerMemHandle);
headerMem->CIH_owner = (optr) (((dword)GeodeGetProcessHandle()<<16) | 0);
headerMem->CIH_flags = 0;      /* Normal transfer; no flags. */
(void) strcpy(headerMem->CIH_name, "Sample Text");
headerMem->CIH_formatCount = 1;
headerMem->CIH_sourceID = 0;
headerMem->CIH_formats[0].CFI_format =
    FormatIDFromManufacturerAndType(MANUFACTURER_ID_ME, CIF_TEXT);
headerMem->CIH_formats[0].CFI_vmChain =
    VMCHAIN_MAKE_FROM_VM_BLOCK(dataVMBlock);
headerMem->CIH_formats[0].CFI_extral = 0;
headerMem->CIH_formats[0].CFI_extra2 = 0;
VMUnlock(headerMemHandle);

/* Now register the transfer item with the Clipboard. This will actually put
 * the transfer item and its header into the Clipboard. */
ClipboardRegisterItem(BlockIDFromFileAndBlock(
    transferVMFile, headerVMBlock),
    0);
}

```

7.3

7.3.5 Handling Paste

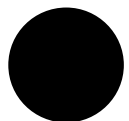
ClipboardRequestItemFormat(), MSG_META_CLIPBOARD_PASTE

The Paste operation pulls data off the Clipboard and places it at the insertion point in the application's data. The Clipboard remains unchanged throughout the operation; the data is simply duplicated and passed on to the application.

The steps in handling a MSG_META_CLIPBOARD_PASTE are simple; each is enumerated below, and a sample method for pasting is shown in Code Display 7-9.

1 Query the Clipboard

First, you must make sure that you have exclusive access to the clipboard item. To this end call **ClipboardQueryItem()**. You should also call



The Clipboard

320

ClipboardRequestItemFormat() to make sure that the present clipboard item is pasteable.

2 Allocate memory if necessary

If necessary, allocate the memory into which the transfer item will be duplicated. You can not simply reference handles to the transfer item in the clipboard because the transfer item may be changed by another thread at any time.

7.3

3 Lock the Transfer VM File and grab the transfer item

Lock the Transfer VM File with a call to

ClipboardRequestItemFormat(). Finally, copy the transfer item into your pre-allocated memory.

4 Unlock the Transfer VM File

By calling **ClipboardDoneWithItem()**, relinquish your exclusive access to the Transfer VM File and to the clipboard item itself. The Paste operation can then be completed entirely by your application by assimilating the pasted data and displaying it properly.

Code Display 7-9 MSG_META_CLIPBOARD_PASTE

```
/* This message handler goes through the necessary steps to grab the transfer item
 * from the Clipboard and copy it into application's memory. This example uses a
 * single global variable called textHandle, a memory handle of the only data block
 * owned by the application. The memory block contains text.
 * MSG_META_CLIPBOARD_PASTE has no parameters and requires no return value. */

@method ClipSampProcessClass, MSG_META_CLIPBOARD_PASTE {
    ClipboardQueryArgs      query;          /* returned by
                                             * ClipboardQueryItem() */

    ClipboardRequestArgs    request;        /* returned by
                                             * ClipboardRequestItemFormat() */

    TextTransferBlockHeader *dataBlock      /* pointer to block header */
    MemHandle               dataBlockMemHandle; /* handle of locked block */
    word                    charsAvail;      /* number of chars in block */
    int                     textLength;      /* length of text */
    word                    transferFlags = 0; /* flags for the transfer
                                             * (normal transfer) */

    /* Call ClipboardQueryItem() to be sure that a normal
     * transfer item exists in the Clipboard. */
    ClipboardQueryItem(transferFlags, &query);
    /* Fills ClipboardQueryArgs structure */
```



```

        /* If a transfer item exists, check for a CIF_TEXT
        * version, the only format we support. */
if (query.CQA_numFormats) { /* if more than zero formats available */
    if (ClipboardTestItemFormat(query.CQA_header,
                                FormatIDFromManufacturerAndType(
                                    MANUFACTURER_ID_ME,
                                    CIF_TEXT))) {

/* A CIF_TEXT version exists. Now we grab that transfer item by calling
* ClipboardRequestItemFormat(). This routine fills ClipboardRequestArgs,
* which contains information about the Transfer VM File and the
* ClipboardItemHeader block in that file. */

        ClipboardRequestItemFormat(FormatIDFromManufacturerAndType(
                                    MANUFACTURER_ID_ME, CIF_TEXT),
                                    query.CQA_header, &request);

/* Now we have the VM file handle of the Transfer VM File and the VM block
* handle of the ClipboardItemHeader structure. From this we can get the
* data in the data block. To speed things up, we will copy the transfer
* text directly into our already-allocated memory block; the handle to our
* memory block is in textHandle. */

        dataBlock = (TextTransferBlockHeader *)VMLock(
                                request.CRA_file,
                                VMCHAIN_GET_VM_BLOCK(request.CRA_data),
                                &dataBlockMemHandle);
        textHugeArray = VMCHAIN_GET_VM_BLOCK(dataBlock->TTBH_text);
        VMUnlock(dataBlockMemHandle);

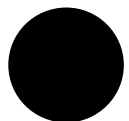
/* Since the data is CIF_TEXT, we know the data block is in the format of
* TextTransferBlockHeader. We get the text by cycling through the
* format's HugeArray. This code has been taken out of this example for
* simplicity; you can look at the ClipSamp sample application source
* code for it. */
    }

/* After copying the text into our block, we signal we're done with the
* transfer file. We do this by calling ClipboardDoneWithItem(). After
* that, we update our view and return. */

ClipboardDoneWithItem(query.CQA_header);
ResetViewArea(); /* Routine defined in ClipSamp. */
}

```

7.3



7.3.6 Unregistering with the Clipboard

```
ClipboardRemoveFromNotificationList()
```

Because the Clipboard sends notification out to all registered geodes, geodes must “unregister” when they are shutting down. Otherwise, the Clipboard will attempt to send a message to a defunct object, and this can cause problems for the operating system. Therefore, in your

7.3

`MSG_GEN_PROCESS_CLOSE_APPLICATION` handler you should make a call to the routine **ClipboardRemoveFromNotificationList()**, which removes the passed object from the notification list.

7.3.7 Implementing Undo

For the most part, implementation of Undo is left up to the application. This is due to the fact that operations that may be undone are typically very application-specific. The text objects and the Ink object are the only exceptions to this; they provide their own Undo functions in response to `MSG_META_UNDO`. For more information on Undo and how it works in GEOS, see “UI Messages” on page 88 of “System Classes,” Chapter 1 of the Object Reference Book.

7.3.8 Transfer File Information

```
ClipboardTestItemFormat(), ClipboardEnumItemFormats(),  
ClipboardGetItemInfo(), ClipboardGetNormalItemInfo(),  
ClipboardGetUndoItemInfo(), ClipboardGetClipboardFile()
```

With the following routines, you can get information about any of the transfer files in use.

ClipboardTestItemFormat()

Given a transfer format, test if the selected transfer item supports that format. Before using this routine, you must first call **ClipboardQueryItem()** to get the transfer item header.

ClipboardEnumItemFormats()

Return a list of supported transfer formats for the selected

transfer item. Before using this routine, you must first call **ClipboardQueryItem()** to get the transfer item header.

ClipboardGetItemInfo()

Return the source identifier for the transfer item. Before using this routine, you must first call **ClipboardQueryItem()** to get the transfer item header.

ClipboardGetNormalItemInfo()

Return the VM file handle and VM block handle of the transfer item header for the “Normal” transfer item.

7.4

ClipboardGetUndoItemInfo()

Return the VM file handle and VM block handle of the transfer item header for the “Undo” transfer item.

ClipboardGetClipboardFile()

Return the VM file handle of the UI transfer file (the one typically used when copying, cutting, and pasting).

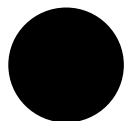
7.3.9 Undoing a Clipboard Change

```
ClipboardUnregisterItem()
```

Using **ClipboardUnregisterItem()**, you can revert one level of clipboard changes. Note that this works for only one level; there is no way to back out more than one change to the clipboard. This routine can not “undo” itself; that is, calling this routine twice in a row will leave the clipboard in a state other than the original state.

7.4 Using Quick-Transfer

An application must understand the Clipboard and its structure before being able to support the quick-transfer feature of the UI. However, because quick-transfer is such a convenient feature for users, every appropriate application should support it. To use the quick-transfer mechanism, you will probably want to understand how mouse input is handled.



The quick-transfer mechanism makes extensive use of the Clipboard's data structures. However, this does not mean that when a quick-transfer is initiated, the Clipboard is altered. Instead, the Clipboard maintains a separate (but similar) data structure within the Transfer VM File.

When a transfer is in progress, the distinction between a normal transfer and a quick transfer is made with the flag `CIF_QUICK`. When passed to the transfer mechanism's routines, this flag indicates that the quick-transfer item should be accessed and the Clipboard data should remain intact.

7.4

7.4.1 Supporting Quick-Transfer

In order for an application to support the quick-transfer mechanism, it must be able to handle several situations. The list below enumerates all the tasks the application must be ready to perform:

- ◆ Recognize initiation of a move/copy operation
When the user initiates a quick-transfer, the object under the mouse pointer will be notified the operation has begun.
- ◆ Provide feedback
Once a quick-transfer is initiated, every object on the screen becomes a potential destination for the transfer. When the mouse pointer moves over an object, that object must indicate what type of operation can be supported, if any.
- ◆ Receive the transfer item
If the move/copy operation is completed by the user when the mouse pointer is over an object, that object must be able to receive the item and provide feedback to the UI about the status of the quick-transfer.
- ◆ Recognize completion of the operation
When a quick-transfer is completed, the source object is notified and informed what type of operation took place.
- ◆ Shut off quick-transfer feedback on shutdown
When an object that can handle quick-transfers is shut down or destroyed, it must remove itself from the quick-transfer notification list. This takes care of any potential synchronization problems between object shutdown and the quick-transfer mechanism.

Applications should also understand the three rules that govern the behavior of a quick-transfer:

- ◆ Transfers within a single document default to move operations.
- ◆ Transfers across documents default to copy operations.
- ◆ Transfers in a format not supported by the destination are “no operation” transfers.

7.4

7.4.2 Quick-Transfer Procedure

Although applications must handle several situations to support the quick-transfer mechanism, the procedure involved in a quick-transfer is quite simple. The steps of how a quick-transfer operation is performed are outlined below:

- 1** The user initiates a quick-transfer
By pressing the right mouse button (in OSF/Motif), the user initiates a quick-transfer. The UI recognizes the mouse press and sends `MSG_META_START_MOVE_COPY` to the object under the pointer image.
- 2** The source object builds the transfer item
The object under the pointer image then becomes the “source” of the quick-transfer. It first calls **`ClipboardStartQuickTransfer()`** to initiate the quick-transfer mechanism. It then builds the transfer item just as it would if the user had clicked on the Copy trigger in the Edit menu. It then logs the transfer item with the quick-transfer mechanism.
- 3** The source becomes a potential destination
Immediately after logging the transfer item, the source object becomes a potential destination of the quick-transfer. It must immediately provide feedback to the UI indicating whether it can accept the transfer item and whether the operation would be a move or a copy. The feedback is provided by calling **`ClipboardSetQuickTransferFeedback()`**. If the source object is a visible object in a GenView, it must also send the message `MSG_VIS_VUP_ALLOW_GLOBAL_TRANSFER` to itself to allow the pointer events to be sent to other objects in other windows (because the GenView grabs the mouse on the press).
- 4** The user moves the mouse
When the user moves the mouse (continuing to hold down the move/copy button), the pointer image may cross over several objects. Each of these

is a potential destination and as such must provide feedback similar to that described in (3) above until the pointer moves outside of its bounds. Each object that receives a MSG_META_PTR should check if a quick transfer is in progress by either checking the passed event flags or by calling **ClipboardGetQuickTransferStatus()**. The object should, in response, provide feedback as to whether it can accept the transfer item or not. It calls **ClipboardSetQuickTransferFeedback()** with the proper feedback signal.

7.4

- 5 The user finishes the transfer
When the user lets up the move/copy button, the object under the pointer image (if any) becomes the destination object (it will receive a MSG_META_END_MOVE_COPY from the UI).
- 6 The destination receives the transfer item
If the transfer item is in a receivable format, the destination will retrieve the item from the Transfer VM File just as if the user had selected the Paste trigger from the Edit menu (except the quick-transfer transfer item is received, not the Clipboard transfer item). The object first checks if it can take the item by calling **ClipboardGetQuickItemInfo()** on the transfer item. If it can handle the item, it calls **ClipboardQueryItem()**, grabs the transfer item, and finally calls **ClipboardEndQuickTransfer()**.
- 7 The UI informs the source of the outcome
After the transfer has been completed by the destination, the UI will send a MSG_META_CLIPBOARD_NOTIFY_QUICK_TRANSFER_CONCLUDED to the source object, informing it about the final outcome of the operation. Some source objects will change shape, shading, or color during a quick-transfer and must know when the transfer is concluded. If the operation is a quick-move, the source must delete the information or object that was moved.

7.4.3 Quick-Transfer Data Structures

The quick-transfer mechanism uses the same structures as the Clipboard. However, there are special data structures that are used exclusively by the quick-transfer mechanism. These data structures are used by individual UI routines and are documented with those routines.

7.4.4 Source Object Responsibility

```
MSG_META_START_MOVE_COPY, ClipboardStartQuickTransfer(),  
MSG_META_CLIPBOARD_NOTIFY_QUICK_TRANSFER_FEEDBACK
```

When the user presses the move/copy button, the UI sends a `MSG_META_START_MOVE_COPY` to the object under the mouse pointer. The selected object can be either a gadget run in the UI thread (such as a GenText object) or a process-run visible object within a view.

7.4

If the object is a process-run visible object in a view, both the application object and the object under the mouse pointer will receive the notification message. If the object is UI-run, only it will receive the message.

Receipt of this message tells an object to begin a quick-transfer operation. This operation consists of several steps:

- 1 Grab the mouse to track it so you can find out when it leaves your object's bounds. (When it leaves, your object is no longer the destination.) Note that only objects need to grab the mouse; if a process is the content of the view, it does not have to.
- 2 Provide feedback to the quick-transfer mechanism immediately.
- 3 Build and register the transfer item.
- 4 Continue providing feedback as long as the mouse is in your bounds.
- 5 If desired, provide visual feedback to the user as the type of operation changes when the user moves the mouse pointer.

The reason for providing feedback to the quick-transfer mechanism is simple: to indicate to the user what is going on. By giving information to the quick-transfer mechanism, objects allow the user to be informed immediately what type of operation is in progress—a move, a copy, or nothing at all.

Immediately after your object has grabbed the mouse, it should call the routine **ClipboardStartQuickTransfer()**. This routine allows the object not only to indicate which type of operation is in progress but also to attach a special graphical region to the cursor (though not required). This allows the application to provide additional information to the user as to what is going on (e.g. GeoManager attaches an image when a file transfer is initiated).

You must also indicate to **ClipboardStartQuickTransfer()** the object that will receive notification when the transfer has concluded. This is important because when a quick-move has been completed, the source object must ensure that the original copy of the item (usually the source object itself) is deleted.

After calling **ClipboardStartQuickTransfer()**, the source object should duplicate and register the transfer item with the Transfer VM File. To do this, register the item as you normally would for a cut or copy operation (see section 7.3.1 on page 311); however, be sure to use the flag `CIF_QUICK` to ensure that the normal Clipboard data remains unaffected.

Once the transfer item has been registered, the source object becomes a potential destination and should act as such. However, you may wish to continue to provide source-related visual feedback to the user as long as the quick-transfer is going on: During a quick-transfer, the source will receive `MSG_META_CLIPBOARD_NOTIFY_QUICK_TRANSFER_FEEDBACK` each time the mode of the quick-transfer changes. This message will tell the source object what the current mode of transfer is in order for the source to give extra visual feedback to the user. This behavior is not required of the source object but can be beneficial to your application. It is also supplemental to the destination-related feedback that must be provided.

7.4.4.1 Responsibilities of a Potential Destination

```
ClipboardGetQuickTransferStatus(),  
ClipboardSetQuickTransferFeedback(),  
MSG_VIS_VUP_ALLOW_GLOBAL_TRANSFER,  
MSG_GEN_VIEW_ALLOW_GLOBAL_TRANSFER
```

All objects that can potentially receive a transfer item are considered potential destinations during a quick-transfer operation. During the operation, the user will likely move the mouse pointer across the screen, entering and leaving several different potential destinations.

When the mouse first moves over the object, the object will receive a `MSG_META_PTR`. When your object receives this message, it must provide immediate feedback to the transfer mechanism to indicate whether a move, a copy, or no operation is to be performed (the object should provide this

feedback with the understanding that the operation type is what would happen if the transfer were to conclude at that moment).

When the first MSG_META_PTR is received, the object should call the routine **ClipboardGetQuickTransferStatus()**. This routine returns whether a quick-transfer is in progress; if so, the object should acquire a mouse grab in order to provide feedback until the mouse pointer leaves its bounds.

The object should then check the quick-transfer Clipboard for supported formats. This is done just as with the Clipboard—with the routine **ClipboardQueryItem()**. If no supported formats are available, the object should provide the “no operation” feedback. However, if one or more is available, the object should determine whether the operation is a move or copy (call **ClipboardGetItemInfo()**) and act accordingly.

7.4

To provide feedback, the object must call **ClipboardSetQuickTransferFeedback()** in its method for MSG_META_PTR. This routine sets the mode of the transfer to one of the enumerated type **ClipboardQuickTransferFeedback**. If the format is not supported, **ClipboardSetQuickTransferFeedback()** is passed CQTF_CLEAR.

When the mouse has left an object's bounds, the object must relinquish its mouse grab. Either a MSG_META_CONTENT_LOST_GADGET_EXCLUSIVE or a MSG_META_PTR with the UIF_IN flag cleared will indicate this situation to the object. The former occurs when the mouse has moved onto a window or other object that is obscuring your object, and the latter is a result of the mouse moving outside of your bounds altogether.

At this point, the object must do two things: Reset the mouse cursor and re-transmit the last mouse pointer event.

To reset the mouse pointer, call **ClipboardSetQuickTransferFeedback()** and pass it CQTF_CLEAR. This will set the default cursor. To re-send the last pointer event received (you must do this because the last one occurred outside your object's bounds and might have been within another object's bounds), you simply have to return the flag MRF_REPLAY when releasing the mouse grab.

If the source object wishes a quick transfer to be able to be carried outside its view, it must send MSG_VIS_VUP_ALLOW_GLOBAL_TRANSFER to itself.

Process objects acting as a content must send `MSG_GEN_VIEW_ALLOW_GLOBAL_TRANSFER` to the GenView.

The object then becomes oblivious to future quick-transfer events until the pointer returns to its window (or unless it was registered for notification of quick-transfer conclusion).

7.4.4.2 Responsibilities of the Destination Object

7.4

`MSG_META_END_MOVE_COPY, ClipboardEndQuickTransfer()`

If, when the mouse pointer is within your object's bounds, the user releases the Move/Copy button, your object becomes the true destination of the transfer. You will be notified by a `MSG_META_END_MOVE_COPY`.

Upon receipt of this message, the object should first determine the move/copy/no-operation behavior as above, with one exception: If either of the flags `CQNF_MOVE` or `CQNF_COPY` is set for the transfer item, then the user has overridden the normal behavior and the destination object should respond with the appropriate operation.

After determining the proper action, the object should retrieve the transfer item (as it would from the Clipboard), passing one of `CQNF_MOVE`, `CQNF_COPY`, or `CQNF_NO_OPERATION`. This flag will cause the proper notification to be sent to the transfer's source and allow it to complete its actions properly. To finish the transfer, the object should call `ClipboardEndQuickTransfer()`.

7.4.4.3 Getting More Information

In addition to the routines above, you can use one other to retrieve information about a quick-transfer item. `ClipboardGetQuickItemInfo()` returns a set of handles for the transfer VM file and the file's header block.

7.4.4.4 When the Transfer Is Concluded

`MSG_META_CLIPBOARD_NOTIFY_QUICK_TRANSFER_CONCLUDED`

After the transfer has concluded, the original source of the transfer will receive `MSG_META_CLIPBOARD_NOTIFY_QUICK_TRANSFER_CONCLUDED`

if it requested notification when it registered the original transfer. This message will be accompanied by a **ClipboardQuickNotifyFlags** record indicating what type of operation the transfer ended up being. The source object should then follow the rules of quick transfer and act appropriately (e.g. delete the source object on a quick-move operation).

7.5 Shutdown Issues

7.5

```
ClipboardClearQuickTransferNotification(),  
ClipboardAbortQuickTransfer(),  
ClipboardRemoveFromNotificationList()
```

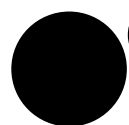
It is possible for a quick-transfer source object to shut down (be destroyed) before the completion of a quick-transfer operation. If the object registered to be notified of transfer completion, it must un-register as it is shutting down. This is done with **ClipboardClearQuickTransferNotification()** or **ClipboardAbortQuickTransfer()**.

Additionally, any application that registers with the Clipboard must un-register when it is shutting down. This is done with the routine **ClipboardRemoveFromNotificationList()**.

The Clipboard

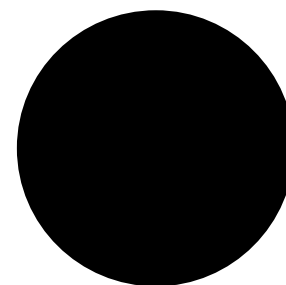
332

7.5



Concepts book

Localization



8

8.1	Localization Goals	335
8.2	How To Use Localization	337
8.3	Preparing for ResEdit	337
8.4	International Formats	340
8.4.1	Number and Measure	341
8.4.2	Currency	343
8.4.3	Quotation Marks	343
8.4.4	Dates and Times	344
8.4.5	Filters for Formats.....	346
8.5	Lexical Functions	348
8.5.1	Comparing Strings.....	348
8.5.2	String Length and Size	349
8.5.3	Casing.....	349
8.5.4	Character Categories	349
8.5.5	Lexical Values	350
8.5.6	DOS Text & Code Pages	350





Localization is the means by which GEOS adapts to foreign environments. The kernel automatically accounts for such country-dependent items as currencies and keyboard layouts. Your geodes can work with foreign character sets and formatting preferences. Using the **ResEdit** tool, anyone can quickly translate an executable's text into other languages. You should look over this chapter whether or not you're planning on distributing your software internationally. The chapter includes documentation for some functions which you're likely to use regardless.

8.1

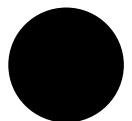
8.1 Localization Goals

The benefit of Geoworks' experience with previous international products was designed into GEOS from the start. The kernel was designed with the following three goals of localization in mind:

- ◆ **Ease of use**
Localization should be easy to use. A cumbersome system would slow programmers and leave them reluctant to use the system.
- ◆ **Integration with the kernel**
Functions used by many geodes should be part of the kernel. This allows for shared code, using less memory and making developers' jobs easier.
- ◆ **Economy**
Anyone should be able to translate an executable's text without having to understand or access the source code.

If you plan to release your software in other languages, you should code it for easy conversion later. Even if you have no such plans, there are still good reasons for using the localization features of GEOS.

- ◆ **Stay Generic and Adaptable**
By using localization code, your programs can work in a variety of environments without modification of source code. Even if you don't translate your geode into another language, users can still use it despite different keyboards and character sets.



- ◆ **Respect user's preferences**
Much of localization deals with user preferences which differ from country to country. Regardless of country, users may make changes using the Preferences desk tool (see Figure 8-1) and will no doubt appreciate having their preferences maintained.
- ◆ **Conform to the system**
You're probably going to end up using localization since much of GEOS is already localized. If your geodes are going to read DOS text files or alphabetize lists, you're going to use localized code; GEOS takes foreign character sets into account. Fortunately, localization functions are easy to use. In those situations where you don't absolutely have to use localized code, usually it's easier to use the localized functions than to write your own.

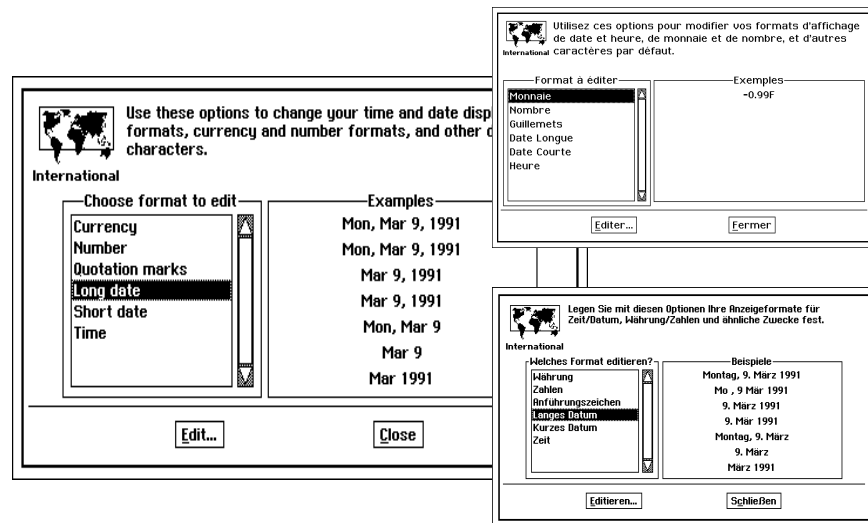


Figure 8-1 *International Formats*

International formats include variations in how numbers, times, currencies, etc. are displayed in various countries. These formats may be set in the International section of the Preferences desk tool.

8.2 How To Use Localization

Your localization workload will vary depending on whether you plan to release your software in other countries. If you don't plan on exporting your software, your only contact with localization will come when you use the common functions that are localized. If you plan on international distribution, you'll be using localization both when writing your geodes and later on, during the translation process, in the following steps:

8.2

- ◆ Use provided routines.
Several GEOS functions have been localized to handle foreign variations, and you should use these whether you plan international release or not. Most of these functions are concerned with "International formats" and character sets.
- ◆ Plan ahead for **ResEdit**.
You'll probably use the **ResEdit** tool if you translate your software into other languages. You'll have to be careful about how you write your programs if **ResEdit** is to work with them properly. Any text in your geode that the user may see will have to be translated. **ResEdit** looks only at chunk is in localizable resources. You should also be prepared for the translations of any string to be longer than the original. You may want to provide instructions for translators. You may include these instructions within the body of your source code, close to the strings that will be translated.
- ◆ Use **ResEdit**.
The **ResEdit** tool makes translation easy. It goes through an application's resources and checks all objects for strings and bitmaps, asking the translator to make any fitting changes. The translator never needs to see the source code and doesn't have to know how to program.



Warning

Be sure to put all text into localizable resources.

8.3 Preparing for ResEdit

ResEdit is a GEOS program which speeds geode translations. This section won't tell you how to use the **ResEdit** tool but will explain how to write your code if you want **ResEdit** to work with it correctly.



The important thing to remember is to put all text and bitmaps to be localized in localizable resources. **ResEdit** looks for localizable objects only in non-code resources. The string or bitmap itself must be stored in a chunk, which you may insure by using either the **@chunk** or **@visMoniker** keywords. Thus, your application icons are probably stored in the following way:

8.3

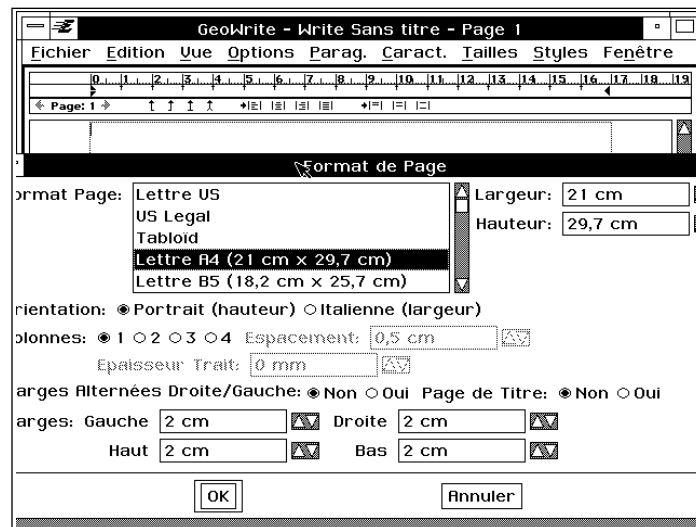
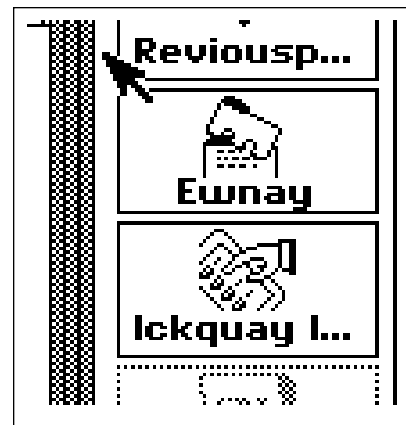


Figure 8-2 Localizing Strings
Some strings may get when translated. If your strings are too large, UI gadgets may extend beyond screen boundaries, as shown in the above screen dump, where a dialog box has become too wide. If you set an absolute size for your UI gadgets, strings may not fit inside them and instead may be truncated, as in the buttons pictured to the right.



```
@start APPSCMONIKERRESOURCE, data
/* Application moniker: */
@visMoniker MyAppSCMoniker = { /* ... */}
@end APPSCMONIKERRESOURCE
```

At first it might seem surprising that bitmaps should be localized, since pictures are supposed to be a sort of universal language. However, this attitude may seem less strange when you try to distribute software to a culture that finds pictures of yellow arrows offensive.

8.3

Your application could keep its strings in a resource in the style illustrated in Code Display 8-1.

Code Display 8-1 Storing Strings in Localizable Resources

```
@start ErrorStrings, data;
@chunk char NoMonkeyError[] =
    "FTPOOMM failed: No monkey present. Acquire a monkey and try again.";
@chunk char NoPeanutError[] =
    "FTPOOMM failed: No peanuts present. Please insert peanuts and try again.";
@end ErrorStrings;
```

When working with assembly language, this means that any object stored in a **code** resource, **idata**, or **udata** will be passed over in the search for localizable resources, and thus will not be localizable.

When you know you are storing your strings such that they are localizable, you can provide information which will be visible to the translator when they use the **ResEdit** tool. Use the **@localize** keyword (**localize** in assembly) to set up this help text. The **@localize** directive should directly follow the chunk it applies to.

Remember the **@localize** syntax:

```
@localize { <string> <min>-<max> };  
@localize { <string> <length> };  
@localize { <string> };  
@localize <string>;  
@localize not;
```

8.4

Code Display 8-2 Storing Strings in Localizable Resources

```
@start ErrorStrings, data  
  
@chunk char NoMonkeyError[] =  
    "FTPOOMM failed: No monkey present. Acquire a monkey and try again";  
  
@localize "The phrase \"acquire a monkey\" appears in another string. Both should \  
be translated in the same way.";   
  
@end ErrorStrings  
  
@object GenGlyphClass BossMon = {  
    GI_visMoniker = "Boss";  
    @localize { "This means hide the game because a boss is coming" 3-6 }; }
```

Another thing to keep in mind is that when strings are translated, they are likely to grow 33% to 50%. You have to remember to leave room for larger strings, both in memory and in UI. If you use the usual generic UI gadgetry, the geometry manager should stretch the various gadgets to fit any larger names. You should be careful that it doesn't have to stretch so far that components get lost off the edge of the screen. If you decide to get around this problem by constraining the size of some gadgetry, keep in mind that if you don't allow it to stretch, your new strings may not fit. See Figure 8-2 for illustrations of these problems.

8.4 International Formats

"International Formats" generally refers to formats which differ from country to country. In GEOS, it signifies those formats which the user can set in the International section of the Preferences desk tool. GEOS provides functions to work with International Formats.

8.4.1 Number and Measure

```
LocalGetNumericFormat(), LocalSetNumericFormat()  
LocalGetMeasurementType(), LocalSetMeasurementType(),  
LocalAsciiToFixed(), LocalFixedToAscii(),  
LocalDistanceToAscii(), LocalDistanceFromAscii()
```

In the USA, 3.142 is a little more than three. In some other countries, it's a little over three thousand. This discrepancy arises from the fact that the decimal and thousands separators are interchanged—in some countries the “.” symbol takes the place of the “,” symbol when expressing numbers. To allow for local number formats, GEOS provides functions for retrieving and setting the user's preferences.

8.4

The number format includes the thousands separator, decimal separator, list separator, and number of decimal digits, as shown in Figure 8-3. The number format also contains a record, **NumberFormatFlags**, which holds one flag, `NFF_LEADING_ZERO`. This flag is on if the user wants a leading zero.

There is no overall automatic formatting command for numbers. GEOS supports many internal numeric formats such as fixed point and floating point. If the corresponding math library doesn't contain a formatting command for the number format you're using, you will need to work directly with the localization functions.

The **LocalGetNumericFormat()** routine returns the numeric format so you may use it to do your own formatting. **LocalSetNumericFormat()**



allows you to reset the user's preferences, though this is inadvisable as it overrides and erases the user's original settings.

8.4

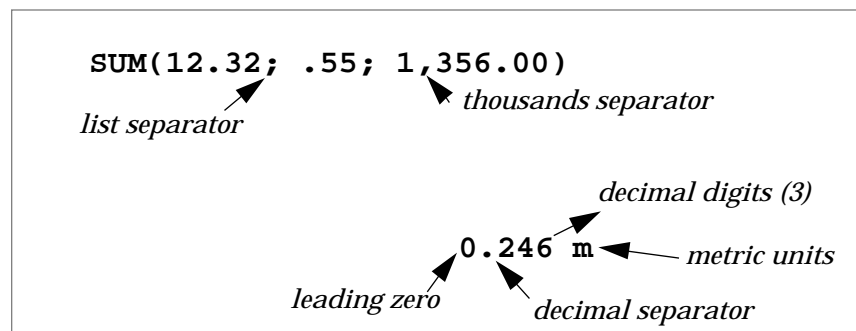


Figure 8-3 *Parts of a Number Format*

These routines use the **LocalNumericFormat** structure to store the number format data.

The **LocalGetMeasurementType()** and **LocalSetMeasurementType()** routines work with the enumerated type **MeasurementType**, of which there are two: **MEASURE_US** and **MEASURE_METRIC**.

LocalGetMeasurementType() retrieves the user's current preferred measurement type; **LocalSetMeasurementType()** sets the user's preference to the passed value. As with any command that resets the user's preference, **LocalSetMeasurementType()** should never be used by most geodes.

The **LocalAsciiToFixed()** routine converts an ascii string (e.g. "12.7") to a fixed point number. **LocalFixedToAscii()** performs the reverse.

To convert between a string like "72 pt" and a number representing a distance, use the **LocalDistanceFromAscii()** routine.

LocalDistanceToAscii() goes the other way, constructing the string corresponding to a given distance. These routines use the **DistanceUnit** enumerated type to specify the measurement units which are used as shown in the table below.

When allocating strings to use as the targets for **Local...ToAscii()** routines, allocating a buffer of size **LOCAL_DISTANCE_BUFFER_SIZE** will be sufficient.

Table 8-1 *DistanceUnit* types

DistanceUnit value	# of Points	Display Format	Entry Format(s)
DU_POINTS	1.000	###.### pt	###.### pt
DU_INCHES	72.000	###.### in	###.### in ###.###"
DU_CENTIMETERS	28.346	###.### cm	###.###cm
DU_MILLIMETERS	2.835	###.### mm	###.###mm
DU_PICAS	12.000	###.### pi	###.### pi
DU_EUR_POINTS	1.065	###.### ep	###.### ep
DU_CICEROS	12.787	###,### ci	###.### ci
DU_POINTS_OR_MILLIMETERS and DU_INCHES_OR_CENTIMETERS: special cases			

8.4

8.4.2 Currency

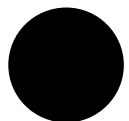
`LocalGetCurrencyFormat()`, `LocalSetCurrencyFormat()`

These functions allow you to work with the currency format (see Figure 8-4). This format consists of the currency symbol string, number of currency digits (which may be different from the number of digits for ordinary numbers), and the **CurrencyFormatFlags** record. The flags in this structure determine whether the currency format includes space around the symbol, a negative sign, or a leading zero. There are flags to determine the relative order of the negative sign, currency symbol, and number. The currency symbol string may be of up to length `CURRENCY_SYMBOL_LENGTH` including the null terminator. **LocalGetCurrencyFormat()** also returns the separator characters from the numeric formats, eliminating the need for a separate call to **LocalGetNumericFormat()**. **LocalSetCurrencyFormat()** allows the changing of the set preferences. Most parts of the currency format are stored in a **LocalCurrencyFormat** structure when passed to or returned by these functions.

8.4.3 Quotation Marks

`LocalGetQuotes()`, `LocalSetQuotes()`

In different countries, people use different types of quotation marks. Germans, for example, might use the » and « characters. Although traditionally, the only computer-generated quotation marks available are “



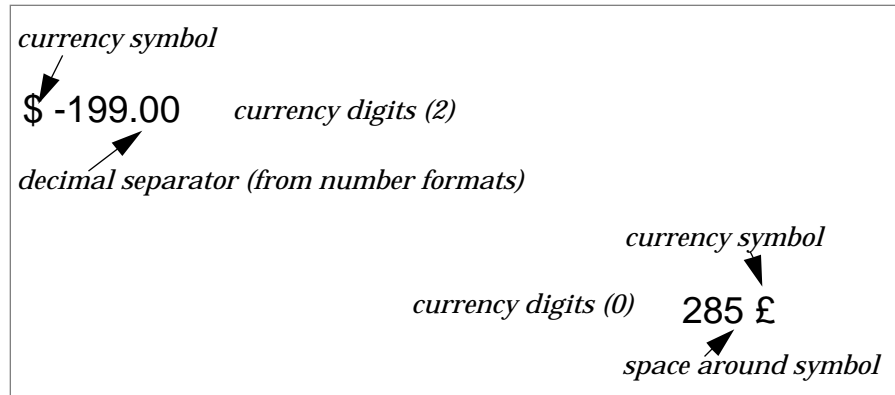


Figure 8-4 *Parts of the Currency Format*

and ", GEOS allows all geodes to use native quotation marks without extraordinary effort.

If a user types in a quotation mark, the keyboard knows which character to print; thus, if you're using **ResEdit** on a string, it's easy to have your translator type in the correct character. In these cases your geode will have no use for **LocalGetQuotes()** or **LocalSetQuotes()**. On the other hand, if your program is going to construct a string which contains quotation marks, use the **LocalGetQuotes()** routine; it will return the correct characters to use. If you want to reset the preferences, use the **LocalSetQuotes()** routine.

These routines work with the **LocalQuotes** structure, which simply holds four characters to use as the four kinds of quotation mark.

8.4.4 Dates and Times

```
LocalFormatDateTime(), LocalParseDateTime(),
LocalGetDateTimeFormat(), LocalSetDateTimeFormat(),
LocalCustomFormatDateTime(), LocalCustomParseDateTime(),
LocalCalcDaysInMonth(), LocalFormatFileDateTime()
```

LocalFormatDateTime() and **LocalParseDateTime()** allow display and parsing of date and time strings in a variety of formats. Passed the appropriate data, **LocalFormatDateTime()** will return a string with the

data appropriately formatted. **LocalParseDateTime()** performs the reverse function, reading a string and attempting to extract information from it. The strings these functions use should be long enough to hold a formatted date or time. The predefined constant `DATE_TIME_BUFFER_SIZE` is the minimum recommended number of characters for one of these strings.

Localization functions use **DateTimeFormats** to keep track of what sort of format is desired. If told to parse "5:28" as hours and minutes,

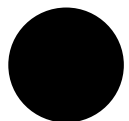
LocalParseDateTime() will (depending on the user's time format preferences) return 5 in the hours place and 28 in the minutes place. If told to parse "5/28" as hours and minutes, **LocalParseDateTime()** would return an error. There are a variety of **DateTimeFormats()** to choose from Table 8-2, and you should decide which format to use based on what information you want your application to display or read.

8.4

Table 8-2 *DateTimeFormat Types*

Format	Sample Output
DTF_LONG	Sunday, March 5th, 1990
DTF_LONG_CONDENSED	Sun, Mar 5, 1990
DTF_LONG_NO_WEEKDAY	March 5th, 1990
DTF_LONG_NO_WEEKDAY_CONDENSED	Mar 5, 1990
DTF_SHORT	3/5/90
DTF_ZERO_PADDED_SHORT	03/05/90
DTF_MD_LONG	Sunday, March 5th
DTF_MD_LONG_NO_WEEKDAY	March 5th
DTF_MD_SHORT	3/5
DTF_MY_LONG	March 1990
DTF_MY_SHORT	3/90
DTF_MONTH	March
DTF_WEEKDAY	Wednesday
DTF_HMS	1:05:31 PM
DTF_HM	1:05 PM
DTF_H	1 PM
DTF_MS	5:31
DTF_HMS_24HOUR	13:05:31
DTF_HM_24HOUR	13:05

DateTimeFormats with sample strings. Note that these samples only illustrate what the U.S. version could produce. The actual output might not look like the samples.





Advanced Topic

You probably won't be working with DTF Strings unless you make up your own DT Formats.

8.4

Each **DateTimeFormat** has a string associated with it that contains formatting information. This string consists of the characters of the format, with place-holding tokens for the fields of the date or time. These tokens are delimited by the special character “|”. For example, the date/time format string “|HH|:|mm| |ap|” would correspond to the formatted string “10:37 am”. The string for each **DateTimeFormat** can be accessed or altered with the **LocalGetDateTimeFormat()** and **LocalSetDateTimeFormat()** commands. The recommended minimum size of a date/time format string is the constant `DATE_TIME_FORMAT_SIZE`.

If you don't wish to work through one of the standard **DateTimeFormat** values, you can construct a string containing some date/time tokens and pass it as an argument to **LocalCustomFormatDateTime()**, along with the appropriate date and time information. To parse a date/time using a custom format, use **LocalCustomParseDateTime()**.

The **LocalFormatFileDateTime()** utility routine works in the same way as **LocalFormatDateTime()**, except it takes a **FileDateAndTime** argument. It is normally used to format the information returned by **FileGetDateAndTime()**.

The **LocalCalcDaysInMonth()** utility routine takes a month number and returns the number of days in that month.

A number of constants have been set up to aid in the computation of the size of string necessary to hold a date/time or a date/time format string. These constants are `DATE_TIME_BUFFER_SIZE`, `DATE_TIME_FORMAT_SIZE`, `MAX_MONTH_LENGTH`, `MAX_DAY_LENGTH`, `MAX_YEAR_LENGTH`, `MAX_WEEKDAY_LENGTH`, `MAX_SEPARATOR_LENGTH`, and `TOKEN_LENGTH`.

8.4.5 Filters for Formats

`LocalIsDateChar()`, `LocalIsTimeChar()`, `LocalIsNumChar()`

The generic UI allows for filters on text objects to keep users from typing inappropriate characters. For example, if the user were expected to type in a number, the “~” character would be ignored as that isn't part of the number format. If you want to use generic text objects, you can give them hints to accept only characters from certain formats. If you want to use the filtering

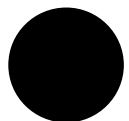
functions for your own purposes, they are available globally. When passed a character, these functions will tell whether the character is a legal part of a format. **LocalIsDateChar()** returns TRUE if the passed character is part of the short date format, DTF_SHORT. **LocalIsTimeChar()** returns TRUE when passed a number or time separator character. **LocalIsNumChar()** returns TRUE when passed a number character, a separator, or negative sign.

8.4

Table 8-3 *DateTimeFormat String Tokens*

Token	Constant	Example
DD	TOKEN_TOKEN_DELIMITER	
LW	TOKEN_LONG_WEEKDAY	Wednesday
SW	TOKEN_SHORT_WEEKDAY	Wed
SM	TOKEN_SHORT_MONTH	Jun
NM	TOKEN_NUMERIC_MONTH	6
ZM	TOKEN_ZERO_PADDED_MONTH	06
LD	TOKEN_LONG_DATE	9th
SD	TOKEN_SHORT_DATE	9
PD	TOKEN_SPACE_PADDED_DATE	9
LY	TOKEN_LONG_YEAR	1991
SY	TOKEN_SHORT_YEAR	91
HH	TOKEN_12HOUR	6
hh	TOKEN_24HOUR	18
Zh	TOKEN_ZERO_PADDED_24HOUR	04
mm	TOKEN_MINUTE	5
Zm	TOKEN_ZERO_PADDED_MINUTE	05
Pm	TOKEN_SPACE_PADDED_MINUTE	5
ss	TOKEN_SECOND	3
ap	TOKEN_AM_PM	pm
Ap	TOKEN_AM_PM_CAP	Pm
AP	TOKEN_AM_PM_ALL_CAPS	PM

Examples show what U.S. version could produce. Actual output might not look like samples.



8.5 Lexical Functions

Since different languages have different alphabets, GEOS has to allow for characters not in the standard English character set. Thus, many standard procedures have been localized. Most of these functions have to do with the lexical value of characters, their place in a lexical, or alphabetic, ordering. These lexical values take the place of ASCII standard character values that you may be used to.

8.5

8.5.1 Comparing Strings

`LocalCmpStrings()`, `LocalCmpStringsNoCase()`,
`LocalCmpStringsNoSpace()`, `LocalCmpStringsNoSpaceCase()`,
`LocalGetLanguage()`, `LocalCmpChars()`, `LocalCmpCharsNoCase()`

LocalCmpStrings() takes two strings as arguments and says which, if either, comes first alphabetically. **LocalCmpStringsNoCase()** does the same thing but is not case sensitive. **LocalCmpStrings()** does a better job of ordering strings than assembly language instructions such as **cmps** that just compare ASCII values. Since the ASCII value of “a” places it after “Z”, it is advisable to use **LocalCmpStrings()**, which uses the localized lexical values. When comparing two strings for equality, assembly instructions like **cmps** may be used safely.

The **LocalCmpStringsNoSpace()** and **LocalCmpStringsNoSpaceCase()** routines are like the **LocalCmpStrings()** and **LocalCmpStringsNoCase()** routines, except that spaces and punctuation marks are ignored. Note that if you are comparing only a certain number of characters that spaces and punctuation marks will not be included in this number.

The sort order of the strings will depend on the language used when sorting them. To find out the current language used for sorting, call **LocalGetLanguage()**.

There are two assembly routines **LocalCmpChars()** and **LocalCmpCharsNoCase()** which allow for the quick lexical comparison of two characters.

8.5.2 String Length and Size

`LocalStringLength()`, `LocalStringSize()`

There are two routines which determine how long a string is.

LocalStringLength() returns the number of characters making up the string, not including the null terminator, if any. **LocalStringSize()** returns the number of bytes in a string, again not counting any null terminators.

Normally these two values will be the same, but any applications which want to support double byte character support will need separate functions to handle those characters that take more than one byte to represent. 8.5

8.5.3 Casing

`LocalUppcaseChar()`, `LocalDowncaseChar()`,
`LocalUppcaseString()`, `LocalDowncaseString()`



Warning

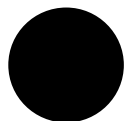
Programmers used to using ASCII values to change a character's case should use our functions instead.

These functions take a character or string and, if possible, transform it to upper or lower case. They work only on alphabetic characters; other characters will be returned unchanged. These functions can deal with all languages, understanding “È” is the upper case for “è”. If you're used to using ASCII, you might be tempted to write your own version of these casing functions by doing arithmetic operations with the lexical or ASCII values of characters, but chances are these will not work with other character sets and should be avoided.

8.5.4 Character Categories

`LocalIsUpper()`, `LocalIsLower()`, `LocalIsAlpha()`,
`LocalIsPunctuation()`, `LocalIsSpace()`, `LocalIsSymbol()`,
`LocalIsControl()`, `LocalIsDigit()`, `LocalIsHexDigit()`,
`LocalIsAlphaNumeric()` `LocalIsPrintable()`, `LocalIsGraphic()`

GEOS assembly provides some commonly used character predicates. C users may use the standard C functions **isupper()**, **islower()**, etc. to access these predicates. Passed a character, each of these functions tells whether that character falls into a certain category. **LocalIsUpper()** returns *true* if the character is an uppercase alphabetic character; **LocalIsLower()** returns



true if the character is lower case. **LocalIsAlpha()** approves any alphabetic character. **LocalIsPunctuation()** checks for punctuation marks.

LocalIsSpace() looks for white space (including spaces, tabs, and carriage returns), and **LocalIsSymbol()** catches just about everything else.

LocalIsControl() detects control characters (e.g. Control-A).

LocalIsDigit() returns *true* for decimal digits. **LocalIsHexDigit()** approves hexadecimal digits (including the characters a-f, A-F).

LocalIsAlphaNumeric() detects alphabetic characters and decimal digits.

LocalIsPrintable() returns *true* when passed a character which takes up a space when printed, corresponding to the standard C function **isprint()**.

LocalIsGraphic() checks for displayable characters, in the manner of the standard C function **isgraphic()**.

8.5.5 Lexical Values

`LocalLexicalValue()`, `LocalLexicalValueNoCase()`

If for some reason you want to get the lexical value of a character, these functions will return it. **LocalLexicalValue()**, passed a character, returns its lexical value. **LocalLexicalValueNoCase()** does the same thing but ignores case. If the lexical value for one character is lower than another, that character comes first alphabetically. For instance, “a” would have a lower lexical value than “z.” This ordering will be valid in any language using the same character set.

8.5.6 DOS Text & Code Pages

`LocalDosToGeos()`, `LocalGeosToDos()`, `LocalDosToGeosChar()`,
`LocalGeosToDosChar()`, `LocalCmpStringsDosToGeos()`,
`LocalIsDosChar()`, `LocalCodePageToGeos()`,
`LocalGeosToCodePage()`, `LocalCodePageToGeosChar()`,
`LocalGeosToCodePageChar()`, `LocalGetCodePage()`,
`LocalIsCodePageSupported()`

There are several functions which work with DOS-format text, converting it to and from GEOS format. Normally, converting DOS text files to GeoWrite documents is handled by Import/Export routines; for those occasions where

your code needs to convert DOS text to GEOS format or vice versa, however, these functions should be sufficient. They are localized because DOS has its own version of localization which it implements as “code pages.” Each code page is a table of characters. Countries or regions with unusual characters have different code pages. If you haven’t worked with code pages up to this point, you probably won’t have to start now but can just rely on GEOS to use the native code page correctly. If your geode is going to allow computers from one country to communicate with those of another, you might need to use specialized functions.

8.5

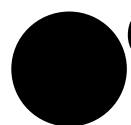
Assuming you want to use the code page native to the user’s machine, you can use **LocalDosToGeos()** and **LocalGeosToDos()** to convert strings from one text format to the other. **LocalDosToGeosChar()** and **LocalGeosToDosChar()** work similarly, converting a single character.

LocalCmpStringsDosToGeos() takes two strings, converts one or both of them to GEOS format, then compares them, returning the same values as the regular string comparing functions. Pass a **LocalCmpStringsDosToGeosFlags** to specify which strings to convert before comparing.

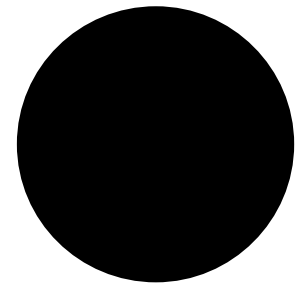
Each of these functions also takes a character as an argument. When a character in one format has no corresponding character in the format it’s being converted to, the character argument will be substituted. At the same time a flag will be set to let your geode know that there were some characters that couldn’t be converted nicely. **LocalIsDosChar()** checks a GEOS character to see if it maps into a DOS character. The **MIN_MAP_CHAR** constant is the smallest value which may need to be mapped. If a character’s value is below this constant, then it will be the same under any code page.

If your geode is interested in which code page it’s using, the **LocalGetCodePage()** instruction will tell you what the default code page is. If you want to use a code page other than the native one, the **LocalCodePageToGeos()**, **LocalGeosToCodePage()**, **LocalCodePageToGeosChar()**, and **LocalGeosToCodePageChar()** behave as their default code page equivalents, except that each takes a code page as an argument to be used in conversion. A number of support code page values are enumerated in the **DosCodePage** type. To find out whether the user’s environment supports a specific code page, call **LocalIsCodePageSupported()**.



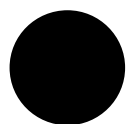


General Change Notification



9

9.1	Design Goals.....	355
9.2	The Mechanics of GCN.....	356
9.3	System Notification	357
9.3.1	Registering for System Notification	357
9.3.2	Handling System Notification	360
9.3.3	Removal from a System List	364
9.4	Application Local GCN Lists	364
9.4.1	Creating Types and Lists	366
9.4.2	Registering for Notification	366
9.4.3	Handling Application Notification.....	368
9.4.4	Removal from Application Lists	371





In a multitasking environment, threads may need to know of condition changes that might affect them. In most cases where shared resources or multiple threads of execution exist, processes and objects must be sure of the integrity of data that they depend on and must be sent notice when that data changes.

In GEOS, this functionality is provided through the General Change Notification (GCN) mechanism. Although one could set up messages between processes and objects manually, the GCN mechanism eliminates the need to keep track of all processes that depend on the particular change and to keep track of all messages sent out to the various processes and objects.

9.1

9.1 Design Goals

General Change Notification allows you to keep track of both system and application events. Objects or processes interested in a particular change may request the GCN mechanism to notify them when that change occurs. That change may be system-wide (such as a file system change) or application-specific (such as a text style change within a word processor). The GCN mechanism allows objects or processes to sign up for such notification and intercept messages sent by the system (or the application) so that you may respond to different changes on a case by case basis.

The manners in which you sign up for these two types of notification differ although the functionality of the notification process is similar. The most straightforward way is to use a **gcnList** field in an application's application object; this is the usual approach used to add an application's GenPrimary object to the application's window list. We will assume that you know how to add an object to a GCN list in this manner, as it is shown in most if not all of the sample applications.

You can also sign up for system-wide notification through the use of certain routines and intercept system messages when the change occurs. Other objects may sign up for application-specific notification supported by

GenApplicationClass. These application specific notifications should only be sent to the GenApplication object.

9.2 **The Mechanics of GCN**

9.2

The basic GCN functionality manages lists of objects that are interested in specific changes. For each particular change that needs to be monitored, a separate list is needed. A completely separate “list of lists” containing an inventory of all GCN lists is also created. This will serve as the “search” table, while the particular GCN list will serve as the “messaging” list. When an event is detected, the GCN mechanism will search through the list of lists, seeing if a notification list is interested in a particular change, and send the appropriate messages if the objects do indeed wish notification of the event.

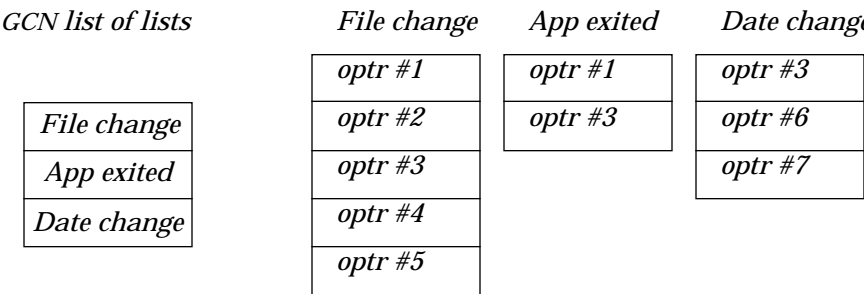


Figure 9-1 *A GCN List of Lists*
Organization of a GCN list of lists and several GCN lists.

There are several reasons why you would want to use GCN:

- ◆ **Ease of use**
The GCN mechanism eliminates the need to monitor and dispatch messages relating to system changes.
- ◆ **Commonality**
The GCN mechanism provides a common platform for communication between applications in certain cases.
- ◆ **The system expects you to**
Many messages sent by the system expect a GCN mechanism to intercept

them. Although you can intercept these messages manually, it is easier to take advantage of GCN's built-in functions.

9.3 System Notification

`GCNListAdd()`, `GCNListSend()`

The system provides several lists for common system changes which might affect your application. After signing up on one of these lists (for example, the file change list) you will be notified by the kernel whenever the specified change occurs. In most cases, all you need to do is register for notification with **GCNListAdd()** and intercept the kernel's notification message.

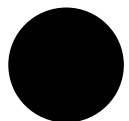
9.3

The GCN mechanism performs its functions through a common series of steps. These steps are:

- 1 The object registers for notification with **GCNListAdd()**.
- 2 The change occurs.
- 3 The GCN mechanism is informed of the change by the acting party (in most cases this is the system itself, although a library may also send notifications). Applications do not send notifications at the system level.
- 4 The GCN mechanism dispatches notification messages to all interested parties with **GCNListSend()**.
- 5 The object is informed of the change.
If you need to perform some work related to this change, you should have a message handler to intercept the system messages.

9.3.1 Registering for System Notification

Whenever an object or process needs to be notified of some system change, it must call the routine **GCNListAdd()** to add itself to the list for that particular change. **GCNListAdd()** finds the appropriate general change notification list—creating a new one if none currently exists—and adds the `optr` of the new object to the end of that list. You may add the `optr` to the GCN list at any time during the process' or object's life, but it is usually convenient for a process to be added in its `MSG_GEN_PROCESS_OPEN_APPLICATION` or



for an object that is on the active list to be added in its MSG_META_ATTACH handler.

Each optr in a GCN list should have a notification ID attached to it. (The combination of a manufacturer ID and a notification type comprises an element's specific notification ID.) **GCNListAdd()** must be passed the optr of the object to add, along with a notification ID. For each separate notification ID, a separate GCN list is needed and will be created automatically.

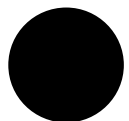
9.3

The GCN routines use a word of data, **GCNStandardListType**, to represent the notification type. The currently recognized **GCNStandardListType** types for MANUFACTURER_ID_GEOWORKS are

- ◆ GCNSLT_FILE_SYSTEM
This **GCNStandardListType** is used for notification of a file system change. Parties on this list will receive the system messages MSG_NOTIFY_FILE_CHANGE and MSG_NOTIFY_DRIVE_CHANGE.
- ◆ GCNSLT_APPLICATION
This **GCNStandardListType** is used for notification of a starting or exiting application. Parties on this list will receive the system messages MSG_NOTIFY_APP_STARTED and MSG_NOTIFY_APP_EXITED.
- ◆ GCNSLT_DATE_TIME
This **GCNStandardListType** is used for notification of a date/time change in the system's internal clock. Note that this will not tell you about timer ticks—the only time changes that will come up are those resulting from system restarts and time changes by the user. Parties on this list will receive the system message MSG_NOTIFY_DATE_TIME_CHANGE. This message does not pass any further information, so your message handler should be able to take care of any changes by itself (such as calling the internal clock for an updated value).
- ◆ GCNSLT_DICTIONARY
This **GCNStandardListType** is used for notification of a user dictionary change. Parties on this list will receive the system message MSG_NOTIFY_USER_DICT_CHANGE.
- ◆ GCNSLT_KEYBOARD_OBJECTS
This list is used for notification when the user has chosen a different keyboard layout. Parties on this list will receive the system message MSG_NOTIFY_KEYBOARD_LAYOUT_CHANGE.

- ◆ GCNSLT_EXPRESS_MENU_CHANGE
This **GCNStandardListType** notifies various system utilities that an express menu has been created or destroyed. The recipient receives the optr of the Express Menu Control. This list should be used in conjunction with the GCNSLT_EXPRESS_MENU_OBJECTS list. Objects on this list receive MSG_NOTIFY_EXPRESS_MENU_CHANGE, which itself passes a **GCNExpressMenuNotificationType** (either GCNEMNT_CREATED or GCNEMNT_DESTROYED) and the optr of the Express Menu Control affected.
- ◆ GCNSLT_INSTALLED_PRINTERS
This list notifies objects when a printer is either installed or removed. The recipient of MSG_PRINTER_INSTALLED_REMOVED might want to call **SpoolGetNumPrinters()** to learn if any printer or fax drivers are currently installed.
- ◆ GCNSLT_SHUTDOWN_CONTROL
This **GCNStandardListType** is used for system shutdown control. Parties on a list of this type will receive the system message MSG_META_CONFIRM_SHUTDOWN which itself passes a **GCNShutdownControlType** (either GCNSCT_SUSPEND, GCNSCT_SHUTDOWN, or GCNSCT_UNSPEND). Shutdown Control is documented in “Applications and Geodes,” Chapter 6.
- ◆ GCNSLT_TRANSFER_NOTIFICATION
This list notifies objects that a transfer item within the clipboard has changed (or been freed). Parties on this list will receive the system message MSG_META_CLIPBOARD_NOTIFY_NORMAL_TRANSFER_ITEM_CHANGED and MSG_META_CLIPBOARD_NOTIFY_TRANSFER_ITEM_FREED.
- ◆ GCNSLT_EXPRESS_MENU_OBJECTS
This list contains all Express Menu Control objects in the system. Typically this list is used to add a control panel item or a DOS task list item to all express menu Control objects. This can be done by sending MSG_EXPRESS_MENU_CONTROL_CREATE_ITEM to the GCN list with **GCNListSend()**.
- ◆ GCNSLT_TRANSPARENT_DETACH
This list contains all application objects that may be transparently detached if the system runs short of heap space, in least recently used (LRU) order. This list should only be used if transparent launch mode is in use.

9.3



General Change Notification

360

9.3

- ◆ GCNSLT_TRANSPARENT_DETACH_DA
This list contains a list of transparently detachable desk accessories if the system runs short of heap space. This list should only be used if transparent launch mode is in use. Objects should not be detached unless all detachable, full-screen applications have been detached.
- ◆ GCNSLT_REMOVABLE_DISK
This list is used to store all application and document control objects that originate from a removable drive. If the disk they originate on is removed, they will be notified to shut themselves down with MSG_META_DISK_REMOVED.

These pre-defined notification types are intended only for use with MANUFACTURER_ID_GEOWORKS. Other manufacturers wishing to intercept their own system changes must define their own change types under their respective manufacturer IDs if they are unable to use MANUFACTURER_ID_GEOWORKS.

Code Display 9-1 Adding a Process Object to a GCN List

```
@method MyProcessClass, MSG_GEN_PROCESS_OPEN_APPLICATION {
    optr          myThread;

    @callsuper;          /* Do default MSG_GEN_PROCESS_OPEN_APPLICATION */

    /* Casts the return value for the process handle into an optr */
    myThread = ConstructOptr(GeodeGetProcessHandle(), NullChunk);

    /* myThread (the process) is added to notification of file changes */

    GCNListAdd(myThread, MANUFACTURER_ID_GEOWORKS, GCNSLT_FILE_SYSTEM);
}
```

9.3.2 Handling System Notification

MSG_NOTIFY_FILE_CHANGE, MSG_NOTIFY_DRIVE_CHANGE,
MSG_NOTIFY_APP_STARTED, MSG_NOTIFY_APP_EXITED,

```
MSG_NOTIFY_USER_DICT_CHANGE ,
MSG_NOTIFY_EXPRESS_MENU_CHANGE
```

When an identified change occurs, either the system (or a library) will call the routine **GCNListSend()**, passing it the appropriate notification message. This routine scans the list of all GCN lists and dispatches notification to all appropriate objects that had requested knowledge of the specified change. If any additional information relating to the change cannot be included in the message, the system allows **GCNListSend()** to pass data in a global heap block. For example, additional information about a file change (name of file, etc.) must be passed in a global heap block.

9.3



Do not free the global heap block (if any) yourself. It will be freed by the system.

The object or process originally requesting notification of the change should be required to handle the appropriate message. If additional data about the change is passed in a global heap block, the process should access that information with **MemLock()** and **MemUnlock()**. You should always call the process's superclass in your message handler to make sure that the global heap block will be automatically freed by **MetaClass**. Therefore, do not free a global heap block manually in a notification handler.

The system provides several messages which you may want to handle. These messages provide notification of file system changes, application start-up or shut-down, system clock changes, etc. These messages are mentioned with the list they correspond to in "Registering for System Notification" on page 357. Messages which require more detailed explanation are also mentioned below.

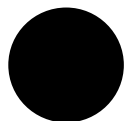
The kernel sends `MSG_NOTIFY_FILE_CHANGE` whenever a change to the file system occurs. All objects signed up on the GCN list `GCNSLT_FILE_CHANGE` will receive this message.

`MSG_NOTIFY_FILE_CHANGE` passes a **FileChangeNotificationType** specifying the change that occurred. Some types indicate the presence of a data block (of type **FileChangeNotificationData**) containing, if applicable, the name of the file changed, the disk handle of the file changed, and the ID of the affected file.

The notification type should be one of the following:

`FCNT_CREATE`

This type indicates that a file or directory was created.



General Change Notification

362

FCND_id stores the ID of the containing directory; *FCND_name* contains the name of the new file or directory that was created.

FCNT_RENAME

This type indicates that a file or directory was renamed.

FCND_id stores the ID of the file or directory that was renamed; *FCND_name* contains its new name.

FCNT_OPEN

9.3

This type indicates that a file was opened. *FCND_id* stores the ID of the file. *FCND_name* is undefined, and may or may not be present. (You can check the size of the block to see if it is indeed present.) This notification type is generated after a call to **FileEnableOpenCloseNotification()**.

FCNT_DELETE

This type indicates that a file or directory was deleted.

FCND_id stores the ID of the file or directory that was deleted. *FCND_name* is undefined and may or may not be present.

FCNT_CONTENTS

This type indicates that a file's contents have changed.

FCND_id stores the ID of the file. *FCND_name* is undefined and may or may not be present. This notification type is generated after a call to **FileCommit()** or **FileClose()** that results in a file modification.

FCNT_ATTRIBUTES

This type indicates that a file's attributes have changed.

FCND_id stores the ID of the file. *FCND_name* is undefined and may or may not be present. This notification type is generated upon completion of all changes in a **FileSetAttributes()**, **FileSetHandleExtAttributes()**, or **FileSetPathExtAttributes()** call.

FCNT_DISK_FORMAT

This type indicates that a disk has been formatted. Both *FCND_id* and *FCND_name* are undefined and may not be present.

FCNT_CLOSE

This type indicates that a file has been closed. *FCND_id* stores the identifier of the file. *FCND_name* is undefined and may not be present. This notification type is generated only after a call to **FileEnableOpenCloseNotification()**.

FCNT_BATCH

This type indicates that this file change notification is actually a group of notifications batched together. In this case, MSG_NOTIFY_FILE_CHANGE passes the MemHandle of a **FileChangeBatchNotificationData** block instead. This data block stores a batch of **FileChangeBatchNotificationItem** structures, each referring to an operation (with its own notification type, disk handle, file name, and file ID). Note that in this batched case, you must assume that all file names and file IDs that are optional (i.e. are undefined) are not present.

9.3

FCNT_ADD_SP_DIRECTORY

This type indicates that a directory has been added as a **StandardPath**. *FCND_disk* contains the **StandardPath** that was added. This notification type is generated after a call to **FileAddStandardPathDirectory()**.

FCNT_DELETE_SP_DIRECTORY

This type indicates that a directory has been deleted as a **StandardPath**. *FCND_disk* contains the **StandardPath** that was deleted. This notification type is generated after a call to **FileDeleteStandardPathDirectory()**.

You may access this data (after locking the block) and perform whatever actions you need within your message handler.

The kernel also sends MSG_NOTIFY_DRIVE_CHANGE to GCN lists of type GCNSLT_FILE_CHANGE. This message passes a **GCNDriveChangeNotificationType** specifying whether a drive is being created or destroyed and the ID of the affected drive.

The kernel sends MSG_NOTIFY_APP_STARTED whenever any application starts up within the system and MSG_NOTIFY_APP_EXITED whenever an application shuts down. All objects signed up on the GCN list GCNSLT_APPLICATION will receive these messages after the change occurs. MSG_NOTIFY_APP_STARTED passes the MemHandle of the application starting up, which you may access to perform any required actions. In a similar manner, MSG_NOTIFY_APP_EXITED passes the MemHandle of the application shutting down.

The kernel sends MSG_NOTIFY_USER_DICT_CHANGE whenever the system changes the current user dictionary in use. All objects signed up for the GCN list GCNSLT_USER_DICT_CHANGE will receive this message after the change

occurs. `MSG_NOTIFY_USER_DICT_CHANGED` passes the `MemHandle` of the Spell Box causing the change and the `MemHandle` of the user dictionary being changed, both of which you may access in your message handler.

9.3.3 Removal from a System List

9.4

You should use `GCNListRemove()` to remove an object from a system GCN list. You must pass the notification ID (`GCNStandardListType` and Manufacturer ID) and the `optr` of the object to be removed. The `optr` of the object in question will only be removed from the list of the particular change specified. If the `optr` is on several GCN lists, those other lists will remain unchanged.

An object or process in the course of dying must remove itself from all GCN lists that it is currently on. You should therefore keep track of all GCN lists you add a particular object to. It is usually convenient for a process to remove itself from these lists within its `MSG_GEN_PROCESS_CLOSE_APPLICATION` message handler or for an object to remove itself in its `MSG_META_DETACH` handler.

Code Display 9-2 Removing a Process from a GCN list

```
@method MyProcessClass, MSG_GEN_PROCESS_CLOSE_APPLICATION {
    optr          myThread;
    myThread = ConstructOptr(GeodeGetProcessHandle(), NullChunk);
    GCNListRemove(myThread, MANUFACTURER_ID_GEOWORKS, GCNSLT_FILE_CHANGE);
    @callsuper;
}
```

9.4 Application Local GCN Lists

The GCN mechanism not only allows you to keep track of system changes but also allows you to keep track of changes within a specific application. These application-specific GCN lists operate in slightly different manners than the system-wide application lists. There are an extensive number of pre-defined

application lists for MANUFACTURER_ID_GEOWORKS. You may use these if you like, but in most cases you will want to create your own list and notification types for your application.

The GenControl objects make extensive use of these types of GCN lists when implementing changes. For a complete discussion of using these lists within the context of a GenControl, see “Generic UI Controllers,” Chapter 12 of the Object Reference Book.

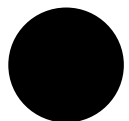
If you will be creating custom GenControl objects or just wish to set up a notification system within your application, you will want to create your own application GCN lists when using this mechanism. To do this, you must follow a few preliminary steps:

9.4

- ◆ Create a new list of type ***YourCompanyNameGenAppGCNListTypes*** within an appropriate ***yourCompanyName.h*** file.
- ◆ Create an GCN notification type of ***YourCompanyNameNotificationTypes*** for the above list type within the ***yourCompanyName.h*** file.

The GCN mechanism in this case performs its functions through a common series of steps. These steps are similar to the steps needed for system-wide notification:

- 1 The object registers for notification with MSG_META_GCN_LIST_ADD.
- 2 The change occurs within your application and invokes your own custom method. Because the change occurs within your application, you are responsible for detecting the change and sending out notification yourself.
- 3 Record the notification event with MSG_META_NOTIFY or MSG_META_NOTIFY_WITH_DATA_BLOCK, the notification list type to use, and the data block to pass (if applicable).
- 4 Use MSG_GEN_PROCESS_SEND_TO_APP_GCN_LIST to pass the event. You may have to pass some **GCNListSendFlags** with this message. This message acts as a dispatch routine, sending all interested parties the recorded event MSG_META_NOTIFY.
- 5 The object is informed of the change with MSG_META_NOTIFY or MSG_META_NOTIFY_WITH_DATA_BLOCK. If you need to perform some work related to this change, you should have a message handler to intercept these messages.



9.4.1 Creating Types and Lists

It is a relatively simple matter to create your own notification types. Within an appropriate company-specific file merely create your own types and lists. (For example, all Geoworks application-local lists are within the file **geoworks.h**.)

9.4 Code Display 9-3 Creating New Notification Types and Lists

```
/* These types should be placed within an appropriate yourCompnayName.h file. */
/* First create a group of Notification types to use for your MANUFACTURER_ID. */
typedef enum {
    <yourCompanyName>_NT_CUSTOM_NOTIFICATION_NUMBER_ONE
    <yourCompanyName>_NT_CUSTOM_NOTIFICATION_NUMBER_TWO
    ...
} <yourCompanyName>NotificationTypes;

/* Then create whatever Notification list types you need. These list types
 * usually correspond one-to-one to the types enumerated above. It is possible,
 * however, for several lists to be interested in a single notification type. */
typedef enum {
    <yourCompanyName>_GAGCNLT_CUSTOM_LIST_TYPE_ONE
    <yourCompanyName>_GAGCNLT_CUSTOM_LIST_TYPE_TWO
    ...
} <yourCompanyName>GenAppGCNListTypes;
```

9.4.2 Registering for Notification

MSG_META_GCN_LIST_ADD

Registering for application notification is simple once you have created your own custom notification lists. Whenever an object or process needs to be notified of an application change, you should call MSG_META_GCN_LIST_ADD to add that object or process to the list interested in that particular change. MSG_META_GCN_LIST_ADD finds the appropriate custom GCN list and adds the optr of the new object to the end of that list. (If no space for the list currently exists because it is empty, the message will allocate space for the

list automatically.) You may add the interested optr at any time during the process' or object's life, but it is usually convenient for a process to be added in its MSG_GEN_PROCESS_OPEN_APPLICATION or for an object to be added in its MSG_META_ATTACH handler.

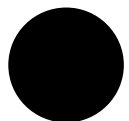
Each optr in a GCN list should have a notification ID attached to it. The combination of a manufacturer ID and a notification type comprises an element's specific notification ID. MSG_META_GCN_LIST_ADD must pass the optr of the object to add, along with a notification ID. For each separate notification ID, a separate GCN list is needed and will be created automatically.

9.4

Geoworks has several pre-defined GCN lists of type **GeoWorksGenAppGCNListType** for use by applications. You will probably have only limited use for these; these list types are used mostly by the UI controllers. For information on these types and how the various classes use them, see "Generic UI Controllers," Chapter 12 of the Object Reference Book.

Code Display 9-4 Adding Yourself to a Custom GCN List

```
@method MyProcessClass, MSG_GEN_PROCESS_OPEN_APPLICATION {  
    @callsuper;          /* Do default MSG_GEN_PROCESS_OPEN_APPLICATION */  
    myThread = ConstructOptr(GeodeGetProcessHandle(), NullChunk);  
    /* myThread (the process) is added to notification of TYPE_ONE changes */  
    @call MyApplication::MSG_META_GCN_LIST_ADD(myThread,  
        yourCompanyName_GAGCNLT_CUSTOM_LIST_TYPE_ONE,  
        MANUFACTURER_ID_yourCompanyName);  
}
```



9.4.3 Handling Application Notification

MSG_META_NOTIFY, MSG_META_NOTIFY_WITH_DATA_BLOCK,
MSG_META_GCN_LIST_SEND

9.4

When a change occurs in the application that needs to send out notification, you must set up the notification message and send it to the interested list. You may attach a data block to this notification for use by the objects on the notification list. To send out these notifications, you should use MSG_META_NOTIFY or MSG_META_NOTIFY_WITH_DATA_BLOCK (when passing data).

In the simplest case without the need to pass data, you should encapsulate MSG_META_NOTIFY with the particular Notification ID (notification type and Manufacturer ID) that should be notified. You should then send MSG_GEN_PROCESS_SEND_TO_APP_GCN_LIST to your application object with this event and the particular GCN list interested in this change. (Note that you will have to keep track of which lists are interested in which notification types.) Make sure that you perform a send (not a call) when using this message as the message may cross threads.

Code Display 9-5 Using MSG_META_NOTIFY

```
@method MyProcessClass, MSG_SEND_CUSTOM_NOTIFICATION {  
  
    MessageHandle event;  
  
    /* First encapsulate the MSG_META_NOTIFY with the type of list and manufacturer ID  
    * interested in the change. Since this message is being recorded for no class in  
    * particular, use NullClass.*/  
  
    event = @record (optr) NullClass::MSG_META_NOTIFY(  
        MANUFACTURER_ID_yourCompanyName,  
        yourCompanyName_NT_CUSTOM_TYPE_ONE);  
  
    /* Then send this MSG_META_NOTIFY using MSG_META_GCN_LIST_SEND. You must make sure  
    * to pass the particular GCN list interested in the changes encapsulated in the  
    * above message. */  
  
    @send MyProcess::MSG_GEN_PROCESS_SEND_TO_APP_GCN_LIST (  
        (word) 0,                /* GCNListSendFlags */  
        event,                   /* Handle to MSG_NOTIFY event above. */  
        0,                       /* No data passed, so no data block. */  
        /* Pass the list interested in NT_CUSTOM_TYPE_ONE notification types. */  
    );  
}
```

```

yourCompanyName_GAGCNLT_APP_CUSTOM_LIST_ONE,
/* Pass your manufacturer ID. */
MANUFACTURER_ID_yourCompanyName);
}

```

If instead you need to pass a data block along with the notification, you should use `MSG_META_NOTIFY_WITH_DATA_BLOCK`. You should set up the structure to pass beforehand. You must also make sure to add a reference count to the data block equal to the number of *lists* (not objects) you wish to send the notification. To do this, call **MemInitRefCount()** with the data block and the total number of lists you are sending the notification to. (In most cases, you will only send notification to one list, although, of course, that list may have several objects.)

9.4

Code Display 9-6 MSG_META_NOTIFY_WITH_DATA_BLOCK

```

@method MyProcessClass, MSG_SEND_CUSTOM_NOTIFICATION {

    typedef struct {
        int number;
        char letterToLookFor;
    } MyDataStructure;

    MemHandle myDataBlock;
    MyDataStructure *myDataPtr;
    MessageHandle event;

/* Allocate and lock down a block for the data structure. This will be passed
 * along with the notification. NOTE: data blocks must be sharable! */

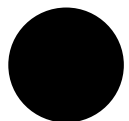
    myDataBlock = MemAlloc(sizeof(MyDataStructure), (HF_DYNAMIC | HF_SHARABLE),
                           HAF_STANDARD);

    myDataPtr = MemLock(myDataBlock);

/* Load up the structure with pertinent information. */
    myDataPtr->count = 200;
    myDataPtr->letterToLookFor = 'z';

/* Unlock it and set its reference count to 1 as we are only sending this to one
 * list. */
    MemUnlock(myDataBlock);
    MemInitRefCount(myDataBlock, (word) 1);
}

```



General Change Notification

370

```
/* Now encapsulate a MSG_META_NOTIFY_WITH_DATA_BLOCK message. Since it is being
 * recorded for no particular class, use NullClass as its class type. */

    event = @record (optr) NullClass::MSG_META_NOTIFY_WITH_DATA_BLOCK(
        MANUFACTURER_ID_yourCompanyName, /* Manufacturer ID */
        NT_CUSTOM_TYPE_ONE,              /* List type. */
        myDataBlock);                    /* handle of data block */

/* Finally, send the message off to our process. The GCNListSendFlags depend on
 * the situation. */

9.4    @send MyProcess::MSG_GEN_PROCESS_SEND_TO_APP_GCN_LIST(
        (word) 0,                        /* GCNListSendFlags */
        event,                          /* Handle to message */
        myDataBlock,                    /* Handle of data block */
        /* Pass the type of list interested in NT_CUSTOM_TYPE_ONE notification. */
        GAGCNLT_APP_CUSTOM_LIST_ONE,
        MANUFACTURER_ID_yourCompanyName);

/* All done! myDataBlock will be MemFree()'d automatically. */
}
```



Do not free the global heap block (if any) yourself. It will be freed by the system.

The object or process originally requesting notification of the change will want to provide a handler for the MSG_META_NOTIFY or MSG_META_NOTIFY_WITH_DATA_BLOCK. If additional data about the change is passed in a data block, the process should access that information with **MemLock()** and **MemUnlock()**. You should always call the process's superclass in your message handler, to make sure that the global heap block will be automatically freed by **MetaClass**. Therefore, do not free a notification data block manually in a notification handler.

Code Display 9-7 Intercepting an Application Notification Change

```
/* Code to implement when MyObjectClass receives MSG_META_NOTIFY with a certain
 * notification type. */

@method MyObjectClass, MSG_META_NOTIFY {

    MyDataStructure myData;            /* Stores the passed data block. */

/* Lock the data structure. */

    myData = MemLock(data);
```

```

/* Check the notification type and implement the changes you wish to occur in
 * response to the previous event. */

    if ((notificationType == yourCompanyName_NT_CUSTOM_TYPE_ONE) &
        (manufID == MANUFACTURER_ID_yourCompanyName)){
        /* Code to implement for your object. */
    }

    MemUnlock(data);

    @callsuper;                                /* Important! Frees data block. */
}

```

9.4

9.4.4 Removal from Application Lists



Never let a process or object die without removing its optr from all current GCN lists.

You should use `MSG_META_GCN_LIST_REMOVE` to remove an object from an application GCN list. You must pass the routine the notification ID (***yourCompanyNameAppGCNListTypes*** and Manufacturer ID) and the optr of the object to remove. Note that the optr of the object in question will only be removed from the list of the particular change specified. If the optr is on several GCN lists, those other lists will remain unchanged.

An object or process in the course of dying must remove itself from all GCN lists that it is currently on, both from the system and from an application. You should therefore keep track of all GCN lists you add a particular object to. It is usually convenient for a process to remove itself from these lists within its `MSG_GEN_PROCESS_CLOSE_APPLICATION` message handler or for an object to remove itself at `MSG_META_DETACH` time.

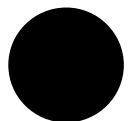
Code Display 9-8 Removing from an Application GCN List

```

@method MyProcessClass, MSG_GEN_PROCESS_CLOSE_APPLICATION {

    @send MyApplication::MSG_META_GCN_LIST_REMOVE(
        MyObject,                                /* optr to remove from list. */
        yourCompanyName_NT_CUSTOM_LIST_ONE,
                                                /* list to remove object from. */
        /* Manufacturer ID of list to remove object from. */
        MANUFACTURER_ID_yourCompanyName);
}

```



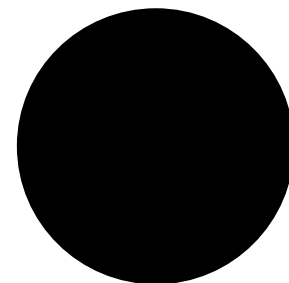
General Change Notification

372

```
    @callsuper;  
}
```

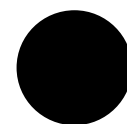
9.4

The GEOS User Interface

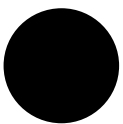


10

10.1	The GUI	375
10.2	The GEOS User Interface	376
10.3	Using the Generic Classes	378
10.3.1	The Generic Class Tree	380
10.3.1.1	GenClass	381
10.3.1.2	GenApplicationClass	381
10.3.1.3	GenPrimaryClass	381
10.3.1.4	GenTriggerClass	382
10.3.1.5	GenInteractionClass	382
10.3.1.6	GenViewClass	382
10.3.1.7	List Classes	382
10.3.1.8	GenValueClass	383
10.3.1.9	GenTextClass	383
10.3.1.10	Document and Document Control Classes	383
10.3.1.11	Display and Display Control Classes	384
10.3.1.12	GenControlClass and UI Controllers	384
10.3.1.13	GenToolControlClass	384
10.3.1.14	GenFileSelectorClass	385
10.3.1.15	GenGlyphClass	385
10.3.1.16	GenContentClass	385
10.3.2	Creating a Generic Object Tree	385
10.4	Using the Visible Classes	385
10.4.1	Visible Objects and the GenView	386
10.4.2	The Visible Object Document	386
10.4.3	Visible Object Abilities	387
10.4.4	The Vis Class Tree	388
10.4.5	Creating a Visible Object Tree	389
10.4.6	Working with Visible Object Trees	390
10.4.6.1	Sending Messages in the Tree	390
10.4.6.2	Altering the Tree	391



10.5	A UI Example.....	391
10.5.1	What TicTac Illustrates.....	391
10.5.2	What TicTac Does	392
10.5.3	The Structure of TicTac	392
10.5.3.1	TicTac's Generic Tree	393
10.5.3.2	TicTac's Visible Tree.....	397
10.5.4	TicTacBoard Specifics.....	400
10.5.5	TicTacPiece Specifics	404



The GEOS user interface (UI) is an integral part of the system software and does an amazing amount of work for applications. The UI is a sophisticated and powerful object library that includes everything from simple buttons to self-scrolling windows to plug-in objects that provide nearly complete word processing.

GEOS also uses a new concept in user interface technology: the generic UI. By using the generic UI, GEOS programs simply declare their UI needs and leave nearly all drawing, screen management, and input management up to GEOS. The generic UI technology also allows a single application executable file to run with several different look-and-feel specifications. 10.1

This chapter will provide an overview of the specifics of the GEOS User Interface. For a general overview of the UI and a high-level description of its components, see “System Architecture,” Chapter 3.

10.1 The GUI

Ever since the first text-based computer application was created, computer users have looked for easier, more intuitive interfaces to their programs. If the user has to struggle to remember commands or to find the right function, the program is of very little use to him. Thus, the user interface of a program, the program’s look and feel, can often be as important as its feature set.

The graphical user interface (GUI) concept has prompted millions of people to begin using computers and has made computer use for millions of others easier. Many corporations and school systems even make GUI a requirement for all their computer acquisitions.

GUIs, however, are not without problems. With so many hundreds of different programs available, even well-designed interfaces can differ significantly from other well-defined interfaces. Although users no longer have to remember whether the “print” command is Ctrl-Alt-P or F4, they may have to search through dozens of menus just to find it. To solve this consistency problem, different manufacturers have created GUI specifications that detail

menu structure, color schemes, input management, and many other aspects of the GUI.

GEOS takes this one step further with its Generic User Interface. Application programmers do not have to decide beforehand which GUI specification will be run by their program; nor do they have to write one version for each GUI supported. Instead, they define their generic UI needs and then let GEOS determine the manifestation of those needs at run-time.

10.2

Another problem with GUIs is their speed. Without powerful hardware, many GUI-based systems are sluggish because of high overhead associated with geometry management and drawing. GEOS, however, is extremely fast because it is entirely object-oriented and is programmed in assembly language. And, since most of the user interface is managed and drawn by GEOS, applications written for the system automatically take advantage of this speed.

10.2 The GEOS User Interface

Most programmers who write applications for GEOS will have worked with other GUI systems. Some will even have worked with object-oriented systems. Using the GEOS UI, however, will be a new experience for nearly every programmer.

The GEOS user interface consists of a number of dynamic libraries containing object classes. Applications include objects of these classes to gain various types of functionality. In general, applications will use two different basic types of objects for their UI: Generic objects provide basic UI functionality including windows, menus, and dialog boxes, and their implementation on the screen is determined by GEOS. Visible objects provide more application-specific functionality and offer complete display control to the application.

Generic object classes are all based on the class **GenClass**. Applications will never use any objects of **GenClass**, but they will use many objects of its subclasses. A list of these subclasses is shown in Figure 10-1, and each of these classes is described in “Using the Generic Classes” on page 378.

Visible object classes are subclassed from **VisClass**. **VisClass** may be used often by applications, as will its subclasses. The subclasses of **VisClass** are shown in Figure 10-2, and each is described in “Using the Visible Classes” on page 385.

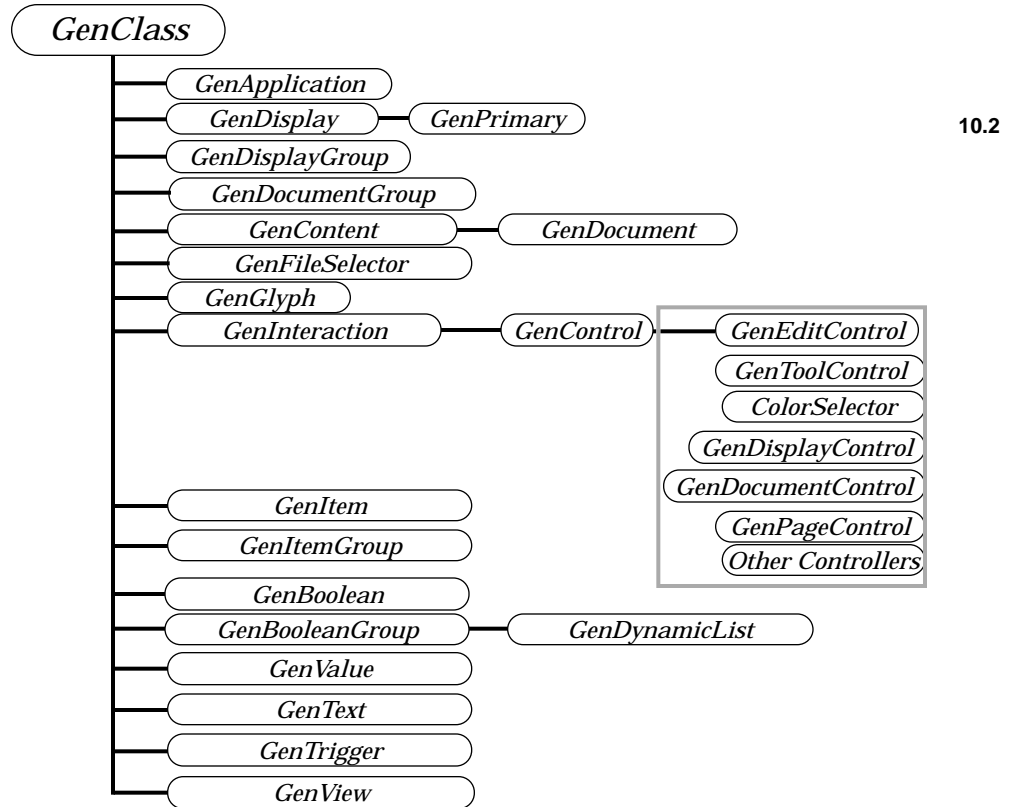


Figure 10-1 *Generic Class Hierarchy*

All the classes shown have the properties of GenClass and can be used in an application's generic UI object tree.

When to use the generic or visible objects is subject to many criteria. In order to learn more about each of these components of the UI, you should read the two sections following this one.

The GEOS UI also contains many features that will often be used by applications but which don't fall into either the generic or the visible world.



These features are called “mechanisms” and include input management, geometry management, general change notification, and the clipboard and quick-transfer functions. As these mechanisms are not UI-specific but are only implemented within the UI libraries, they are documented elsewhere in this book.

10.3

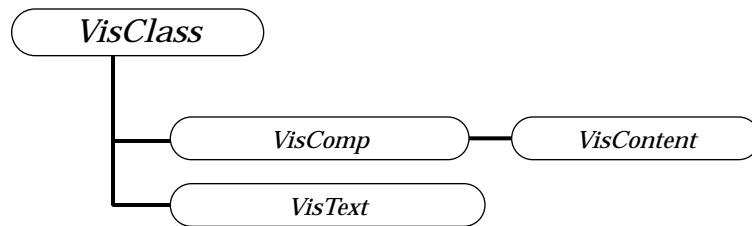


Figure 10-2 *Visible Class Hierarchy*

All the classes shown have the properties of VisClass and can be used in an application's visible object tree. Visible objects are more flexible but require more programming than do generic objects.

10.3 Using the Generic Classes

Most of your application's UI needs will be satisfied by the use of generic UI objects. Many applications may need only the generic classes. Generic UI objects are powerful and easy to use, and they provide a number of services normally left up to application code.

Generic object classes have no inherent visual representation. Rather, each generic object represents a certain set of UI functions instead of UI components. For example, the *GenInteraction* class implements grouping and organizational functions; it can appear in several forms including menus and dialog boxes. It can also have no visual representation but merely provide geometry management for other generic objects.

What visual form a generic object takes, if any, is determined by two factors: the specific UI library in use, and the instance data of the generic object.

- ◆ A specific UI library is essentially a “UI driver” that translates the generic class into its specific representation. Just as each video card has

its own driver, each UI specification (e.g. OSF/Motif or Presentation Manager) has its own specific UI library. This library uses the generic object's instance data to determine the exact form of that object's output and input. A GenValue object, for example, may be implemented as a spinner object in one specific UI but as a slider object in another.

The generic object's instance data determines the features of the object as implemented by the specific UI. Across different GUI specifications, each similar object may appear different or handle input differently. For example, a button in one GUI may invert itself when pressed while the same button in another GUI might simply darken its outline when pressed. However, both buttons will do the same task. Therefore, a generic object's instance data must be categorized into two types: *Attributes* are instance data that will cause the same results no matter what GUI is used (e.g. the function the button performs). *Hints* are instance data that suggest particular implementations that may or may not be implemented by the GUI in use. 10.3

The way the generic object gets translated by GEOS into a visible representation is a complicated process. All generic classes are subclasses of **GenClass**, which is defined as a variant class. This means that **GenClass** has no defined superclass, that the superclass can be changed from time to time. The superclass is determined when the object is instantiated and resolved—when its visual representation is called for. (See “GEOS Programming,” Chapter 5.)

The specific UI library contains objects subclassed off **VisClass**. These objects know how to handle input events and have very specific representation on the screen. These specific UI classes connect to the generic classes through the master/variant mechanism; in this way, the superclass of a generic object is assigned to one of the classes in the specific UI library. The generic object thus inherits the specific class' visible representation.

The above process happens entirely at run-time. The application does not have to have any knowledge of which particular GUI is in use; the same application executable code will work with any specific UI library. Because of the generic UI of GEOS, a user can run the same application under several look-and-feel specifications; this is desirable to the programmer because he only has to code the application once to receive the benefits of several different GUIs.

10.3.1 The Generic Class Tree

Generic objects have a tremendous amount of built-in functionality. Much of this is built into **GenClass**, the topmost class in the generic class tree. For full details on **GenClass** and the other generic classes, see the Object Reference Book.

Among the features offered by all generic classes are

10.3

- ◆ **Visual Representation**
Through the specific UI library, every generic object provides its own visual representation. The application does not have to do any gadget drawing.
- ◆ **Monikers**
Every generic object can have a moniker, which is a name or graphic that gets displayed along with the object.
- ◆ **Input Management**
Through the specific UI library, every generic object properly handles mouse and keyboard input.
- ◆ **Messaging**
As with all objects in the system, generic objects can receive and send messages. In addition, generic objects can pass messages up or down the generic object tree automatically.
- ◆ **Enabled and Usable States**
Generic objects understand their usable and enabled states. If an object is not enabled, it may be visible but the user cannot invoke its action. If an object is not usable, it will not be visible on the screen (in most specific UIs). Entire object trees can be set usable or enabled with one command.
- ◆ **State Saving**
Generic objects automatically save their state when the system shuts down. Therefore, when the system comes back up, dialog boxes and windows will automatically be restored to the same state they were left in when shut down.
- ◆ **Object Tree Management**
All generic objects must be part of a generic object tree to be displayed. Generic objects inherently understand the tree structures and functions. UI gadgetry can dynamically be added, moved, or removed.

Each of the different generic classes is described in overview depth below. Note that every one of these classes may be subclassed to add, change, or remove functionality. Changing or removing functions from a generic class is not encouraged, however, as it can cause a specific UI library to give unpredictable results. For a diagram of all the generic classes in their class hierarchy, see Figure 10-1 on page ● 377.

10.3.1.1 GenClass

10.3

GenClass provides the functionality basic to all generic objects. **GenClass** is not used directly by any applications and has no visible representation. Rather, all generic classes are subclassed off **GenClass**. **GenClass** provides instance fields common to all of its subclasses. Instance fields of special interest include

- ◆ links between parents and children, providing the means of constructing a generic tree.
- ◆ text or graphics strings to serve as an object's visual moniker.
- ◆ a keyboard accelerator to activate an object through keyboard events.
- ◆ a state field relating to the usable state of an object.
- ◆ an attributes field relating to other default behavior, such as how the object handles busy states when an application is waiting for a routine to finish.

GenClass also implements scores of hints that can affect UI geometry, visual representation, data structures, and functions.

10.3.1.2 GenApplicationClass

GenApplicationClass provides the basic functionality to open and close applications within GEOS. An object of this class serves as the top object in any application for GEOS.

10.3.1.3 GenPrimaryClass

GenPrimaryClass is a subclass of **GenDisplayClass**. The GenPrimary is the chief UI grouping object of an application, and it usually appears as the

application's primary window. An application's GenPrimary object manages all controls and output areas that are invoked when an application is first launched. You will usually create a GenPrimary as the sole child of your GenApplication object.

10.3.1.4 GenTriggerClass

10.3

A GenTrigger is a simple pushbutton that executes an action when activated by the user. Typically, the trigger will have a moniker displayed within it and will be activated by a mouse click or by a special keystroke sequence. GenTriggers are very common in applications.

10.3.1.5 GenInteractionClass

GenInteraction objects are essentially grouping mechanisms. GenInteractions are the key objects for creating both menus and dialog boxes, and they can be used to organize the geometry of other generic objects. Typically, a GenInteraction will have a number of children, each of which will appear within the interaction on the screen. The Interaction itself may or may not have a visible representation.

10.3.1.6 GenViewClass

The GenView object provides a scrollable window in which the application has complete drawing control. Most applications will use a GenView, and many will use it in conjunction with a VisContent object. The View is extremely powerful, providing all clipping, scrolling, resizing, and scaling automatically. A View can even be splittable or linked to other views. The GenView can display either normal graphic documents or hierarchies of visible objects.

10.3.1.7 List Classes

Together, GenBoolean, GenBooleanGroup, GenItem, GenItemGroup, and GenDynamicList provide many different types of lists. List objects may be used to create lists that are dynamic or static; scrollable or not; exclusive,

non-exclusive, or otherwise. List objects may appear within menus or dialog boxes as well as within an application's primary window.

10.3.1.8 GenValueClass

The **GenValue** object allows the user to set a value within a particular range. This may be implemented as a slider, a spinner, or a pair of up/down buttons next to the value. Ranges may use scalar or distance values and can have their maximum and minimum values set by the application.

10.3

10.3.1.9 GenTextClass

GenTextClass is tremendously versatile and can be used for text displays or text-edit fields. The **GenText** is used by nearly every application that either displays text or requires text input. It is so versatile and powerful that it can provide the power of an entire word processor without any additional code in the application.

The **GenText** object supports selection and control of fonts, point sizes, text color, paragraph color, paragraph borders, margin settings, tab stops, manual leading, and character kerning, as well as several other features. The text library also provides several controllers that work with the **GenText** to allow the user to set all these features.

10.3.1.10 Document and Document Control Classes

Together, **GenDocumentClass**, **GenDocumentGroupClass**, and **GenDocumentControlClass** provide all the functions necessary to create, save, open, and edit document files. These classes provide not only the UI menus and tools (the File menu) but also the functions for managing the document files. Applications that use these classes never have to call routines to open, close, or save files—all that is done automatically, including the file selector mechanisms to aid the open and save-as functions.

GenDocument objects are created and managed automatically by the **GenDocumentGroup**. Each document object represents a single file which has been opened or newly created by the user.

10.3.1.11 Display and Display Control Classes

Together, the **GenDisplay**, **GenDisplayGroup**, and **GenDisplayControl** provide display windows and the UI gadgetry to manage them. Typically, these objects will be used in conjunction with the document and document control objects to provide one display for each document.

10.3

The **GenDisplayGroup** object creates and manages multiple **GenDisplay** document windows. The **GenDisplayControl** object creates and maintains a Window menu to allow the user to operate on the individual displays. If your application will have multiple documents or multiple displays open, you will want to use these objects.

10.3.1.12 GenControlClass and UI Controllers

GenControlClass is used to create UI controller objects. Applications will most likely not use **GenControlClass** directly, though some object libraries might. For example, the Text Library uses controller objects for font control, point size control, and style control, among other things. Any application that uses the Text Library can include the font controller object; the user will then be able to select and apply fonts without the application having to do any work to support it.

The UI and various libraries provide many controllers you can use immediately. Some examples are the **GenEditControlClass**, which creates and maintains the Edit menu and tools; **ColorSelectorClass**, which creates and displays UI gadgetry to set color information; and **GenViewControlClass**, which creates and maintains a View menu allowing the user to set scaling and scrolling behavior.

10.3.1.13 GenToolControlClass

GenToolControlClass lets the user select which of an application's tools are available and where they should be placed (in a toolbox, in a menu, etc.). Tools are provided by UI controllers. Typically, an application that uses controllers will provide several tool areas and a **GenToolControl**; the **GenToolControl** will automatically create all the UI gadgetry to let the user select which tools are active and where they will appear.

10.3.1.14 **GenFileSelectorClass**

The **GenFileSelector** provides user interface to allow the user to navigate through his or her file system. It is used most often by the document control objects and is used directly by only some applications.

10.3.1.15 **GenGlyphClass**

GenGlyph displays simple text or graphics strings. Text displayed by a **Glyph** object is not selectable or editable; these objects are typically used for labeling areas or items on the screen.

10.4

10.3.1.16 **GenContentClass**

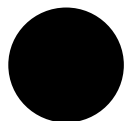
GenContentClass is used with a **GenView** to display other generic UI objects within a scrollable window. The **GenContent** is rarely used because having some of an application's UI objects not visible can confuse some users. More often, a **GenDocument** is used as the view's content; **GenDocumentClass** is subclassed from the **GenContent**.

10.3.2 **Creating a Generic Object Tree**

You don't have to understand all the generic object classes to create a complete generic object tree for your application. For insight into and an example of creating a generic tree (including the primary window, a menu, a dialog box, and a scrolling view window), see "First Steps: Hello World," Chapter 4.

10.4 **Using the Visible Classes**

The visible classes in GEOS provide custom objects that can be used for any number of purposes. There are many visible object classes and they are so versatile that everything from spreadsheets to drawing programs to interactive games can be created from them.



10.4.1 Visible Objects and the GenView

Visible objects are designed for flexibility and for interacting with the user. Typically, visible objects will reside in an object tree, the root of which is a VisContent or GenDocument object connected to a GenView (this section assumes the VisContent, though the GenDocument is roughly equivalent for the purposes of overview). The relationship between these objects is simple, yet it accomplishes an immense amount of work for the application.

10.4

The GenView provides a window that may be scrollable, scalable, and resizable. The view is directly connected to the VisContent object and works very closely with it to handle input and drawing events. The VisContent acts as a manager for the visible object tree; it passes input events (mouse moves and mouse clicks) on to the proper visible object directly under the mouse pointer, and it makes sure all visible objects in the tree draw themselves at the appropriate time.

The view and VisContent may be connected tightly or very loosely for sizing purposes. For example, the view may be forced to resize itself to the proper size of the VisContent; or, the view might be scrollable and resizable independent of the content's size. The VisContent can choose whether mouse events are expected or required, and the view will notify the VisContent whenever the window has been invalidated and the document needs redrawing.

10.4.2 The Visible Object Document

Visible objects exist in an object tree and draw themselves in the GEOS graphic space. Every visible object knows where in the graphic coordinate space it sits and how big it is. The top object of the object tree, the VisContent, manages the entire tree. Composite objects (of class **VisCompClass**) can be used as organizational objects that control other objects.

Using the VisContent, the VisComposite, and the standard Vis objects, you can create just about any interactive document you want. (There are other available object libraries such as the bitmap and the ruler libraries, and they are based on these three building blocks.)

Every visible object knows what its bounds are; that is, each object knows exactly how big it is and where it sits in the graphics document. The bounds are always rectangular in the base **VisClass** format, though a subclass of **VisClass** could easily be created to handle more complex shapes. Bounds are in the standard graphics coordinates (i.e. 16-bit numbers based on a 72 dpi grid). Vis objects may also exist in large documents (32-bit coordinates), but the objects must handle the majority of the extra coordinate manipulation on their own.

10.4

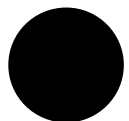
Visible objects do not inherently know how to draw themselves. However, the GenView, when resized or scrolled (for example), will send MSG_META_EXPOSED to the VisContent. The VisContent object will then send a MSG_VIS_DRAW message to itself and to all its visible object children, directing them to draw themselves in the proper place. For the visible object to draw itself, it must handle MSG_VIS_DRAW. In its handler, it can call any normal graphics commands to draw anything wherever it wants (not just within its bounds).

For examples of visible objects that know their bounds and that handle MSG_VIS_DRAW, see “A UI Example” on page 391.

10.4.3 Visible Object Abilities

Visible objects can be used in innumerable situations; with a little work from the application developer, they provide dozens of useful features including

- ◆ **Knowledge of Their Bounds**
Every visible object knows exactly where in the application’s document it sits and exactly how big it is. It can easily change its own bounds, either moving itself in the document space or resizing itself.
- ◆ **Ability to Draw Themselves**
Every visible object is responsible for drawing itself. When it receives a drawing message (MSG_VIS_DRAW), it must recognize the context of the message and draw itself appropriately. **VisClass** does not have a default handler for MSG_VIS_DRAW; each subclass must handle this message itself.
- ◆ **Input Handling**
Pointer and click events as well as keyboard input can be automatically transmitted through the view to the VisContent if desired. Several



combinations of event pass-throughs are available. When the VisContent receives a mouse event, it can intercept and handle it or pass it on to whatever object is under the pointer. If the VisContent does not handle the event (such as MSG_META_PTR) explicitly, the message will automatically be passed on to the object under the pointer.

10.4

- ◆ Geometry Management
Both the view and the VisContent can interact to provide specific sizing behavior for the window and content. Additional geometry management is available to help a VisComp composite object manage its children.
- ◆ Object Tree Management
Messages may easily be passed to an object's parent or children. Also, objects may be added to or removed from the visible tree without difficulty.
- ◆ Use of VisMonikers
In addition to simply drawing itself, a visible object can have a visible moniker that it can draw as well. This simplifies drawing text or handling MSG_VIS_DRAW in that the moniker can be set beforehand and simply drawn with a single call.

Using visible objects allows an application a lot more flexibility in display and input handling than is available with the generic objects. It also requires more programming; however much of the most difficult work is made easy by the view and VisContent objects. The application does not have to worry about clipping or scrolling or even determining what type of input event is taking place. All this is somewhat automatic and can, for the most part, be ignored.

10.4.4 The Vis Class Tree

There are four base visible classes on which the other object libraries are founded. **VisClass** is the most basic and at the root of the visible class tree. Under it are VisComp and VisText. VisContent is a subclass of VisComp.

VisTextClass is special in that it is rarely subclassed as it already contains nearly all the functionality an application will need from a visible text object.

The other three classes are typically subclassed by applications. None of the three can draw itself; instead, the subclass must handle MSG_VIS_DRAW, the

message that indicates the object must draw itself. The three different classes are used as follows:

◆ **VisClass**

VisClass is the head of the visible class tree and therefore is the most broad-based in functionality. **VisClass** objects can not have children and therefore can only exist as the leaves of the object tree.

◆ **VisCompClass**

VisCompClass provides a composite **VisClass** object. Essentially, this class is the same as **VisClass** except that it can have children. It also includes several special geometry options that allow it to manage and place its children as well as set their bounds.

10.4

◆ **VisContentClass**

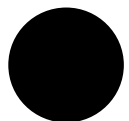
VisContentClass is used only as the content of a GenView. It is a subclass of **VisCompClass** and therefore can have children and manage them with the same geometry features available to VisComp. In addition, the VisContent interacts with the GenView to handle input and drawing. The content can interact with the view to determine sizing behavior as well as input behavior.

10.4.5 Creating a Visible Object Tree

You can create a visible object tree either in your Goc source code or at run-time. As stated earlier, the visible tree is linked to the generic UI object tree through the GenView object. Without a view object, you will not be able to display your visible objects.

The visible object tree must have a **VisContentClass** object (or a subclass of **VisContentClass**) as its root object. If another class is chosen for the topmost object, that class will have to handle all the messages that would normally be received by a content; otherwise, results are unpredictable and your application will likely not function the way you expect.

VisContent objects are rarely, if ever, used lower in the visible tree than as the top node. For levels further down in the tree, you can use **VisCompClass** (or, again, a subclass of **VisCompClass**). VisComp allows the object to have children and does not incur the same amount of overhead associated with a VisContent. Any object in the visible tree that may have children must be a “composite” object, or a subclass of **VisCompClass**.



Leaf nodes of the visible tree can be direct subclasses of **VisClass**. Objects of **VisClass** can not have children, but they have all the attributes and instance data you need for a leaf node. You must subclass **VisClass**, however, if your object is ever to be shown on the screen: Since visible objects can have any application-defined visible representation, you must write the `MSG_VIS_DRAW` handler yourself; this can be done only in a subclass.

10.4 10.4.6 Working with Visible Object Trees

Working with visible object trees is quite easy and provides immense flexibility to your applications. Entire groups of objects can be added to, removed from, or moved around in the display with a single command. New objects can be created during execution, and others can be destroyed at will.

10.4.6.1 Sending Messages in the Tree

Often, an object in the visible tree will need to contact its parent or its children. For example, a child that needs to know the state of its parent must be able to find and contact that parent. Likewise, a parent that must notify all its children to redraw themselves must be able to quickly send out the notification.

An object can send a message to its parent by simply specifying **@visParent** as the message recipient. That same object could also contact all its children by specifying the recipient as **@visChildren**. The message sent (only with **@send** since there are multiple destinations) will go to each of the object's children.

If you need more complicated message passing, you can build this into your **Vis** subclasses. It is not difficult for an object to retrieve the `optr` of any of its children or of its parent; it can then send the message appropriately. Each recipient can handle it in the same manner, forwarding it on to the next layer of the tree.

VisClass also employs several messages known as “visual upward queries,” or “VUP” messages. These travel up the visible tree until they encounter an object of the proper class, where they will be handled.

10.4.6.2 Altering the Tree

VisClass and **VisCompClass** can handle messages that retrieve various information about the visible tree. They also have messages for altering the visible tree's structure.

10.5 A UI Example

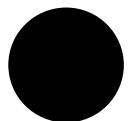
10.5

This section uses the sample application TicTac, a simple Tic Tac Toe board and pieces that can be moved around the board.

10.5.1 What TicTac Illustrates

The TicTac sample application can teach you several things about how the visible world works and about how to manage visible objects in an application. It is simple but illustrates the following concepts:

- ◆ **Drawing Visible Objects**
Every visible object must be able to draw itself. **VisClass** does not have any inherent code to make visible objects visible; instead, subclasses of **VisClass** must handle MSG_VIS_DRAW.
- ◆ **Changing an Object's Position**
The pieces of the TicTac game allow the user to move them around the screen using the mouse. This entails tracking the mouse and then changing the object's position in the document.
- ◆ **Sending Messages Up and Down the Visible Tree**
The pieces must interact with the VisContent object to ensure they stay on the game board, and the VisContent must notify the pieces when the "New Game" button has been pressed. Therefore, messages must be passed both up and down the visible tree.
- ◆ **Handling Input Events**
In order for the user to move the pieces around the board, the pieces must be able to receive mouse events. Several different types of mouse events are handled to show the pieces moving around the board and to handle input properly.



- ◆ **Drawing Background Graphics**
Because the game board does not change at all, its background is drawn by the content object. The content could change its drawing behavior if necessary, but background graphics are most easily drawn by the content.
- ◆ **Visible Tree Structure**
A simple, two-layer visible tree is used in the TicTac application: The VisContent is the root of the tree, and each of the pieces is a leaf.

10.5

10.5.2 What TicTac Does

The TicTac sample application is extremely simple. It draws a Tic Tac Toe board and its outline, and it has ten pieces which the user can move. Five of these pieces are gray squares, and five are gray circles. Each of the pieces may be moved by clicking and dragging it with the mouse. Any piece may be moved anywhere on the board, but pieces may not be moved off the board.

A Game menu with a “New Game” trigger replaces all the pieces to their original locations. Because this is a simple example, no actual rules of any kind are enforced. Recognition of winning sequences and rules involving turn sequencing or playing against the computer are left as exercises for the reader.

10.5.3 The Structure of TicTac

The TicTac sample application is coded in two files: The first, **tictac.gp**, is the geode parameters file. The other, **tictac.goc**, contains all the code for objects in the application. The geode parameters file is similar to other **.gp** files and is not discussed in this section.

The application uses two different object trees, one for its generic UI and one for the game board and game pieces. The first consists of generic UI objects and the second of objects subclassed off **VisClass** and **VisContentClass**. The two object trees are diagrammed in Figure 10-3 and are described below.

10.5.3.1 TicTac's Generic Tree

The TicTac application begins with two standard generic objects common to all applications: GenApplication and GenPrimary. These two objects are described in detail in other sections and are not covered here, though their definitions are shown in Code Display 10-1.

The primary object, TicTacPrimary, has two children. One, TicTacGameMenu, implements the game menu; the other, TicTacView, provides the window through which the user can interact with the visible tree.

10.5

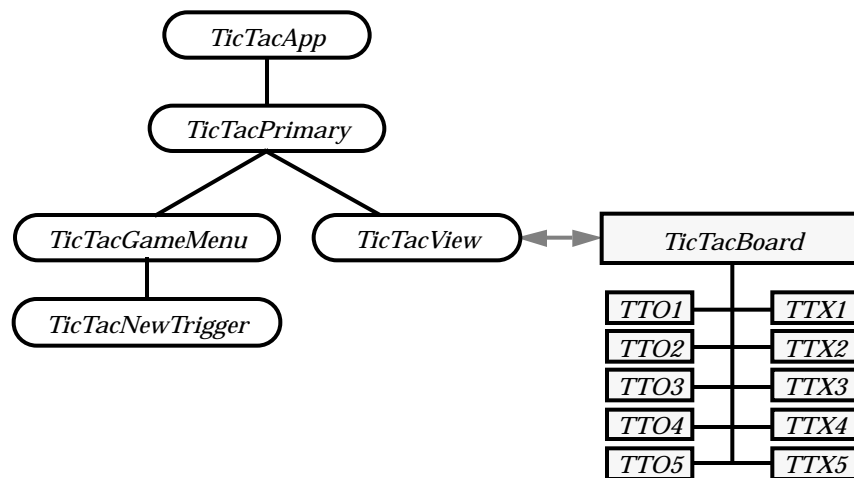


Figure 10-3 The TicTac Object Trees

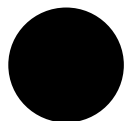
The generic UI object tree has five objects, and the visible object tree has eleven. The two trees communicate via the TicTacView/TicTacBoard link, determined by the fact that TicTacBoard is set as TicTacView's content object.

Code Display 10-1 TicTacApp and TicTacPrimary

```

@start AppResource;
/* The AppResource resource block contains the TicTacApp object only. This
 * object is in its own resource for performance purposes. */

```



The GEOS User Interface

394

```
@object GenApplicationClass TicTacApp = {
    GI_visMoniker = list { TicTacTextMoniker }
    GI_comp = TicTacPrimary;
    gcnList(MANUFACTURER_ID_GEOWORKS, GAGCNLT_WINDOWS) = TicTacPrimary;
}

@visMoniker TicTacTextMoniker = "TicTacToe";

@end AppResource

10.5 @start Interface;

    /* This is the Primary window of the application. It is not minimizable
     * (since no icon is defined for it). It has two children: The View
     * object and the Menu object. */
@object GenPrimaryClass TicTacPrimary = {
    GI_comp = TicTacView, TicTacGameMenu;
    ATTR_GEN_DISPLAY_NOT_MINIMIZABLE;
    HINT_SIZE_WINDOW_AS_DESIRED;
}
```

TicTacGameMenu

This object is a standard GenInteraction object set up as a menu. The code for the entire menu (including the TicTacNewTrigger) is shown in Code Display 10-2 and is heavily commented as to how each attribute is used.

The New Game trigger may be invoked at any time except when a piece is being moved. When a piece is being moved, that piece object has the “mouse grab,” meaning that it will receive all mouse and pointer events until it releases the grab. When an object has the mouse grab, no mouse events may be sent to another object, and therefore the menu object can not be clicked on while the piece has the grab.

When the user presses the New Game trigger, the trigger sends its message, MSG_TICTAC_NEW_GAME, to the TicTacBoard object. The TicTacBoard object is the top object in the visible tree and, upon receipt of this message, resets the game board and notifies all the piece objects of the reset. This process is described below in “TicTacBoard Specifics” on page 400.

Code Display 10-2 The TicTac Game Menu

```

/* The TicTacGameMenu object is the only menu of this application. Its only child
 * and only menu entry is the TicTacNewTrigger object. */

@start Interface;          /* In the same resource block as TicTacPrimary. */

@object GenInteractionClass TicTacGameMenu = {
    GI_visMoniker = "Game";          /* The name of the menu. */
    GI_comp = @TicTacNewTrigger;     /* The only menu item. */
    GII_visibility = GIV_POPUP;      /* This attribute indicates that this
                                     * interaction is a menu rather than
                                     * a dialog. */
}

@object GenTriggerClass TicTacNewTrigger = {
    GI_visMoniker = "New Game";      /* The name of the menu item. */
    GTI_destination = @TicTacBoard;  /* The object to receive the "New Game"
                                     * message: the game board object. */
    GTI_actionMsg = MSG_TICTAC_NEW_GAME; /* The message to be sent when the trigger
                                     * is pressed. */
}

@end Interface              /* End of the Interface resource block */

```

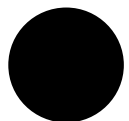
10.5

TicTacView

The TicTacView object is a standard GenView set up to run the TicTacBoard object. The code for TicTacView is shown in Code Display 10-3 and is heavily commented to show what each of the view's attributes is used for.

TicTacView's content object is TicTacBoard, the game board object. This means that any appropriate input events as well as all messages sent out by the view will be passed directly to the TicTacBoard object. The sizing attributes and the fact that the view is not marked GVDA_SCROLLABLE in either dimension makes sure the view sizes exactly to the game board's bounds.

A GenView object is necessary in every case where a visible object tree is used. The view not only displays the visible tree but also handles all clipping, scaling, scrolling, and sizing if any is desired (none is used in TicTac). It also takes and passes on any appropriate mouse or keyboard input events.



Additionally, it interacts directly with its content object (in this case TicTacBoard) to determine proper geometry and sizing behavior of the content and the view.

This view also provides the background color of the game board, dark blue.

Code Display 10-3 The TicTacView Object

```
10.5  /* This object provides the window through which the user interacts with the
      * visible object tree. This object communicates with the game board object (a
      * subclass of VisContentClass) to coordinate drawing, clipping, sizing, and
      * even input handling. */

      @start Interface;          /* In the same resource block as TicTacPrimary. */

      @object GenViewClass TicTacView = {
          GVI_content = @TicTacBoard; /* The content object of this view is the
                                     * TicTacBoard object, the root object of the
                                     * visible object tree. */

          GVI_color = { C_BLUE, 0, 0, 0 }; /* The background color of this view
                                     * should be dark blue. */

          /* The horizontal attributes of this view set it to the same
           * size as the game board, and the view is not scrollable. */

          GVI_horizAttrs = @default      | GVDA_NO_LARGER_THAN_CONTENT
                                     | GVDA_NO_SMALLER_THAN_CONTENT;

          /* The vertical attributes of this view set it to the same size
           * as the game board, and the view is not scrollable. */

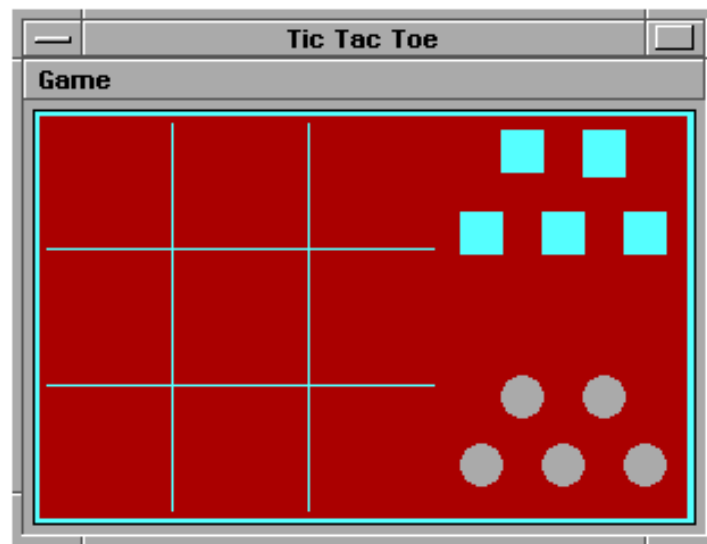
          GVI_vertAttrs = @default |    GVDA_NO_LARGER_THAN_CONTENT
                                     | GVDA_NO_SMALLER_THAN_CONTENT
                                     | GVDA_KEEP_ASPECT_RATIO;

          /* The user won't need to type anything, so there's no need for
           * a floating keyboard. */
          ATTR_GEN_VIEW_DOES_NOT_ACCEPT_TEXT_INPUT;
      }

      @end Interface          /* End of the Interface resource block */
```

10.5.3.2 TicTac's Visible Tree

The visible tree contains eleven objects. One acts as TicTacView's content and is of **TicTacBoardClass**, a subclass of **VisContentClass**. The other ten are all game pieces of class **TicTacPieceClass**, a subclass of **VisClass**. Both the class definitions and the object definitions are given in Code Display 10-4.



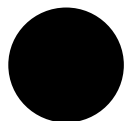
10.5

Figure 10-4 *The TicTac Game Board*

The TicTacBoard object draws the game board and manages the game pieces on the board. The pieces know their starting locations, shown here.

All eleven of the visible objects remain on the screen and in the view during their entire existence. The game board and all its pieces are shown in Figure 10-4; this illustration represents the basic configuration of the game board when the application first starts or when the user presses the “New Game” trigger in the “Game” menu.

The TicTacBoard object draws the border around the board and makes sure the view window sizes itself to the same size as the board. The board is 180 points (2.5 inches) in height and 270 points (3.75 inches) in width; these numbers are stored as the constants `BOARD_HEIGHT` and `BOARD_WIDTH`. TicTacBoard also draws the playing field—this consists of the four white lines on the left side of the game board. TicTacBoard's other main function is



to ensure that all the children (game pieces) behave properly; it makes sure the child's bounds are on the game board when the piece is moved, and it notifies the game pieces when they must draw themselves due to a view exposure. Finally, TicTacBoard receives the "New Game" message from the Game menu; it then redraws the game board and notifies each of the game pieces that they should return to their initial locations.

10.5

Each of the game piece objects knows about its location and status. Each piece knows its initial location, current location, and proposed location (during a move). Every game piece is an instance of **TicTacPieceClass**. This class is shown in Code Display 10-4; it contains a number of instance data fields for these locations. It also has an instance data field indicating what type of piece (i.e. "box" or "ring") the object is.

The "box" objects are designated by having the value TTPT_BOX in their *TTPT_pieceType* fields; the "ring" objects have TTPT_RING in that field. Both types of objects act and react in the same way to various events; the only difference is in their shape and color.

Code Display 10-4 TicTacBoardClass and TicTacPieceClass

```
/* The TicTacPieceTypes enumerated type lists the different types of game pieces a
 * particular piece object can be. In this game, a piece is either a "box" (gray
 * square) or a "ring" (light gray circle). */

typedef ByteEnum TicTacPieceTypes;
#define TTPT_BOX 0
#define TTPT_RING 1

/*****
 * TicTacBoardClass
 * This class is a subclass of VisContentClass and provides the game board
 * for this application. It also manages all the children (piece objects).
 * Because it is a subclass of VisContentClass, it inherits all the instance
 * data fields and messages of that class.
 *****/

@class TicTacBoardClass, VisContentClass; /* this class is a subclass
 * of VisContentClass */
```

```

/* Message definitions for this class */
@message void MSG_TICTAC_NEW_GAME();
    /* This message is sent by the New Game trigger in the Game menu
     * when the user wants to reset the game. It is sent directly to
     * the game board object and causes the board object first to
     * send the "new game" message to each of its children and then
     * to redraw the game board. */

@message Boolean MSG_TICTAC_VALIDATE_BOUNDS(word bottom, word right,
                                           word top, word left);
    /* This message is sent by a game piece that is being moved by the
     * user and is about to be set down. The four parameters are the
     * proposed new bounds of the moved piece; if they are within the
     * game board's limit, this message returns TRUE. If they are at
     * all outside the game board, this message returns FALSE.*/

@endc

/* Declare the class in memory so the method table will be built. */
@classdecl TicTacBoardClass;

/*****
 * TicTacPieceClass
 * This class is a subclass of VisClass and provides all the functions
 * necessary for a game piece in this game. Because it is a subclass of
 * VisClass, it inherits all the instance data fields and messages of
 * that class.
 *****/

@class TicTacPieceClass, VisClass;      /* this class is a subclass
                                         * of VisClass */

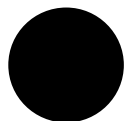
    /* The instance data fields of this class: */
    @instance TicTacPieceTypes TTP_pieceType;
    /* TTP_pieceType defines whether the object of this class is
     * a "box" or a "ring." */

    @instance int TTP_vertPos;
    /* TTP_vertPos indicates the current y position of
     * the piece. This does not indicate the piece's actual
     * bounds but rather where its moving outline appears. */

    @instance int TTP_horizPos;
    /* TTP_horizPos indicates the current x position of
     * the piece. This does not indicate the piece's actual
     * bounds but rather where its moving outline appears. */

```

10.5



```
@instance int TTP_origVertPos;
/* TTP_origVertPos indicates the y position where this
 * piece should return when the New Game trigger is pushed
 * and the piece goes back to its original location. */

@instance int TTP_origHorizPos;
/* TTP_origHorizPos indicates the x position where this
 * piece should return when the New Game trigger is pushed
 * and the piece goes back to its original location. */

10.5 @instance Boolean TTP_dragging;
/* A flag indicating whether the user is in the process of dragging
 * the game piece around the board. */

/* Message definitions unique to this class. */
@message void MSG_PIECE_NEW_GAME();
/* This message notifies the piece object that the user has pushed *
 * the New Game trigger and that the piece should return to its *
 * original position on the board (the TTP_orig(Horiz/Vert)Pos *
 * fields). */

@endc

/* Declare the class in memory so the method table will be built. */
@classdecl TicTacPieceClass;
```

10.5.4 TicTacBoard Specifics

The TicTacBoard object, being of a subclass of **VisContentClass**, handles many messages specific to content objects. However, only three messages are handled specifically by **TicTacBoardClass**. These messages are

MSG_VIS_DRAW

This message notifies the object it must draw itself and any accompanying graphics. TicTacBoard responds by drawing the game board (the border and crossed lines) and by passing the MSG_VIS_DRAW on to all of its children.

MSG_TICTAC_NEW_GAME

This message is sent by the New Game trigger in the Game menu. TicTacBoard responds by sending another new game message, MSG_PIECE_NEW_GAME, to each of its children. Each child will position itself and draw itself properly;

TicTacBoard will then redraw the game board by sending itself a MSG_VIS_DRAW.

MSG_TICTAC_VALIDATE_BOUNDS

This message is sent by a game piece object when it is being moved. TicTacBoard checks the parameters passed and determines whether they are on the game board or not; if they are it returns TRUE, and if they aren't it returns FALSE.

Each of the methods for the above messages is shown in Code Display 10-5. Each is heavily commented and explains the theory behind the method.

10.5

Code Display 10-5 Methods of TicTacBoardClass

```

/*****
 *
 * MESSAGE:      MSG_TICTAC_NEW_GAME for TicTacBoardClass
 *
 * DESCRIPTION:  This method notifies each of the visible children that
 *               a new game is beginning; they should take their places,
 *               and then the board object will redraw itself
 *
 * PARAMETERS:
 *               void ()
 *
 *****/

@interface TicTacBoardClass, MSG_TICTAC_NEW_GAME {
    WindowHandle win;           /* the window to draw to */
    GStateHandle gstate;       /* the gstate of the window */

    /* First notify all the children (game pieces)
     * that a new game is beginning. */

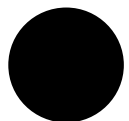
    @send @visChildren::MSG_PIECE_NEW_GAME();

    /* Now initiate a new gstate for the view window.
     * Get the window handle from the view, and then
     * create a new gstate for it. */

    win = @call TicTacView::MSG_GEN_VIEW_GET_WINDOW();
    gstate = GrCreateState(win);

    /* Invalidate the game board rectangle in the document.
     * This will cause the view object to generate a
     * MSG_META_EXPOSED for the rectangle, causing MSG_VIS_DRAW
     * to be sent to this object (TicTacBoard). */

```



The GEOS User Interface

402

```
GrInvalRect(gstate, 0, 0, BOARD_WIDTH, BOARD_HEIGHT);

/* Now destroy the temporary gstate. This is important
 * to keep too many gstate handles from being locked
 * and slowing down the system. */

GrDestroyState(gstate);
}

/*****
 *
10.5 * MESSAGE:      MSG_VIS_DRAW for TicTacBoardClass
 *
 * DESCRIPTION: This method draws the board's outline and the
 *              lines of the playing field. It is sent each time
 *              a portion of the view window becomes invalid (such
 *              as when the primary is moved).
 *
 * PARAMETERS:
 *      void (word drawFlags, GStateHandle gstate)
 *              gstate is the handle of the graphics state associated
 *              with the exposed portion of the view window
 *****/

@method TicTacBoardClass, MSG_VIS_DRAW {
    /* Set up the graphic state properly. The board
     * lines are to be white and three points thick. */

    GrSetLineColor(gstate, CF_INDEX, C_WHITE, 0, 0);
    GrSetLineWidth(gstate, 3);

    /* Now draw the border of the game board. It is a
     * rectangle that outlines the entire board. */

    GrDrawRect(gstate, 0, 0, BOARD_WIDTH, BOARD_HEIGHT);

    /* Set and draw the Tic Tac Toe playing field. The
     * lines are now set to 4 points thickness, and the
     * lines are drawn with HLine and VLine graphics
     * commands. Ideally, preset constants would be used.*/

    GrSetLineWidth(gstate, 4);
    GrDrawHLine(gstate, 5, 60, 175);
    GrDrawHLine(gstate, 5, 120, 175);
    GrDrawVLine(gstate, 60, 5, 175);
    GrDrawVLine(gstate, 120, 5, 175);
}
```

```

    /* When the MSG_VIS_DRAW is received by the game board,
    * it must pass it on to its visible children. It must
    * also pass on the parameters of the message as passed
    * to ensure all drawing is done properly. */
    @send @visChildren::MSG_VIS_DRAW(drawFlags, gstate);
}

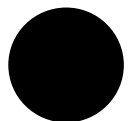
```

10.5

```

/*****
 *
 * MESSAGE:      MSG_TICTAC_VALIDATE_BOUNDS for TicTacBoardClass
 *
 * DESCRIPTION:  This method checks to see if the bounds passed
 *               are on the game board. This is invoked when a game
 *               piece is in motion and receives an END_SELECT message
 *               indicating it is being put down. The piece must
 *               determine whether the suggested bounds are on the
 *               game board; the piece should always query the board
 *               object rather than check directly; if the board were
 *               resizable, the piece could be incorrect sometimes.
 *
 * STRATEGY:     Check the four bounds against the board's edges. If
 *               all four are on the board, return TRUE. If any one
 *               of the four is off the board, return FALSE.
 *
 * PARAMETERS:
 *               void (word bottom, word right, word top, word left)
 *****/
@method TicTacBoardClass, MSG_TICTAC_VALIDATE_BOUNDS {
    if (((bottom < BOARD_HEIGHT) && (top > 0))
        && ((right < BOARD_WIDTH) && (left >= 0))) {
        return(TRUE);
    } else {
        return(FALSE);
    }
}

```



10.5.5 TicTacPiece Specifics

TicTacPieceClass contains most of the game's functionality. Since the user interacts directly with each game piece, the piece must know not only how to draw itself but also how to react to user input.

MSG_PIECE_NEW_GAME

10.5

This is the only message generated by the game itself that a game piece receives; it is sent by the TicTacBoard object when the user has pressed the New Game trigger. The game piece object responds by resetting its bounds to the original settings. It does not have to redraw or invalidate its old bounds because the TicTacBoard object will send a MSG_VIS_DRAW later and will invalidate the entire board.

MSG_VIS_DRAW

This message notifies the object that it must draw itself and any accompanying graphics. The game piece responds by drawing the proper shape in the proper color in the proper place. Since every **VisClass** object inherently knows its location and bounds, the object already knows where and how big the shape should appear. Whether a gray box or circle is drawn depends on the *TTP_pieceType* instance data field.

MSG_META_START_SELECT

This message is sent by the system when the user clicks a mouse button. The UI sends the message to whatever object lies under the pointer. The UI objects then pass the message down the object tree until it gets handled. The progression sends the message to TicTacApp, which passes it to TicTacPrimary, which passes it to TicTacView, which passes it to TicTacBoard, which (by letting the default **VisContentClass** method handle it) passes it to the proper game piece object (if any) under the pointer. The game piece responds by grabbing the mouse and all subsequent pointer events.

MSG_META_DRAG_SELECT

This message, like MSG_META_START_SELECT, indicates that the user has clicked a mouse button and has initiated a drag event. (Normally, this is used to select ranges or groups of objects; in TicTac, however, it is treated like MSG_META_START_SELECT.)

MSG_META_DRAG

This message is sent after the user has clicked the mouse button and is now moving the mouse pointer (and has not released the button yet).

MSG_META_PTR

This message is sent when the pointer image is over the bounds of the game piece whether or not a mouse button has been pressed. (After the object has grabbed the mouse events by handling MSG_META_START_SELECT, the pointer event is sent whenever the mouse pointer is moved.) The piece determines whether or not it is being dragged around the screen. This is known as a “drag event,” and the game piece responds by drawing a piece-shaped outline around the mouse pointer. This outline will follow the pointer around the screen until the user releases the mouse button (causing a MSG_META_END_SELECT, below). The outline is first drawn in either the MSG_META_START_SELECT or MSG_META_DRAG_SELECT handler (whichever is called to start the drag event). MSG_META_PTR and MSG_META_END_SELECT erase the outline before drawing a new one. The game piece will maintain three locations in its instance data: Its *VI_bounds* field maintains its position when selected. Its *TTP_orig(horiz/vert)Pos* fields maintain its original position when the game was first started. Its *TTP_(horiz/vert)Pos* fields maintain the current position of the outline and where the object would relocate to if the move was ended now. If the event is not a drag, the object will not react because it is assumed that no mouse button has been pressed and therefore the user is taking no action. See Figure 10-5.

MSG_META_END_SELECT

This message is sent when the user releases a pressed mouse button, ending the select-and-drag process. The game piece reacts by checking if the location of the pointer is a legal position on the game board. (It does this by sending a verification message to the TicTacBoard object to make sure the proposed new bounds are on the game board.) If the position is legal, the game piece moves itself there, erasing any leftover outlines (from the drag sequence) and its original image on the board. It then causes itself to draw in the new location by sending itself a MSG_VIS_DRAW. If the new location is not legally on the game board, then the object will reset all

its instance data and erase any leftover outlines, causing it to revert to its location before the select-and-drag sequence began.

Each of the methods for the above messages is shown in Code Display 10-6.

Code Display 10-6 Methods for TicTacPieceClass

10.5

```
/* *****  
 *  
 * MESSAGE:      MSG_PIECE_NEW_GAME for TicTacPieceClass  
 *  
 * DESCRIPTION:  This message causes the piece to replace itself  
 *               to its original position. It is invoked when the  
 *               user presses the New Game trigger; the trigger sends  
 *               MSG_TICTAC_NEW_GAME to the TicTacBoard object, and  
 *               the board object sends this message to each of  
 *               the game piece objects.  
 *  
 */
```

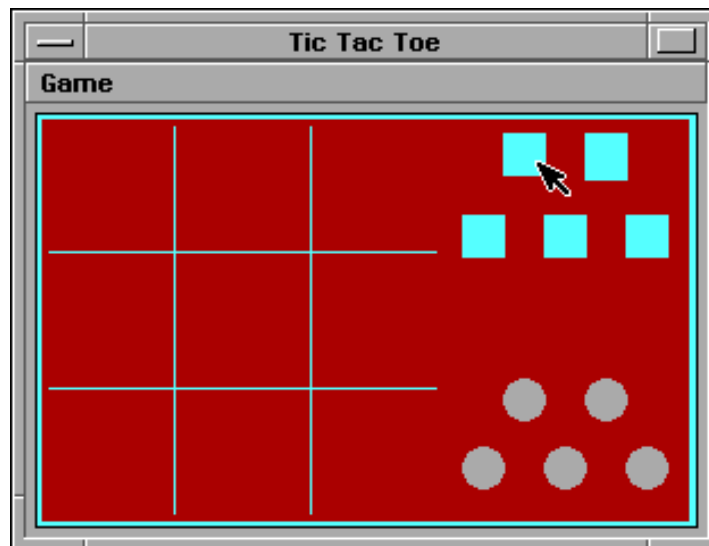


Figure 10-5 *Selection of a Game Piece*

If the user does not have the mouse button pressed, the game piece (TTX1) will ignore the pointer. If the button is pressed now, the game piece will receive a MSG_META_START_SELECT.

```

*
* PARAMETERS:
*     void ()
*****/

@method TicTacPieceClass, MSG_PIECE_NEW_GAME {

    /* Set the current (motion) positions to the original positions. */

    pself->TTP_vertPos = pself->TTP_origVertPos;
    pself->TTP_horizPos = pself->TTP_origHorizPos;

    /* Send a MSG_VIS_BOUNDS_CHANGED to ourselves to make
     * sure the old bounds get redrawn. This message will
     * cause an invalidation of the document where the old
     * (passed) bounds were, causing that portion of the
     * window to be redrawn. */

    @call self::MSG_VIS_BOUNDS_CHANGED(pself->VI_bounds.R_bottom,
                                       pself->VI_bounds.R_right, pself->VI_bounds.R_top,
                                       pself->VI_bounds.R_left);

    /* Set the bounds of the object (VI_bounds) back to
     * their original values. The Rectangle structure
     * contains four fields, each of which must be set.*/

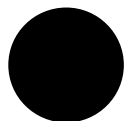
    pself->VI_bounds.R_left = pself->TTP_origHorizPos;
    pself->VI_bounds.R_top = pself->TTP_origVertPos;
    pself->VI_bounds.R_right = (pself->TTP_origHorizPos + PIECE_WIDTH);
    pself->VI_bounds.R_bottom = (pself->TTP_origVertPos + PIECE_HEIGHT);

    /* This method does not need to invoke a MSG_VIS_DRAW
     * because the TicTacBoard object will do that. The
     * piece object will later receive a MSG_VIS_DRAW that
     * will cause the piece to be redrawn back at its
     * original location (the newly set bounds). */
}

/*****
*
* MESSAGE:      MSG_VIS_DRAW for TicTacPieceClass
*
* DESCRIPTION:  Draw the piece at the current location. If the piece
*               is a "box," draw a gray square. If the piece is a
*               "ring," draw a gray circle. This message is received
*               whenever a portion of the view window becomes invalid;
*               TicTacView will send a MSG_META_EXPOSED to TicTacBoard,
*               which will send itself (by default) a MSG_VIS_DRAW.
*               The MSG_VIS_DRAW will be handled and then will be

```

10.5



The GEOS User Interface

408

```
*          passed on to each of the game pieces. Then each piece
*          (in this handler) will draw itself at its own bounds.
*
* PARAMETERS:
*   void (word drawFlags GStateHandle gstate)
*
*****/

10.5 @method TicTacPieceClass, MSG_VIS_DRAW {
    /* Set the mode to MM_COPY; this means that the image
       * drawn now will be drawn over whatever is there now.*/

    GrSetMixMode(gstate, MM_COPY);

    /* If the type is TTPT_BOX, set the color to gray and
       * draw a rectangle the size of the object's bounds.
       * Otherwise (since there are just two types), set the
       * color to gray and draw an ellipse of that size.*/

    if (pself->TTP_pieceType == TTPT_BOX) {
        GrSetAreaColor(gstate, CF_INDEX, C_DARK_GRAY, 0, 0);
        GrFillRect(gstate, pself->VI_bounds.R_left, pself->VI_bounds.R_top,
                    pself->VI_bounds.R_right, pself->VI_bounds.R_bottom);
    } else {
        GrSetAreaColor(gstate, CF_INDEX, C_LIGHT_GRAY, 0, 0);
        GrFillEllipse(gstate, pself->VI_bounds.R_left, pself->VI_bounds.R_top,
                      pself->VI_bounds.R_right, pself->VI_bounds.R_bottom);
    }

    /* After handling the message, call the superclass to
       * ensure that no default behavior has been mucked up.
       * This is actually not necessary in this particular case. */
    @callsuper();
}

/*****
* MESSAGE:      MSG_META_START_SELECT for TicTacPieceClass
*
* DESCRIPTION:  Grabs the mouse and calls for future pointer events.
*               When the user clicks in the view, TicTacView will pass
*               the click event to TicTacBoard. Since TicTacBoardClass
*               does not intercept the event, VisContentClass passes
*               it on to its child object currently under the pointer.
*
*               When the piece object receives this message, it means
*               it has been clicked on by the user and the mouse button
*               is still down. The piece must grab the mouse so that it
```

```

*           gets all future mouse events, and it must request that
*           all future mouse events be sent to it. This ensures
*           that if the pointer leaves the object's bounds while
*           the button is still pressed, the piece object will still
*           receive all the pointer events (otherwise they would be
*           sent to whatever object was under the new pointer
*           position).
* PARAMETERS:
*           void (MouseReturnParams *retVal, word xPosition,
*               word yPosition, word inputState)
*****/
10.5

@method TicTacPieceClass, MSG_META_START_SELECT {

    /* First grab the gadget exclusive so we're allowed to
    * grab the mouse. Then grab the mouse, so all future
    * pointer events get passed directly to the game piece. */

    @call @visParent::MSG_VIS_TAKE_GADGET_EXCL(oself);
    @call self::MSG_VIS_GRAB_MOUSE();           /* grab mouse */

    /* Finally, return that this particular click
    * event has been processed. If this flag is
    * not returned, the system will send out the
    * click event again. */

    retVal->flags = MRF_PROCESSED;               /* this event processed */
}

/*****
*
* MESSAGE:      MSG_META_DRAG_SELECT for TicTacPieceClass
*
* DESCRIPTION:  This message is sent to the piece object when the
*               select button has been pressed and the mouse has been
*               moved, resulting in a "drag-select" event.
*               For event processing from the View, see the header
*               for MSG_META_START_SELECT.
*
* PARAMETERS:
*           void (MouseReturnParams *retVal, word xPosition,
*               word yPosition, word inputState)
*****/

@method TicTacPieceClass, MSG_META_DRAG_SELECT {
    GStateHandle gstate;           /* temporary gstate to draw to */
    WindowHandle win;             /* window handle of view window */

```



The GEOS User Interface

410

10.5

```
/* Start off by setting the flag indicating that
 * the piece is being dragged around the screen. */
pself->TTP_dragging = TRUE;

/* Next, get the window handle of the view window.
 * Then, create a new, temporary gstate to draw into
 * for that window. */

win = @call TicTacView::MSG_GEN_VIEW_GET_WINDOW();
gstate = GrCreateState(win);

/* Now, set the current position of the game piece
 * to be centered on the pointer. */

pself->TTP_vertPos = yPosition - (PIECE_HEIGHT/2);
pself->TTP_horizPos = xPosition - (PIECE_WIDTH/2);

/* Now, set the drawing mode of the game piece
 * to MM_INVERT to draw a new game piece outline.
 * MM_INVERT is chosen so the outline can be redrawn
 * in invert mode later to erase it and not destroy
 * anything under it. */

GrSetMixMode(gstate, MM_INVERT);

/* Now draw the outline. If the game piece is of type
 * TTPT_BOX, draw a rectangle outline. Otherwise, draw
 * an ellipse outline. */

if (pself->TTP_pieceType == TTPT_BOX) {
    GrDrawRect(gstate, pself->TTP_horizPos, pself->TTP_vertPos,
                (pself->TTP_horizPos + PIECE_WIDTH),
                (pself->TTP_vertPos + PIECE_HEIGHT));
} else {
    GrDrawEllipse(gstate, pself->TTP_horizPos, pself->TTP_vertPos,
                  (pself->TTP_horizPos + PIECE_WIDTH),
                  (pself->TTP_vertPos + PIECE_HEIGHT));
}

/* Next, destroy the temporary gstate. This is important
 * to make sure the gstate does not stay in memory and
 * begin to slow down the system as more and more
 * temporary gstates are created but not destroyed.*/

GrDestroyState(gstate);

/* Finally, return that this event has been processed
 * by this method. */
```



```

    retVal->flags = MRF_PROCESSED;
}

/*****
 *
 * MESSAGE:      MSG_META_PTR for TicTacPieceClass
 *
 * DESCRIPTION:  This message is received whenever the pointer passes
 *               over this game piece object's bounds (and another
 *               game piece is not sitting directly on top of it).
 *               See MSG_META_START_SELECT for a description of how the event
 *               gets passed from TicTacView to this object.
 *
 *               This message can be either a drag event or a simple
 *               pointer event. If the latter, we want to do nothing
 *               because no mouse button is pressed. If the latter,
 *               we want to execute the same function as MSG_META_DRAG.
 *
 * PARAMETERS:
 *     void (MouseReturnParams *retVal, word xPosition,
 *           word yPosition, word inputState)
 *****/

@interface TicTacPieceClass, MSG_META_PTR {
    GStateHandle gstate;           /* temporary gstate to draw to */
    WindowHandle win;             /* window handle of view window */

    /* First check if this is a drag event. If not, do
     * nothing. If so, then draw a new outline and erase
     * the old outline. */

    if (pself->TTP_dragging) {

        /* Get the view's window handle and create a
         * temporary gstate for drawing into. */

        win = @call TicTacView::MSG_GEN_VIEW_GET_WINDOW();
        gstate = GrCreateState(win);

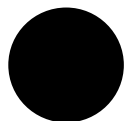
        /* Set the drawing mode of the game piece to
         * MM_INVERT for outline drawing. */

        GrSetMixMode(gstate, MM_INVERT);

        /* Erase the old outline by drawing an inverse
         * outline at the old bounds. */
    }
}

```

10.5



10.5

```
if (pself->TTP_pieceType == TTPT_BOX) {
    GrDrawRect(gstate, pself->TTP_horizPos, pself->TTP_vertPos,
                (pself->TTP_horizPos + PIECE_WIDTH),
                (pself->TTP_vertPos + PIECE_HEIGHT));
} else {
    GrDrawEllipse(gstate, pself->TTP_horizPos, pself->TTP_vertPos,
                  (pself->TTP_horizPos + PIECE_WIDTH),
                  (pself->TTP_vertPos + PIECE_HEIGHT));
}

/* Now set the current motion position to be
 * centered on the pointer. */

pself->TTP_vertPos = yPosition - (PIECE_HEIGHT/2);
pself->TTP_horizPos = xPosition - (PIECE_WIDTH/2);

/* Draw the new outline at the current position.*/

if (pself->TTP_pieceType == TTPT_BOX) {
    GrDrawRect(gstate, pself->TTP_horizPos, pself->TTP_vertPos,
                (pself->TTP_horizPos + PIECE_WIDTH),
                (pself->TTP_vertPos + PIECE_HEIGHT));
} else {
    GrDrawEllipse(gstate, pself->TTP_horizPos, pself->TTP_vertPos,
                  (pself->TTP_horizPos + PIECE_WIDTH),
                  (pself->TTP_vertPos + PIECE_HEIGHT));
}

/* Destroy the temporary gstate and return that
 * this event has been processed. */

GrDestroyState(gstate);
}
retVal->flags = MRF_PROCESSED;
}

/*****
 *
 * MESSAGE:      MSG_META_END_SELECT for TicTacPieceClass
 *
 * DESCRIPTION:  This message is received when the selection button has
 *               been released and this game piece had the mouse grab.
 *               All it does is release the gadget exclusive, which will
 *               cause us to end any dragging in progress and release
 *               the mouse.
 *               When we release the gadget exclusive, the UI will then
 *               sent MSG_VIS_LOST_GADGET_EXCL to this piece, which will
```

```

*           tell us to erase the outline and draw the game piece.
* PARAMETERS:
*           void (MouseReturnParams *retVal, word xPosition,
*               word yPosition, word inputState);
*****/

@method TicTacPieceClass, MSG_META_END_SELECT {
    /* Release the gadget exclusive, then return that the
       * event has been processed. */
    @call @visParent::MSG_VIS_RELEASE_GADGET_EXCL(oself);
    retVal->flags = MRF_PROCESSED; /* this event processed */
}

/*****
*
* MESSAGE:      MSG_VIS_LOST_GADGET_EXCL for TicTacPieceClass
*
* DESCRIPTION:  This message is received when the piece lots go of the
*               gadget exclusive (see MSG_META_END_SELECT, above).
*               It first checks to see if the new, proposed bounds are
*               on the game board. If the bounds are valid, then
*               it sets the objects VI_bounds field to the new values
*               and causes the object to erase its original drawing
*               and draw itself at its new bounds. If the bounds are
*               not on the game board, it will retain the original bounds
*               and redraw using them.
*
* PARAMETERS:
*           void ()
*
*****/

@method TicTacPieceClass, MSG_VIS_LOST_GADGET_EXCL {
    WindowHandle win;           /* window handle of view window */
    GStateHandle gstate;        /* temporary gstate to draw to */

    /* First check if the piece was being dragged.
       * If not, we don't have to do anything. */
    if (pself->TTP_dragging) {

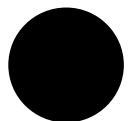
        /* Get the window handle of the view window and
           * create a temporary gstate for it to draw to. */

        win = @call TicTacView::MSG_GEN_VIEW_GET_WINDOW();
        gstate = GrCreateState(win);

        /* Set the mode for drawing the outline.          */

```

10.5



10.5

```
GrSetMixMode(gstate, MM_INVERT);

/* If the game piece type is TTPT_BOX, draw a rectangle
 * outline. Otherwise draw an ellipse outline.  */

if (pself->TTP_pieceType == TTPT_BOX) {
    GrDrawRect(gstate, pself->TTP_horizPos, pself->TTP_vertPos,
               ((pself->TTP_horizPos) + PIECE_WIDTH),
               ((pself->TTP_vertPos) + PIECE_HEIGHT));
} else {
    GrDrawEllipse(gstate, pself->TTP_horizPos, pself->TTP_vertPos,
                  ((pself->TTP_horizPos) + PIECE_WIDTH),
                  ((pself->TTP_vertPos) + PIECE_HEIGHT));
}

/* Check to see if the new bounds are on the game
 * board. If they are, set the object's bounds to the
 * new values. If they are not, retain the original
 * values and set the values to those last stored.*/

if (@call TicTacBoard::MSG_TICTAC_VALIDATE_BOUNDS(
    ((pself->TTP_vertPos) + PIECE_HEIGHT),
    ((pself->TTP_horizPos) + PIECE_WIDTH),
    pself->TTP_vertPos,
    pself->TTP_horizPos)) {

    /* Invalidate the original drawing of the game piece.
     * Send the VI_bounds rectangle as the parameters
     * because they have not been changed since the
     * START_SELECT. This message is the equivalent of
     * calling GrInvalRect() with the same bounds.  */

    @call self::MSG_VIS_BOUNDS_CHANGED(pself->VI_bounds.R_bottom,
    pself->VI_bounds.R_right,
    pself->VI_bounds.R_top,
    pself->VI_bounds.R_left);

    /* Now set the current position to be centered
     * on the pointer image.  */

    pself->TTP_vertPos = yPosition - (PIECE_HEIGHT/2);
    pself->TTP_horizPos = xPosition - (PIECE_WIDTH/2);

    /* Set the game piece object's bounds to
     * the new coordinates.  */
}
```

```
    pself->VI_bounds.R_left = pself->TTP_horizPos;
    pself->VI_bounds.R_right = (pself->TTP_horizPos) + PIECE_WIDTH;
    pself->VI_bounds.R_top = pself->TTP_vertPos;
    pself->VI_bounds.R_bottom = (pself->TTP_vertPos) + PIECE_HEIGHT;
} else {
/* If the bounds are not on the game board, then reset
 * the current positions to be the original bounds. */

    pself->TTP_horizPos = pself->VI_bounds.R_left;
    pself->TTP_vertPos = pself->VI_bounds.R_top;
}

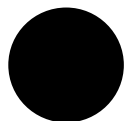
/* Now, the game piece must draw itself at its newly-
 * set bounds (will draw itself over its original
 * picture if the new bounds were invalid). */
@call self::MSG_VIS_DRAW(0, gstate);

/* Destroy the temporary gstate used for drawing. */
GrDestroyState(gstate);

/* Finally, clear the dragging flag to indicate that
 * no drag event is in progress. */
pself->TTP_dragging = FALSE;
}

/* Release the mouse grab now that the move has
 * finished. Other objects in the view (other game
 * pieces, for example) may now receive pointer,
 * select, and drag events.
*/
@call self::MSG_VIS_RELEASE_MOUSE();
}
```

10.5



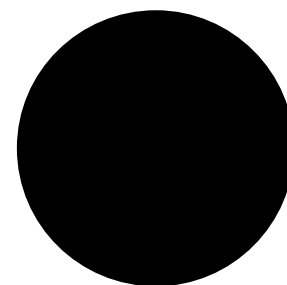
The GEOS User Interface

416

10.5

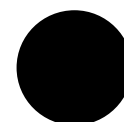


Input



11

11.1	Input Flow.....	419
11.1.1	Devices and Drivers	421
11.1.2	Input Manager and GenSystem.....	422
11.1.3	Input Events	422
11.1.4	Input Hierarchies.....	423
11.2	Mouse Input.....	424
11.2.1	Mouse Events.....	425
11.2.1.1	Structure of Mouse Events	428
11.2.1.2	Return Values for Mouse Events.....	430
11.2.2	Gaining the Mouse Grab	431
11.2.3	Large Mouse Events.....	433
11.2.4	Setting the Pointer Image.....	434
11.2.4.1	Defining the Pointer Image	434
11.2.4.2	Setting the Pointer Image	436
11.3	Keyboard Input	437
11.3.1	Keyboard Input Flow	437
11.3.2	Keyboard Events	438
11.4	Pen Input and Ink.....	442
11.4.1	Ink Data Structures	443
11.4.2	Ink Input Flow	444
11.4.2.1	Determining if a Press Is Ink.....	444
11.4.2.2	Controlling the Ink	446
11.4.2.3	How Ink Is Stored and Passed On	447
11.5	Input Hierarchies	448
11.5.1	The Three Hierarchies	449
11.5.2	Common Hierarchy Basics	449
11.5.2.1	Special Terminology	451
11.5.2.2	Modifying the Active Path.....	451
11.5.2.3	Sending Classed Messages	452



11.5.3	Using Focus	453
11.5.3.1	Grabbing and Releasing the Focus	454
11.5.3.2	Gaining and Losing the Focus	455
11.5.3.3	Sending Classed Events to the Focus	455
11.5.4	Using Target	456
11.5.4.1	Grabbing and Releasing the Target	458
11.5.4.2	Gaining and Losing the Target	458
11.5.4.3	Sending Classed Events to the Target	458
11.5.5	Using Model	460
11.5.5.1	Changing the Model Exclusive	461
11.5.5.2	Intercepting the Model	462
11.5.6	Extending the Hierarchies	462



Nearly all applications will in some way handle user input. User input can come from a keyboard, a mouse, or a stylus. (Other types of input are generally considered data input.) For the most part, GEOS hides the actual mechanics of input from the application; at the same time, it allows extreme flexibility of input handling.

Input to generic UI objects is handled by these objects—unless you have very specific needs and will subclass a generic gadget, you can leave its input handling up to GEOS. The same is true with most specialized system-provided objects such as the GrObj, the VisText, the Spreadsheet, and others. Other objects (such as basic visible objects you subclass yourself) will need more attention depending on the behavior you want them to have.

11.1

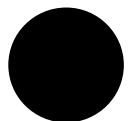
Read this chapter if your application will handle user input. Before reading this chapter, however, you should be relatively familiar with the GEOS system architecture, Goc syntax, and the GEOS object model and message passing. It is also suggested that you spend some time programming with generic and visible objects and getting used to the rest of the system before handling input events.

11.1 Input Flow

User input must flow in an orderly way from the user's device to the proper GEOS object. In a multithreaded environment, just determining the appropriate input recipient can be a monumental task.

GEOS uses several layers of indirection to process user input and to distribute it to its proper destination so applications don't have to. The scheme used not only isolates applications from dealing with devices directly, but it provides convenient and easy ways to control input flow, both within an application and throughout the entire system.

Figure 11-1 shows the flow of input from the user's device to its destination. User input is managed primarily by device drivers acting in concert with the UI, and several threads are involved in the input flow. For example, the kernel provides a thread solely for input management that has a high



priority; this insures that the pointer image always moves immediately with the mouse.

Input is registered by a device such as a keyboard, stylus, or mouse. The device sends signals to GEOS device drivers, which take the signals and translate them into raw input messages the UI understands. These raw messages are translated by the UI's Input Manager, which refines the events even further. The Input Manager then sends the events on to the topmost UI object in the system, the GenSystem object.

The GenSystem examines the events and their contexts and determines which window or application should receive them. Typically, events you will be interested in will flow from the GenSystem through your GenApplication object, its GenPrimary window, and finally through the appropriate GenView or GenDisplay. These window objects will pass the event on to a visible tree (through a VisContent), which will handle the events.

It is highly unlikely you will ever need to know anything about an input event before it gets to your GenApplication object. Typically, you will only

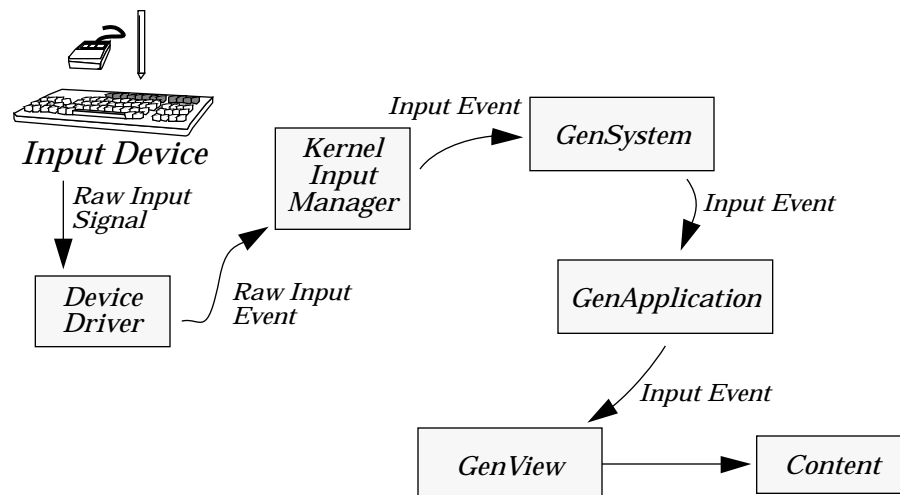


Figure 11-1 *Input Flow*

A device driver translates input signals and passes raw events to the Input Manager, which refines them and passes them to the GenSystem object. The GenSystem passes them to the active application object, which then dispatches them to the appropriate window or object.



Input message reference can be found under MetaClass.

handle mouse, pen, or keyboard events in custom visible objects and can ignore them in all other cases.

Input events take the form of standard GEOS messages. These messages are defined by the UI but are actually part of **MetaClass** (they are allocated in **MetaClass** and exported to the UI). The reference entries for these events can therefore be found under **MetaClass** (see “System Classes,” Chapter 1 of the Object Reference Book).

Input events give information about the type of event (e.g. a mouse event), the action taking place (e.g. the right mouse button was pressed), and additional information (e.g. the location of the mouse at the time). Each device has its own set of events—for example, keyboard presses induce MSG_META_KBD_CHAR, and mouse moves induce MSG_META_MOUSE_PTR.

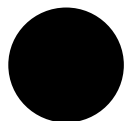
11.1

11.1.1 Devices and Drivers

GEOS supports a number of input devices through the use of device drivers. The basic GEOS system software supports mouse, keyboard, and stylus input. Other input devices may be supported by the system software in the future by the addition of new device drivers. Other devices may provide data input in one form or another—a scanner, for example—but these devices are not considered *user* input devices. These are considered *data* input devices and do not generate input events through the Input Manager.

The use of device drivers isolates your application and libraries from the specifics of the user’s machine. GEOS includes drivers for most mice and keyboards (pen input uses a derivative of the mouse drivers), and drivers for new hardware may be added without requiring changes to your application.

A device driver takes raw input signals from the input device and translates it into raw input events that the Input Manager will understand. Your objects will never receive these events; they are only sent to the Input Manager for translation into pointer, button, or keyboard events.



11.1.2 Input Manager and GenSystem

The Input Manager is part of the User Interface and primarily assesses input events in the context of the system. It takes a raw input event from the device driver, passes it through various pre-set monitors (for the UI), and then adds contextual information if necessary. The Input Manager passes the refined input event on to the topmost system object, the GenSystem object.

11.1

The GenSystem object determines where the event should be passed. Typically, it will send the event on to whichever application is designated as the active application.

11.1.3 Input Events

GEOS uses three basic types of input events. These three types make up all the events necessary for mouse, keyboard, and pen input. They are *pointer events*, *button events*, and *keyboard events*.

Pointer and button events deal with mouse and pen input. Keyboard events provide keyboard key press information. Pointer and button events are often referred to as “mouse” events and are dealt with together because most objects interested in one type will also be interested in the other.

A Pointer event details motion of the mouse pointer. If the user moves the mouse, a MSG_META_MOUSE_PTR will be generated by the Input Manager and sent to the proper application. The type of pointer event (not always MSG_META_MOUSE_PTR) is affected by the current keyboard state, mouse state, and other context information.

A Button event details the press or release of a particular mouse button. Because different Specific UIs have different meanings for each mouse button, the function of the button determines which message is sent out. In the OSF/Motif specific UI, the left mouse button is used for selection; therefore, presses on that button cause MSG_META_START_SELECT, and releases cause MSG_META_END_SELECT. The exact message sent out, like pointer events, can also be affected by input state information.

Mouse events also carry with them several flags indicating various statistics about the event: whether the click is actually a double-click, whether any keys on the keyboard are being held down, etc.

A keyboard event details the press or release of a particular key on the user's keyboard. Usually, objects interested in keyboard events will only be interested in presses and can ignore releases. `MSG_META_KBD_CHAR` is the basic keyboard event, and it carries flags detailing the state of the various modifier keys (e.g. Shift, Ctrl, Alt, NumLock). This message is always sent to the current focus.

Pen input comes in the form of a special data structure called *Ink*. Ink input is instigated in the same manner as button events, but it is handled in a completely separate manner. An object expecting Ink input must be aware when Ink events are being passed as opposed to normal mouse events. Special flags are used for this; see section 11.4 on page 442.

11.1

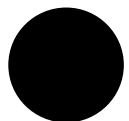
11.1.4 Input Hierarchies

Because the entire User Interface of GEOS is implemented as a hierarchy (tree) of objects, input messages naturally flow from the topmost object in the tree down to the object that should receive the input. At each level in the tree, the object decides whether to handle the event or pass it on to one of its children. Typically, the event will be passed all the way down the tree to a leaf node, where it will be handled or ignored.

To simplify and shorten this process, the UI maintains several special hierarchical input flow channels where special input should always flow. These hierarchies include the *focus*, the *target*, and the *model*. All three are described in full in section 11.5 on page 448.

The Focus is the object that should receive all keyboard events. For example, a modal dialog box always has the focus; if the user presses Enter, the dialog box will take its default action. A text object into which the user is typing has the focus as long as the user does not click outside its bounds.

The Target is what is selected by the user: the object on which actions should be performed. Controller objects (of **GenControlClass**) typically operate on the target. An example is a text object with selected text. When you set the text's style using a menu item (a style control object), that menu item sends its message to the current target object. The text object, set as the current Target, receives the message and sets the style of the selection.



An object is said to “have the focus” or “have the target” if it is the object specified by the hierarchy to receive that type of input. The system provides special *travel options* to allow you to send messages directly to the focus or target without knowing in advance which object is the destination.

The Model represents the non-visible model of the application’s data. For example, a secondary selection in a database will not be acted on like the normal target, but it may still need to have similar properties. The Model hierarchy can be used to maintain this type of secondary or non-visible selection.

11.2

11.2 Mouse Input

GEOS is built around the user’s use of a mouse for most input. Although the system has extensive keyboard navigation capabilities, the mouse is still the preferred input tool for a GUI. Therefore, most applications will in some way accept and handle mouse input directly.

Typically, you will handle mouse input if you are using visible objects (**VisClass**, etc.) that will react to the mouse. If you are using only generic objects in your application, you will not need to handle mouse events.

Mouse input comes in the form of pointer and button events. Each event describes the mouse’s location, the type of event (move, button press, etc.), whether any buttons on the mouse are pressed, and whether any modifier keys on the keyboard are pressed.

Objects can “grab” the mouse exclusive, taking all mouse events for themselves. This is useful when one object needs to get all mouse input for a period of time; when it no longer needs all mouse input, it releases the exclusive. Mouse events are dispatched directly to whichever object has the “active grab” (the exclusive).

If no object has the active grab, the event will be passed to the application under the pointer. The application will pass the event to the window or GenView under the pointer, and the window or view will pass it to its content. If the view displays a visible tree, the event will eventually reach the visible object under the mouse pointer, where it should be handled. Thus, the object under the mouse pointer is said to have the “implied grab.”

The default behavior of the VisContent (passing the event to its first child with the appropriate bounds) is adequate for most input needs. If you are using large documents or visible layers, you will have to subclass the VisContent to handle these special cases.

11.2.1 Mouse Events

Each time the user moves the mouse or clicks a mouse button, GEOS generates a mouse event and passes it to the proper object. Mouse events are actually **MetaClass** messages that any object may intercept. Default handlers for these messages typically do nothing, so if you do not handle a particular event, it will likely be ignored.

11.2

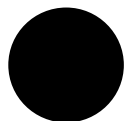
GEOS uses three basic types of mouse events: button, pointer, and drag events. Button events indicate when the user has either pressed or released any mouse button (double-clicks are considered button events). Pointer events indicate that the mouse has been moved. Drag events indicate the user has clicked and dragged the mouse. All of these types pass information about the mouse context and the operations currently in progress.

The Input Manager checks both the button state and the mouse movement before deciding on an event's type. For example, if the user clicks twice quickly, the Input Manager decides whether the user intended a double-click or a click-move-click. Objects receiving the events do not have to differentiate between the two; the events automatically contain the required information.

Pointer events are fairly straightforward; when the mouse is moved, the Input Manager generates the proper MSG_META_MOUSE_PTR indicating the move. This message also indicates the state of the mouse buttons and keyboard modifier keys.

Button events are quite a bit more complex. With each button press or release, the Input Manager generates a MSG_META_MOUSE_BUTTON which gets translated into the proper press/release/drag event before being passed on to the application. The final message generated depends on which button was pressed, whether large graphics coordinates are being used, and whether the event is being sent pre-passive or post-passive.

Among button events, there are two basic types: the press and the release. All press events are of the form MSG_META_START_..., and all release events



are of the form MSG_META_END_.... The last portion of the message name is the function of the button pressed or released. The various buttons are referred to by their meaning to the Specific UI, as follows:

- | | |
|-----------|--|
| SELECT | The button used for making selections. In OSF/Motif and many other Specific UIs, this is the left mouse button. |
| MOVE_COPY | The button used for quick-copies and quick-moves of data. In OSF/Motif, this is the right mouse button. For single-button mice, a key sequence plus a mouse click is often used. |
| FEATURES | The button used to bring up a “features” pop-up menu or dialog box. In some Specific UIs, this is the middle or the right mouse button. For single- or two-button mice, a key sequence plus a mouse click is often used. |
| OTHER | Any button not designated one of the three categories above. This category can also be used to indicate when the user presses more than one button at time (often referred to as “chording”). |

A press event indicates the user pressed down on the particular button. A release event indicates the user released the button. If the user presses a button and moves the mouse, a drag event will be sent after the initial press event. If the user double-clicks, a special flag will be sent with a single press event; it is up to the application to handle double-clicks differently.

Drag events are of the form MSG_META_DRAG_..., similar to button events. Each of the above button types has a corresponding drag message. If the user presses a mouse button and quickly moves the mouse more than a specified distance, or if he holds a particular mouse button down more than a specified time, the Input Manager will send a drag event after the press event. A single release event signifies the user released the mouse button, just as with normal presses.

There is also a complete set of events used for large documents. If an object has large bounds, or if a GenView is set up to display a large content, large mouse events will be generated instead of normal mouse events. Large events take the form MSG_META_LARGE_.... For example, the large version of MSG_META_MOUSE_PTR is MSG_META_LARGE_PTR.

Below are listed all the standard mouse events your objects may be interested in handling. Most objects will be interested in only a small subset of these.

MSG_META_MOUSE_PTR

The standard pointer event, generated whenever the mouse moves without a button down.

MSG_META_START_SELECT

Generated when the user presses the select button.

11.2

MSG_META_END_SELECT

Generated when the user releases the select button.

MSG_META_START_MOVE_COPY

Generated when the user presses the move/copy button.

MSG_META_END_MOVE_COPY

Generated when the user releases the move/copy button.

MSG_META_START_FEATURES

Generated when the user presses the features button.

MSG_META_END_FEATURES

Generated when the user releases the features button.

MSG_META_START_OTHER

Generated when the user presses a button combination not recognized as select, move/copy, or features.

MSG_META_END_OTHER

Generated at the release of the combination that generated the MSG_META_START_OTHER.

MSG_META_DRAG_SELECT

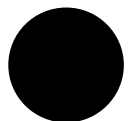
Generated between a press and release of the select button if the user holds the button down or if the user moves the mouse.

MSG_META_DRAG_MOVE_COPY

Generated between a press and release of the move/copy button if the user holds the button down or if the user moves the mouse.

MSG_META_DRAG_FEATURES

Generated between a press and release of the features button



if the user holds the button down or if the user moves the mouse.

MSG_META_DRAG_OTHER

Generated between a press and release of a button not specified above if the user holds it down or if the user moves the mouse.

Listed below are the large equivalents of the above messages.

11.2

MSG_META_LARGE_PTR
MSG_META_LARGE_START_SELECT
MSG_META_LARGE_END_SELECT
MSG_META_LARGE_START_MOVE_COPY
MSG_META_LARGE_END_MOVE_COPY
MSG_META_LARGE_START_FEATURES
MSG_META_LARGE_END_FEATURES
MSG_META_LARGE_START_OTHER
MSG_META_LARGE_END_OTHER
MSG_META_LARGE_DRAG_SELECT
MSG_META_LARGE_DRAG_MOVE_COPY
MSG_META_LARGE_DRAG_FEATURES
MSG_META_LARGE_DRAG_OTHER

All the normal (as opposed to large) mouse events pass and return the same values. Each event differs based on the message itself; an object knows that MSG_META_START_SELECT is inherently different from MSG_META_START_MOVE_COPY, even though they may pass the exact same values.

11.2.1.1 Structure of Mouse Events

Each mouse event passes three items of data and one pointer to a return structure. The three parameters are listed below; the fourth, the return structure, is detailed in the next section.

<i>xPosition</i>	The horizontal position of the pointer in document coordinates in the window when the event was generated.
<i>yPosition</i>	The vertical position of the pointer in document coordinates in the window when the event was generated.

inputState A word of flags indicating the state of the mouse buttons and the UI functions (such as quick-transfer) that were active when the event was generated.

The first two indicate the position of the mouse in the document. The third, *inputState*, consists of two bytes of flags. The first byte indicates the type of button event and the state of the mouse buttons during the event. It is a record of type **ButtonInfo** and has seven flags:

BI_PRESS Set if this is a press event rather than a release or drag. 11.2

BI_DOUBLE_PRESS
Set if this is actually a double-press (GEOS automatically detects double presses).

BI_B3_DOWN Set if button number three is being held down.

BI_B2_DOWN Set if button number two is being held down.

BI_B1_DOWN Set if button number one is being held down.

BI_B0_DOWN Set if button number zero is being held down.

BI_BUTTON Set if this is a button event, clear if a pointer event.

The second byte of *inputState* is a record of **UIFunctionsActive**, which describes which of several UI functions are currently underway. The flags set in this byte are used primarily by the UI, and you will probably not have to check them. The flags allowed, however, are listed below.

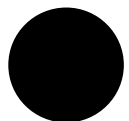
UIFA_SELECT
Set if the basic mouse function is underway.

UIFA_MOVE_COPY
Set if a move/copy (quick-transfer) operation is underway.

UIFA_FEATURES
Set if the features popup menu or dialog is open.

UIFA_CONSTRAIN
Set if a modifier key set as a “constraint” key is pressed (e.g. the user holds the shift key while grabbing an object control point).

UIFA_ADJUST
This is the same bit as UIFA_MOVE and UIFA_POPUP, below. When UIFA_SELECT is also set, this flag indicates that the select event should act as a toggle event.



UIFA_EXTEND

This is the same bit as UIFA_COPY and UIFA_PAN, below. When UIFA_SELECT is also set, this flag indicates that the select event should extend any selection already made (add the area/objects to the current selection).

UIFA_MOVE This is the same bit as UIFA_ADJUST and UIFA_POPUP. When UIFA_MOVE_COPY is also set, this flag indicates that the operation should be considered a move rather than a copy.

UIFA_COPY This is the same bit as UIFA_EXTEND and UIFA_PAN. When UIFA_MOVE_COPY is also set, this flag indicates the operation should be considered a copy rather than a move.

UIFA_POPUP This is the same bit as UIFA_ADJUST and UIFA_MOVE, above. When UIFA_FEATURES is also set, this flag indicates the “features” button brings up a popup menu or dialog box.

UIFA_PAN This is the same bit as UIFA_EXTEND and UIFA_COPY, above. When UIFA_FEATURES is also set, this flag indicates the “features” button has initiated pan-style scrolling.

11.2.1.2 Return Values for Mouse Events

One of the parameters of every mouse event is a pointer to a **MouseReturnParams** structure. This structure is passed empty; it is up to the event handler to fill it with the proper return values. The **MouseReturnParams** structure’s definition is given below:

```
typedef struct {
    word                unused; /* for alignment */
    MouseReturnFlags    flags;
    optr                ptrImage;
} MouseReturnParams;
```

Every time you handle a mouse event, you must fill in the *flags* field. This describes how the event was handled so the UI knows whether to pass it on to another object, change the pointer image, or treat the event as having been handled. The flags you can return are

MRF_PROCESSED

The event was handled. If you do not set this flag upon return,

the window will think the event was not handled and may pass it on to another child.

MRF_REPLAY

The event may or may not have been handled, but it should be rebroadcast as if it had not been. This is used primarily when an object gives up the mouse grab because the mouse has strayed outside its bounds.

MRF_PREVENT_PASS_THROUGH

11.2

The event should *not* be passed through to the active or implied grab object. This is set only by pre-passive handlers which want to filter the event and keep the active or implied grabs from receiving it.

MRF_SET_POINTER_IMAGE

The pointer should be set to a new image. If this flag is returned, you must also return an *optr* in the *ptrImage* field (see below).

MRF_CLEAR_POINTER_IMAGE

The pointer image will be reset to the default. See “Setting the Pointer Image” on page 434.

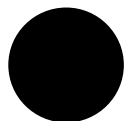
If you return MRF_SET_POINTER_IMAGE, you must specify the new image to be set. An object may want to set the pointer to a new image when it is over the object’s bounds or during a drag operation. For example, if the user selects a drawing tool, the tool might set the pointer to cross-hairs instead of the default arrow.

To set the pointer’s image, you must return the handle and chunk handle of a chunk containing a **PointerDef** structure. Return it in the form of an *optr* in the *ptrImage* field of the return structure. For full information on **PointerDef** and defining pointer images, see section 11.2.4 on page 434.

If you are not setting the pointer image, return a *NullOptr* in *ptrImage*. You do not have to return anything in the *unused* field.

11.2.2 Gaining the Mouse Grab

When no object has the active mouse grab, mouse events are passed to whichever object is directly under the pointer’s image. If any object has the



grab, however, mouse events will be passed only to the object with the grab, no matter where the pointer is on the screen.

When the user clicks inside a view, the GenView automatically grabs the mouse while the user holds the mouse button down. This allows the view to know when drag scrolling takes place (the user drags the pointer outside the view's bounds, but the mouse event is still passed to the GenView). Process objects acting as a GenView's content therefore do not have to worry about grabbing the mouse. If you are displaying visible objects within the GenView, however, your objects will likely want to grab the mouse for themselves.

Typically, a visible object will grab the mouse on a `MSG_META_START_...` event and release it on a `MSG_META_END_...` event. To grab the mouse, a visible object has to send itself the message `MSG_VIS_GRAB_MOUSE`; to release the grab, it must send itself `MSG_VIS_RELEASE_MOUSE`.

The TicTac sample application shows an example of visible objects grabbing the mouse and handling several mouse events. This sample application is described in detail in "The GEOS User Interface," Chapter 10.

There are two other types of mouse grabs besides the active grab. With very rare exceptions, application programmers can ignore these. They are the pre-passive and post-passive grabs.



Advanced Topic

You will probably never need to use pre- or post-passive grabs.

If an object has a pre-passive grab, it will receive copies of all button events before the events are passed on to their true destinations. Pre-passive grabs are used by window objects to bring themselves to the front when the user clicks in them. Any number of objects may have pre-passive grabs. To gain a pre-passive grab, a visible object must send itself `MSG_VIS_ADD_BUTTON_PRE_PASSIVE`; to release it, the object must send itself `MSG_VIS_REMOVE_BUTTON_PRE_PASSIVE`.

If an object has a post-passive grab, it will receive copies of all button events after the events have been handled. Any number of objects may have post-passive grabs. To gain a post-passive grab, a visible object must send itself `MSG_VIS_ADD_BUTTON_POST_PASSIVE`; to release the grab, the object must send itself `MSG_VIS_REMOVE_BUTTON_POST_PASSIVE`.

GEOS uses different messages to indicate that the event is a pre-passive or a post-passive event. All the normal button events have pre-passive and post-passive counterparts which take the form `MSG_META_PRE_PASSIVE_...`

and MSG_META_POST_PASSIVE_.... For example, the pre-passive version of MSG_META_START_SELECT is MSG_META_PRE_PASSIVE_START_SELECT.

Below are listed the pre-passive equivalents of the above messages. Note that the drag events and large events do not have pre-passive equivalents.

```
MSG_META_PRE_PASSIVE_START_SELECT
MSG_META_PRE_PASSIVE_END_SELECT
MSG_META_PRE_PASSIVE_START_MOVE_COPY
MSG_META_PRE_PASSIVE_END_MOVE_COPY
MSG_META_PRE_PASSIVE_START_FEATURES
MSG_META_PRE_PASSIVE_END_FEATURES
MSG_META_PRE_PASSIVE_START_OTHER
MSG_META_PRE_PASSIVE_END_OTHER
```

11.2

Listed below are the post-passive equivalents of the above messages. Note that the drag events and large events do not have post-passive equivalents.

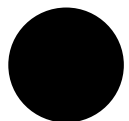
```
MSG_META_POST_PASSIVE_START_SELECT
MSG_META_POST_PASSIVE_END_SELECT
MSG_META_POST_PASSIVE_START_ADJUST
MSG_META_POST_PASSIVE_END_MOVE_COPY
MSG_META_POST_PASSIVE_START_FEATURES
MSG_META_POST_PASSIVE_END_FEATURES
MSG_META_POST_PASSIVE_START_OTHER
MSG_META_POST_PASSIVE_END_OTHER
```

11.2.3 Large Mouse Events

Applications and visible objects that use large documents (32-bit coordinate spaces) receive special versions of the mouse input messages. These special messages pass the pointer position differently; otherwise, they pass the same flags as passed by their normal counterparts.

Large mouse messages take the form MSG_META_LARGE_.... There is one large mouse event for each of the normal pointer, button, and drag events; note, however, that no large events exist for pre-passive or post-passive grabs. For the full list of large mouse events, see page 428.

One special note about large events: Any object that handles large mouse events should also be prepared to handle normal mouse events. The visible



parent of the large object can only send out one type (large or normal) of event at a time to its children; if it has any normal children at all, it will send out the normal event. If the large object should handle the normal event and doesn't, the event will be lost or passed on to another child.

For full information on handling large mouse events within a visible object tree, see "VisClass," Chapter 23 of the Object Reference Book. (This information also applies to Process objects acting as the contents of GenViews.)

11.2

11.2.4 Setting the Pointer Image

The GEOS User Interface uses several default settings for the mouse pointer, all defined by the Specific UI.

Some applications will want to modify the mouse pointer's image in order to provide certain feedback to the user. For example, a chess game may set the pointer to an image of the piece being moved and then reset it to an arrow after the move is completed; a graphics application could set the pointer to an image of whatever tool is currently in use. Another time an application may change the mouse pointer is when a quick-transfer is in progress.

11.2.4.1 Defining the Pointer Image

The pointer is defined as a bitmap 16 pixels on each side. It has a "hot spot" of five pixels that acts as the active point of the image. When the user clicks, the area affected is the hot spot, not the entire image. See Figure 11-2 for a diagram of the pointer and the hot spot.

The pointer is stored in a **PointerDef16** structure. This structure defines the image bitmap, the horizontal and vertical offsets to the hot spot, and the definition of how the image should mix with the background. This definition is known as the pointer's mask; how it affects the image is shown in Table 11-1.

The definition of **PointerDef16** is given below. When setting the pointer image, you will typically pass an optr (a combination handle and chunk handle) to a **PointerDef16** structure; the structure can be set up in a sharable chunk beforehand, or it can be set in a local or global variable.

Typically, you will set the entire pointer image in a resource chunk and compile it into your geode; this is much easier and faster to use than building it at run-time.

```
typedef struct {
    byte    PD_width;
    byte    PD_height;
    sbyte   PD_hotX;
    sbyte   PD_hotY;
    byte    PD_mask[STANDARD_CURSOR_IMAGE_SIZE];
    byte    PD_image[STANDARD_CURSOR_IMAGE_SIZE];
} PointerDef16;
```

11.2

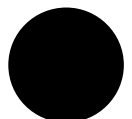
The *PD_hotX* field defines the horizontal offset from the upper-left corner of the image to the center of the hot spot. Similarly, *PD_hotY* defines the vertical offset from the upper-left corner to the hot spot. Both are in pixels and must be less than 16 (since the image is only 16 pixels in each dimension).

The *PD_mask* field contains the 16 by 16 bitmap of the mask, while *PD_image* holds the bitmap of the image. Both are stored as 32-byte arrays, two bytes per row. For example, if the first line of the image was all black and the second all white, the first four bytes of *PD_image* would be 0xFF, 0xFF, 0x00, and 0x00.

Table 11-1 *The Pointer Image Mask*

Image Pixel	Mask Pixel	Drawing Result
0	0	Screen Pixel Unchanged
1	0	Screen Pixel XOR Image Pixel
0	1	Screen Pixel drawn black
1	1	Screen Pixel drawn white

The mask, applied to the image bitmap, results in the mixing behavior given for the screen pixel.



Note that the block the pointer chunk is stored in must be declared sharable.

11.2

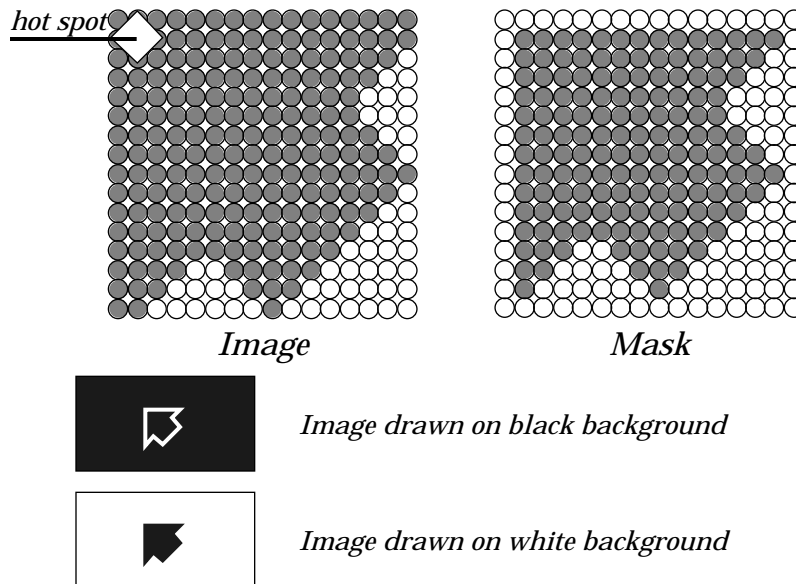


Figure 11-2 A Pointer Image and Its Mask

The mask determines how the image mixes with the background. The outline of the arrow is XORed with the background; the interior of the arrow is always drawn black.

11.2.4.2 Setting the Pointer Image

You can set the pointer's image in three ways. First, you can send a message to the GenView to set the pointer image whenever the pointer is over the view's window (see "GenView," Chapter 9 of the Object Reference Book). Second, you can set the pointer to a default image type during a quick-transfer operation (see "The Clipboard," Chapter 7). Third, you can set the pointer image after handling a mouse event. Only the third option is discussed here.

An object may wish to set the pointer image while it is selected, while the pointer is over its bounds, or in any number of other situations. For example,

a chess piece currently being moved may set the pointer's image to the shape of the piece so the user knows exactly what type of piece is being moved.

Every mouse event you handle must return a **MouseReturnParams** structure, defined in "Return Values for Mouse Events" on page 430. Make sure to return the following to set the pointer image:

- ◆ **MRF_SET_POINTER_IMAGE** in *flags*
When the UI sees this flag in the returned structure, it looks for the address of a pointer definition in the *ptrImage* field. If you do not return this flag, the UI will not try to set the pointer. 11.3
- ◆ The *optr* of the pointer image in *ptrImage*
This field gives the handle and chunk handle of the **PointerDef** structure that defines the pointer image to be set. The structure must be in a sharable block; most likely you will have it set in a geode resource.

11.3 Keyboard Input

Keyboard input is scanned by a device driver and passed on to the Input Manager, just as mouse input is. The keyboard driver and Input Manager check the state of the modifier keys (shift, ctrl, alt, num lock, etc.) and parse each keypress as much as possible based on its context. Each keypress is called a *keyboard event* and is passed in MSG_META_KBD_CHAR.

11.3.1 Keyboard Input Flow

Each time the user presses a character (with or without modifier keys) on the keyboard, the Input Manager generates a single keyboard event and passes it to the GenSystem object. The GenSystem object delivers the event to all the proper objects based on passive grabs (if any) and the current focus hierarchy.

As with mouse events, all keyboard events are passed down the object hierarchy from the GenSystem object through an application's GenApplication object and on down to a leaf object. When handling keyboard input, you can choose to intercept the events at the GenApplication level or at any node below it where the event should be handled.

Unlike with mouse input, there is no “implied grab” for keyboard input. Keyboard input is always passed to the object that has the focus (for information on the Focus hierarchy, see section 11.5.3 on page 453). Therefore the focus is the keyboard’s “active grab.”

Objects can, however, set up passive grabs as they can with mouse input. A pre-passive grab allows the visible object to receive keyboard events before the focus object gets them. A post-passive grab delivers events to the visible object after they have been sent to the focus. For more information about gaining and releasing the passive keyboard grabs, see “VisClass,” Chapter 23 of the Object Reference Book.

If you intercept keyboard events, you must be sure to return any events that you don’t use. For example, if you only want to detect when the user hits the “a” key, you must return all keypresses that are not “a.” Otherwise, your `MSG_META_KBD_CHAR` handler will eat all keypresses, and keyboard menu navigation will not work. If intercepting keyboard events in a generic object, you can simply call the superclass for each unused character. If intercepting keyboard events in a visible object, you must send `MSG_META_FUP_KBD_CHAR` for each unused character.

To ensure that a particular object (for example, a specific GenText object) automatically will receive keyboard input, you must set `HINT_DEFAULT_FOCUS` for that object and all its parents up to the GenPrimary containing it. Otherwise, the generic UI does not know which object should gain the default focus.

11.3.2 Keyboard Events

Each keyboard event your application and objects receive has gone through preliminary parsing by the keyboard driver. Keyboard drivers are intelligent enough to know several things about different types of keystrokes depending on the character set and the modifier keys held down.

All keyboard events pass two sets of information: the current character as interpreted by the keyboard driver, and the actual scan code of the character pressed. This allows you to do additional parsing or to ignore any extraneous information about the keypress that may have been included by the driver.

For example, if the user presses **ctrl-1**, the keyboard driver passes on the character “1” with a flag indicating the control key was pressed. If the user presses **shift-1**, the keyboard driver passes on the “!” character *without* a flag indicating the shift key is pressed (the shift key is eaten by the keyboard driver); in this case, it also passes on the scan code of the “1” key. If the user then presses **ctrl-shift-1**, different keyboard drivers may pass different characters. Whether the driver passes “1” or “!,” however, the scan code for the “1” key will also be passed.

11.3

The keyboard driver also understands special “extended” keypresses and “temporary accents”. Some keyboard drivers may use two keystrokes to specify special characters; for example, some keyboard drivers may require the ö character to be entered as “ctrl-q o” (this is called an extended keypress), and some may require it to be entered as two separate keys: the “o” and the umlaut (this is called a temporary accent).

In addition to the actual character and the scan code, every keyboard event gives flags indicating the state of the modifier keys (ctrl, alt, shift), the state of the toggle keys (caps lock, num lock, etc.), and what type of event it is (first press, repeated press, release, or first press of an extended sequence).

Standard keyboard events come in MSG_META_KBD_CHAR. This message has the same parameters and return values as its pre-passive and post-passive counterparts, MSG_META_PRE_PASSIVE_KBD_CHAR and MSG_META_POST_PASSIVE_KBD_CHAR. The parameters are listed below:

<i>character</i>	The character value determined by the keyboard driver and Input Manager.
<i>flags</i>	A word value: The high byte is a record of ShiftState detailing the modifier keys pressed, and the low byte is a record of CharFlags giving information about the type of character passed. Both of these records are detailed below.
<i>state</i>	A word value: The high byte is the scan code of the key pressed (without modifiers), and the low byte is a record of ToggleState detailing the state of the toggle keys. ToggleState is detailed below.

Three different records of flags define the keyboard event. The **ShiftState** record describes which modifier keys are pressed and has the following flags:

SS_LALT The left Alt key is pressed.



SS_RALT The right Alt key is pressed.

SS_LCTRL The left Ctrl key is pressed.

SS_RCTRL The right Ctrl key is pressed.

SS_LSHIFT The left Shift key is pressed.

SS_RSHIFT The right Shift key is pressed.

SS_FIRE_BUTTON_1
The first joystick-style “fire button” (if any) is pressed.

SS_FIRE_BUTTON_2
The second joystick-style “fire button” (if any) is pressed.

The **ToggleState** record describes which toggle keys are currently active. It has the following flags.

TS_CAPSLOCK
The Caps Lock is set.

TS_NUMLOCK
The Num Lock is set.

TS_SCROLLLOCK
The Scroll Lock is set.

The **CharFlags** record contains several flags indicating whether this event is a first press, a release, a repeat press, or an extended or temporary character. Its flags are listed below.

CF_STATE_KEY
Set if either a **ShiftState** key or a **ToggleState** key is being pressed along with the passed character.

CF_EXTENDED
Set if this event is part of an extended keystroke. This flag is generally only used by the keyboard driver during its parsing; you will not have to use it. (Extended keypresses are passed by the Input Manager to your application as a single event with the resultant character, not as two events.)

CF_TEMP_ACCENT
Set if this event is part of a temporary accent keystroke. Temporary accents are used only by the keyboard driver, like extended keypresses above.

CF_FIRST_PRESS

Set if the event represents the user's first press of the key. This is akin to a button press event and will, at some point, be followed by a CF_RELEASE event.

CF_REPEAT_PRESS

Set if the event is generated by the user holding down the key (as opposed to a first press, above). If you want the user to hit the key for each individual character, you should ignore this type of event.

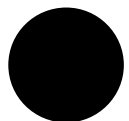
11.3

CF_RELEASE Set if the user lets up on a key. Applications typically are not interested in these events for text, and they can automatically be ignored by setting GVA_DONT_SEND_KBD_RELEASES in your GenView's instance data (this only affects handling keyboard events in visible objects).

A sample handler for MSG_META_KBD_CHAR is shown in Code Display 11-1. It is used by a visible object that simply increments one of its instance fields each time the greater-than key (>) is pressed.

Code Display 11-1 Sample MSG_META_KBD_CHAR Handler

```
/* This method is used by a visible object of MyVisObjClass. It takes all
 * occurrences of the greater-than (>) key and increments the object's MVOCI_data
 * instance field, ignoring when the user holds down the key. Also, the GenView
 * is set to pass along key releases as well as presses; the method must also
 * ignore releases.
 *
 * Note that the object will only get keyboard input when it has the focus. */
/* The format of this message is
 * void (word character, word flags, word state) */
@method MyVisObjClass, MSG_META_KBD_CHAR {
    /* First, check if the character is the one we want. */
    if (character == '>') {
        /* If it is, make sure the user is not holding the key down and
         * that this is not a release event. Check that CF_REPEAT_PRESS and
         * CF_RELEASE are not set in the low byte of the flags parameter.
         * If either is set, ignore the character and send MSG_META_FUP_KBD_CHAR
         * so the UI may provide default keyboard navigation. */
    }
```



```
if ((flags & CF_REPEAT_PRESS) || (flags & CF_RELEASE)) {
    @call self::MSG_META_FUP_KBD_CHAR(character, flags, state);
} else {
    /* If we get here, we know the character is what we want. Increment
     * the instance field and return. */

    (pself->MVOCI_data)++;
}

/* If the character is not a greater-than, we must send it on to the UI
 * with MSG_META_FUP_KBD_CHAR. If we do not, all other keyboard presses
 * we receive will never be handled; this will cause keyboard
 * accelerators and menu navigation not to work. */
} else {
    @call self::MSG_META_FUP_KBD_CHAR(character, flags, state);
}
}
```

11.4 Pen Input and Ink

GEOS supports pen-based systems and applications by means of the Ink data structure formats. Ink is simply a standard way of representing raw pen input as a data structure.

At the lowest level, individual visible objects can receive Ink input directly, much like they receive mouse input normally. (In addition, the VisText object has some Ink handling and handwriting recognition built into it.)

The Pen Library exports InkClass, a visible object class which accepts and stores pen input. It stores the ink compactly. The ink object allows the pen to act as an eraser, removing part or all of a pen stroke, perhaps splitting some strokes in two. See “Pen Object Library,” Chapter 21 of the Object Reference Book for more information.

If GEOS is running on a pen-based system, it will set a particular flag in its private data. It will also install a special input monitor to alter the events the Input Manager receives from the device driver. The Input Manager and the UI then react to input events in a different manner and do not send them directly as with mouse events.

Because Ink events are not passed on immediately by the UI, the UI actually takes care of drawing them directly to the screen. This provides the user direct, immediate feedback.

11.4.1 Ink Data Structures

Ink input is stored in data blocks. The Input Manager stores up Ink events into a data block and then transfers the block to the proper window or application with MSG_META_NOTIFY_WITH_DATA_BLOCK. (This is a general change notification message.) The type of the data block passed is NT_INK, the manufacturer ID is MANUFACTURER_ID_GEOWORKS.

11.4

The data block is headed by an **InkHeader** structure (defined below). The header is followed by a list of points, all in screen coordinates (not points as with normal graphics commands). The block will contain as many points as are registered during the Ink input sequence.

The **InkHeader** structure has the following format:

```
typedef struct {
    word      IH_count;
    Rectangle IH_bounds;
    optr      IH_destination;
    dword     IH_reserved;
    Point     IH_data;
} InkHeader;
```

The fields of the structure are listed below with the information they contain.

IH_count The number of points stored in the data block.

IH_bounds The bounds of the Ink input on the screen (in document coordinates).

IH_destination
The destination of the Ink input. An object can use this field to see if the Ink was sent directly to it or if it received the Ink simply due to an overlap.

IH_reserved Internal data; do not use.



IH_data Actually a label indicating the beginning of the list of points. Following this label will be a number of **Point** structures, each one detailing a single Ink point.

11.4.2 Ink Input Flow

11.4

When the Input Manager receives a button event, it checks to see if that event should be treated as Ink input. It first holds up input, putting the button event in the holdup queue, and then it queries the appropriate application with `MSG_META_QUERY_IF_PRESS_IS_INK`. Typically the application will pass the message down its object tree to the appropriate visible or generic object that should initially have received the button event.

11.4.2.1 Determining if a Press Is Ink

It is up to the application or its visible object to determine whether the input should be treated as Ink. If you don't ever handle Ink, you do not need to do anything different. If any of your objects handles Ink, however, you should set up your GenView to pass Ink events on to its content. Depending how you set up your GenView, you may or may not have to write a handler for `MSG_META_QUERY_IF_PRESS_IS_INK`. The context and details of this message are described below.

To set up your GenView for passing Ink input, set the proper flag in its *GVInkType* instance field. The possible flags are detailed in full in “GenView,” Chapter 9 of the Object Reference Book but are reviewed below for convenience.

`GVIT_PRESSES_ARE_NOT_INK`

Objects running under this GenView never handle Ink input. This flag is set as the default. Your objects will not have to handle `MSG_META_QUERY_IF_PRESS_IS_INK` or `MSG_VIS_QUERY_IF_OBJECT_HANDLES_INK`.

`GVIT_PRESSES_ARE_INK`

Objects running under this GenView always want Ink input. Your objects will not have to handle the message `MSG_META_QUERY_IF_PRESS_IS_INK`, but they will have to handle `MSG_VIS_QUERY_IF_OBJECT_HANDLES_INK`.

GVIT_INK_WITH_STANDARD_OVERRIDE

Objects running under this GenView want Ink, but the user can override it to give mouse events. Your objects will not have to handle MSG_META_QUERY_IF_PRESS_IS_INK, but they will have to handle MSG_VIS_QUERY_IF_OBJECT_HANDLES_INK.

GVIT_QUERY_OUTPUT

The Objects running under this GenView want Ink input, but only under certain circumstances. Your objects will have to handle both MSG_META_QUERY_IF_PRESS_IS_INK and MSG_VIS_QUERY_IF_OBJECT_HANDLES_INK. Note that this option may cause the UI to query across threads, something that could cause performance lags on a busy system. Therefore, whenever possible, you should set one of the other three flags.

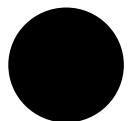
11.4

Depending on how you set up your GenView, you may need to subclass MSG_META_QUERY_IF_PRESS_IS_INK (see above). This message is sent by the Input Manager to see if the object clicked on can handle Ink input. Objects that handle Ink should return IRV_DESIRE_INK. (There is an analogous MSG_META_LARGE_QUERY_IF_PRESS_IS_INK for objects and contents using large documents.)

If the application sends the query message to a Process object acting as a view's content, the Process object should *not* return a value. Instead, it should respond by sending MSG_GEN_APPLICATION_INK_QUERY_REPLY to its GenApplication object; the application object will then pass the appropriate value on to the Input Manager.

Some objects may want both Ink and mouse input; the VisText object is a good example of this—it accepts Ink for handwriting recognition mouse events for cursor movement and text selection. GEOS uses a convention for objects that want both types of input: quick-clicks and click-and-holds in the object are mouse events, but click-and-drags are Ink events.

To use both pen and mouse input, your object should return IRV_INK_WITH_STANDARD_OVERRIDE instead of IRV_DESIRE_INK in its MSG_META_QUERY_IF_PRESS_IS_INK handler. (Or if all objects running in the same GenView want this behavior, you can simply set the GenView's GVIEW_INK_WITH_STANDARD_OVERRIDE flag.)



11.4.2.2 Controlling the Ink

When an object requests Ink input by returning `IRV_DESIRE_INK` or `IRV_INK_WITH_STANDARD_OVERRIDE`, it must also specify the eventual destination of the Ink data block. Additionally, it can set any characteristics in a `GState` for the Ink—clipping area, drawing color, etc.

11.4

These specifics are set in an **InkDestinationInfo** structure set up by your `MSG_META_QUERY_IF_PRESS_IS_INK` handler (through a call to **UserCreateInkDestinationInfo()**); the handler for the routine returns the handle of a block containing the structure. Certain default values and behavior are implemented if you return a null handle or if you set the `GenView` flags to avoid the query (`GVIT_PRESSES_ARE_INK` or `GVIT_INK_WITH_STANDARD_OVERRIDE`).

If you choose not to set any of these specifics, Ink will have the following default behaviors:

- ◆ The destination of the Ink will be the object that would have received the initial click event (typically the implied mouse grab) or the object that fielded the `MSG_META_QUERY_IF_PRESS_IS_INK`.
- ◆ The Ink will not be clipped; the user may draw over the entire screen, overlapping generic objects and other windows. (All the Ink will still be sent to the same destination, however.)
- ◆ The Ink will be drawn in a standard color and brush thickness.

All of the above behavior can be changed in the **InkDestinationInfo** structure. The structure should be created automatically with the routine **UserCreateInkDestinationInfo()**; you can then pass the block handle of this structure blindly to routines that demand an **InkDestinationInfo** structure. Among the four fields of the structure, the following need to be passed to **UserCreateInkDestinationInfo()**:

<i>dest</i>	The <code>optr</code> of the object that will receive the Ink data block after all the Ink has been collected. An object may use this field to force the destination of the Ink to be any particular object.
<i>gs</i>	The handle of a <code>Graphic State (GState)</code> containing clipping, color, or other information about how the Ink should be drawn on the screen. To use the default values, set this to zero. This will allow the user to draw Ink all over the screen. See

“Graphics Environment,” Chapter 23 of the Object Reference Book for full information on GStates.

brushSize

The thickness of the brush used when drawing the Ink on the screen. This is the same as the brush thickness for polylines; see “Drawing Graphics,” Chapter 24 of the Object Reference Book for more information on brush thickness.

callback

Virtual fptr to a callback routine to determine whether a stroke is a gesture or not. This callback routine will be passed to **ProcCallFixedOrMovable()**.

11.4

11.4.2.3 How Ink Is Stored and Passed On

When the user presses the pen to the screen and the input is determined to be Ink, the Input Manager generates a mouse button press event and begins storing up the subsequent mouse events. The UI then determines whether that press event should be treated as Ink or not (the default behavior is to treat them as normal mouse events).

If the events should not be treated as Ink, they are passed on as normal mouse events. If they are to be treated as Ink, they are stored and are *not* passed on. Instead, the UI stores them up in a special data block until the Ink input is stopped. Ink input is stopped when the user removes the pen from the screen for a user-specified amount of time.

After the user finishes entering the Ink input, the Input Manager determines which object will receive the data block (it may not be the original object that received the initial press). It first queries the original object with `MSG_VIS_QUERY_IF_OBJECT_HANDLES_INK`. Any object handling Ink input must subclass this message and make sure that the top Ink point is not above its upper bound; it should then return its upper bound and its `optr`. GEOS provides a routine that does just this; if your object will always handle Ink input, you can set this routine up as the method to be used as follows:

```
@method VisObjectHandlesInkReply, YourClassName,
MSG_VIS_QUERY_IF_OBJECT_HANDLES_INK;
```

If the object handles Ink, the Input Manager sends the entire data block off to the GenSystem object using `MSG_META_NOTIFY_WITH_DATA_BLOCK`

with the identifier `NT_INK`. (For information on this and other notification messages, see “General Change Notification,” Chapter 9.)

11.5 Input Hierarchies

11.5

As previously mentioned in this chapter, the Input Manager channels input events to particular objects. The objects are chosen based on the current input grabs, the position of the pointer, and the state of the application. Sometimes, though, input events must be sent to a particular object not specified by one of these means. One example is keyboard input: While an object can grab the keyboard exclusive, it should not; instead, it should make itself the *focus*. The object that has the focus receives all keyboard input (`MSG_META_KBD_CHAR`).

Similarly, your application or other objects may need to send messages to the object which is currently active. For example, a Text Sizes menu may need to know which of several text objects currently has an active selection. Then, when the user selects a new text size, the menu can send its message to the proper text object. The object that receives these messages is called the *target* object.

The menu in this case, however, should not have to explicitly name the destination of its message. The menu should be able to specify that its message should go to the current target object; the UI knows which object is the target, and it redirects the message automatically.

The GEOS UI uses three special hierarchical mechanisms to fulfill these two purposes (specialized input flow and specification of destinations). These hierarchical mechanisms are built into the basic structure of GEOS objects and are therefore universal and consistent for all applications and all objects. The three hierarchies used are the *focus*, the *target*, and the *model*. (There is a fourth, the controller, but it is used only internally by GEOS.) These hierarchies all function in the same manner but are used for different purposes.

11.5.1 The Three Hierarchies

As stated above, GEOS offers three hierarchies you can use for input flow and specification of message destinations. These hierarchies, with their basic purposes, are listed below.

◆ Focus

The Focus hierarchy designates which object should currently receive keyboard input. Essentially, the focus object has the active keyboard grab. The text objects (VisText and GenText) manage the focus automatically; if you use only these objects for keyboard input, you will not need to understand the focus. The Focus hierarchy is described in full in section 11.5.3 on page 453.

11.5

◆ Target

The Target hierarchy designates the active view, window, and visible selection which may be acted upon. In most cases, the target will be an object the user has selected and now wishes to alter based on menu and dialog selections. For example, a graphic object selected becomes the target object of the application; when the user selects “Rotate” or some other menu function, the menu sends its message to the selected object via the target hierarchy. The Target hierarchy is described in full in section 11.5.4 on page 456.

◆ Model

The Model hierarchy designates the model of the data the user can manipulate. This is provided for highly complex applications that need to extend the Target hierarchy for more specific handling of selections. It extends from the GenSystem through the GenApplication and the document control objects. The Model hierarchy is described in full in section 11.5.5 on page 460.

11.5.2 Common Hierarchy Basics

All three of the input hierarchies function in a similar manner. Each relies on the basic tree structure of UI objects to build a path from the topmost system UI object (GenSystem) down to a leaf node that has the hierarchical properties (e.g. the target object).

Each composite object can have exactly one child with each hierarchical property. The composite determines which of its children currently has the

exclusive of the hierarchy; for example, a composite with ten children may designate the fourth child as the Target node at its level. If that child is a composite with its own children, it will select one of them as the Target node at the next level.

11.5

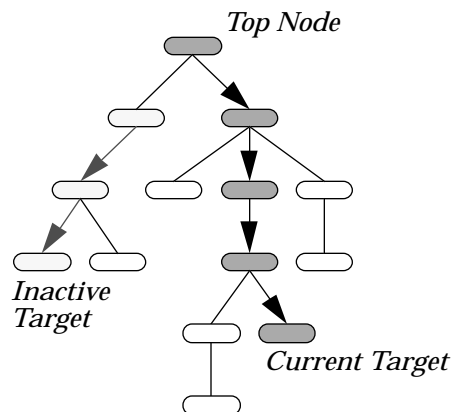


Figure 11-3 A Sample Target Hierarchy

The striped objects represent nodes in the target hierarchy. Each composite designates which of its children has the target exclusive; that child is currently a member of the active target path. Inactive target paths may also be specified; the lightly shaded objects would become the active target path if the top node changed its target child.

Figure 11-3 shows a hierarchical path from the top node of an object tree to the leaf object having the target. A message sent to any object in the path will be passed down the tree until it gets to the active target leaf, the object that actually has the system's target exclusive.

A special feature of the hierarchical structure is that inactive paths may be maintained. Figure 11-3 also shows an inactive path. The inactive path is stored in the objects themselves. The topmost object in the tree, however, has decided that object on the right has the target exclusive; if later the top object decides the object on the left has the exclusive, the target will automatically be given to the leaf node specified without actually changing the object tree (just the pointer stored in the top node).

Note that the other hierarchies operate in a similar manner. The behavior described is not unique to the target hierarchy.

11.5.2.1 Special Terminology

Before reading the in-depth sections on the individual hierarchies, you should know how several terms are defined and used. These terms are listed below and are diagrammed in Figure 11-4.

Node Any composite in the object tree that can be part of a path for one of the hierarchies.

Exclusive The active child of a composite node in the hierarchy is said to have the “exclusive.” Only one child of any composite may have the exclusive at that level. 11.5

Level A group of siblings, all of which can potentially be part of a hierarchical path. All the nodes in a given level will have the same parent. Only one node at each level may have the exclusive for that level.

Grab When an object grabs a hierarchy (e.g. “grabs the focus”), it notifies its parent that it wants to gain that hierarchy’s exclusive for its level. Any other node at the same level will be forced to give up the exclusive.

Active Path

The path of exclusive nodes from the top object to the leaf having the active exclusive. Only objects which have a complete active path to the top node can exhibit the property of the hierarchy (e.g. a text object in a minimized application can not have the focus because its primary window does not have the focus exclusive).

Inactive Path

Any path of exclusive nodes that is broken from the top node. If the topmost node of an inactive path is given the exclusive at its level, the entire inactive path will automatically become the active path.

11.5.2.2 Modifying the Active Path

Each of the three hierarchies has messages understood by **MetaClass** that alter the hierarchy’s active path. The Specific UI takes care of most of the modifications to the active path, however, so simple applications will not need to bother with it.

Any node can “grab” a hierarchical exclusive by sending the grab message to its parent. (Outside agents can alter the path by sending the grab message to the child’s parent on behalf of the child.) The grab forces any other node on the child’s level to give up the exclusive. Any node can release the exclusive with the appropriate message (typically done when forced to release because of another node’s grab).

11.5

Whenever a node gains the exclusive of a hierarchy, it will be notified via a message. This is to notify objects which already had their level’s exclusive that they now have the active hierarchical exclusive. Similarly, if another node grabs the exclusive at any level equal or above an exclusive node, that node will be notified that it has lost the exclusive at its level. See the discussions of the individual hierarchies for more detail on the messages used.

11.5.2.3 Sending Classed Messages

Any object can easily send messages to the object having the active exclusive of a given hierarchy using MSG_META_SEND_CLASSSED_EVENT. This message passes a recorded event (message) to the first object of a specified

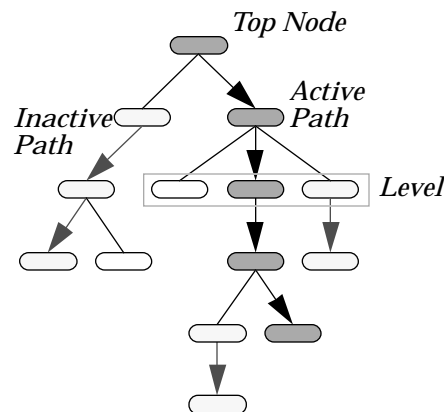


Figure 11-4 *Hierarchy Terminology*

Each object in the tree is a node. Each arrow points to the node having the exclusive at its level. The striped path is the active path; the shaded paths are all inactive paths.

class encountered within the hierarchy. (If the destination should be a leaf, no class should be specified.)

To pass events to the active member of a specific hierarchy, you should use a **TravelOption** with `MSG_META_SEND_CLASSSED_EVENT`. **TravelOption** is a type which describes the destination for the specified event. There are three types of **TravelOption** for each hierarchy:

- ◆ Direct
The “direct” travel options are `TO_FOCUS`, `TO_TARGET`, and `TO_MODEL`. These send the classed event down the hierarchy’s active path, treating the recipient of `MSG_META_SEND_CLASSSED_EVENT` as the active path’s top node. 11.5
- ◆ Application-direct
The “application-direct” travel options are `TO_APP_FOCUS`, `TO_APP_TARGET`, and `TO_APP_MODEL`. These send the classed event first to the `GenApplication` object associated with the recipient of `MSG_META_SEND_CLASSSED_EVENT`; the `GenApplication` object is taken as the top node, and the event is passed down the active path until it is handled.
- ◆ System-direct
The “system-direct” travel options are `TO_SYS_FOCUS`, `TO_SYS_TARGET`, and `TO_SYS_MODEL`. These send the classed event to the `GenSystem` object, the topmost object in the entire UI object tree. The classed event will be passed down the active path, using the `GenSystem` object as the path’s top node.

11.5.3 Using Focus

The specific UI handles most of the manipulation of the focus hierarchy. Most specific UIs interpret keyboard and mouse events and know when to switch the focus from one object to another. For example, in OSF/Motif, clicking on a window will switch the focus to that window from the previous focus window. Before the mouse is clicked, however, the focus exclusive of that window will not change. In other specific UIs which incorporate real-estate cursor behavior, merely moving the cursor over a window will transfer the focus to that window.

By default, an application will come up with certain objects having the focus exclusive within their focus levels. This allows the user to immediately begin typing or operating on data. For an object to grab the default exclusive, it must first be focusable. A focusable object can be any child of a valid focus node. Valid nodes for the focus hierarchy are:

- GenSystem
- GenField
- GenApplication
- GenPrimary
- GenDisplayControl
- GenDisplay
- GenView (with GVA_FOCUSABLE bit set)
- GenInteraction (independently displayable)
- VisContent

Note that a GenView must have the GVA_FOCUSABLE bit set in its *GVI_attrs* field. If an object you need to act as a focus node is not in the above list (for example, a custom gadget), you must subclass the object and add instance fields and message handlers to handle focus functionality.

If an independently displayable objects is focusable, the focus will be given to the object which is visually “on top” of all other objects. For all other objects within a focus exclusive level, the focus will be granted to the object with HINT_DEFAULT_FOCUS in its instance data. If no objects have this hint at a particular focus level, then the focus will be granted to the first focusable child.

11.5.3.1 Grabbing and Releasing the Focus

To grab the focus exclusive in its level, a node should send itself MSG_META_GRAB_FOCUS_EXCL. The default handler for this message grabs the focus exclusive for the object from its parent and forces any other object on the caller’s level to give the focus up. The active exclusive will not be granted unless the caller is part of the active path after the grab.

When another node grabs the focus exclusive, the node currently having the exclusive on that level must give it up. MSG_META_RELEASE_FOCUS_EXCL releases the focus exclusive for the other node.

Note that these messages only modify the exclusive at a single level. Therefore, they only affect the focus optr of the parent's node; further up the focus hierarchy there is no effect. Changing the focus exclusive of an object will only change the direction of the focus path if all parents of the new exclusive are also focus exclusives.

MSG_META_GET_FOCUS_EXCL may be sent to any focusable composite node to get the optr of the node's child having the exclusive. This message may be used even on nodes in the inactive path.

11.5

11.5.3.2 Gaining and Losing the Focus

When an object gains the active focus exclusive and is a node in the active path, it receives MSG_META_GAINED_FOCUS_EXCL. This indicates to the object that it will receive all keyboard input as the active keyboard object. At some point later, when the object has lost the focus exclusive, it will receive MSG_META_LOST_FOCUS_EXCL.

11.5.3.3 Sending Classed Events to the Focus

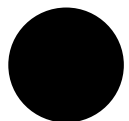
Frequently, you may wish to send messages to objects in the active Focus path. The easiest way to deliver a message to an object in the focus hierarchy is to use MSG_META_SEND_CLASSSED_EVENT with the **TravelOption** TO_APP_FOCUS. To send a message to the leaf node of the active path, send this message with a null class. The message will be sent down the hierarchy until it reaches the leaf object, where it will be processed. (See Code Display 11-2.)

This approach is desirable over using MSG_META_GET_FOCUS_EXCL to return an optr for later use, as the system may have corrupted the optr in the meantime.

Code Display 11-2 Delivering Messages to the Focus

```
@method MyProcessClass, MSG_SEND_MESSAGE_TO_FOCUS_INTERACTION {
    EventHandle event;

    /* The classed event is recorded to be handled by the first object of class
       * GenInteractionClass. */
    event = @record GenInteractionClass::MSG_TEST_MESSAGE();
```



```
/* This message is then sent to the GenApplication object, from which it
 * will travel down the focus hierarchy until handled at the
 * GenInteraction level. */
@call MyApplication::MSG_META_SEND_CLASSSED_EVENT(event, TO_FOCUS);
}

/* You may also declare an object with the TravelOption TO_APP_FOCUS. This will
 * send the message to the application object and then down the focus hierarchy. */

11.5 @object GenTriggerClass MyFocusTrigger = {
    GI_visMoniker = "Send test message to the focus";
    GTI_actionMsg = MSG_TEST_MESSAGE;
    GTI_destination = (TO_APP_FOCUS);
    ATTR_GEN_DESTINATION_CLASS = { (ClassStruc *)&GenInteractionClass };
}

/* To send a message to the focus leaf node, use a null class. */
@method MyProcessClass, MSG_SEND_TEST_TO_FOCUS_LEAF {
    EventHandle event;

    /* A classed event with the class of null is recorded. */
    event = @record null::MSG_TEST_MESSAGE();

    /* This event is sent to the GenApplication, where it will travel down the
     * focus hierarchy to the leaf object, where it will be handled. */
    @call MyApplication::MSG_META_SEND_CLASSSED_EVENT(event, TO_FOCUS);
}

/* As an alternative, an object could be set to deliver its message to the
 * leaf focus node of the application. */
@object GenTriggerClass MyFocusLeafTrigger = {
    GI_visMoniker = "Send test message to the focus leaf object";
    GTI_actionMsg = MSG_TEST_MESSAGE;
    GTI_destination = (TO_APP_FOCUS);
    /* An ATTR_GEN_DESTINATION_CLASS of zero specifies a null class. */
    ATTR_GEN_DESTINATION_CLASS = 0;
}
```

11.5.4 Using Target

The specific UI handles most of the manipulation of the target hierarchy. Most specific UIs interpret keyboard and mouse events and know when to switch the target from one object to another. For example, in OSF/Motif,

clicking on a window will switch the target to that window from the previous target window. Before the mouse is clicked, however, the target exclusive of that window will not change. In other specific UIs which incorporate real-estate cursor behavior, merely moving the cursor over a window will transfer the target to that window.

By default, an application will come up with certain objects having the target exclusive within their target levels. This allows the user to immediately begin typing or operating on data. For an object to grab the default exclusive, it must first be targetable. A targetable object can be any child of a valid target node. Valid nodes for the target hierarchy are:

11.5

- GenSystem
- GenField
- GenApplication
- GenPrimary
- GenDisplayControl
- GenDisplay
- GenView (with GVA_TARGETABLE bit set)
- GenInteraction (independently displayable)
- VisContent

In addition, however, a generic object must be set GA_TARGETABLE in its *GI_attrs* field to become a valid target object. This bit is set by default for the following classes:

- GenField
- GenApplication
- GenPrimary
- GenDisplayControl
- GenDisplay
- GenView

If you wish a generic object to be targetable and it does not appear in this list, set it GA_TARGETABLE. If the object does not by default appear as a valid node in the Target hierarchy (such as for a custom gadget), you must subclass the object and add target instance fields and message handlers.

For all targetable objects within a target exclusive level, the target will be granted to the object with HINT_DEFAULT_TARGET in its instance data. If no object has this hint at a particular target level, then the target will be granted to the first targetable object at that level.

11.5.4.1 Grabbing and Releasing the Target

To grab the target exclusive in its level, a node should send itself `MSG_META_GRAB_TARGET_EXCL`. The default handler for this message grabs the target exclusive for the object from its parent and forces any other object on the caller's level to give the target up. The active exclusive will not be granted unless the caller is part of the active path after the grab.

11.5

When another node grabs the target exclusive, the node currently having the exclusive on that level must give it up. `MSG_META_RELEASE_TARGET_EXCL` releases the target exclusive for the other node.

Note that these messages only modify the exclusive at a single level. Therefore, they only affect the target `optr` of the parent's node; further up the target hierarchy there is no effect. Changing the target exclusive of an object will only change the direction of the target path if all parents of the new exclusive are also target exclusives.

`MSG_META_GET_TARGET_EXCL` may be sent to any targetable composite node to get the `optr` of the node's child having the exclusive. This message may be used even on nodes in the inactive path.

11.5.4.2 Gaining and Losing the Target

When an object gains the active target exclusive and is a node in the active path, it receives `MSG_META_GAINED_TARGET_EXCL`. This indicates to the object that it will receive all keyboard input as the active keyboard object. At some point later, when the object has lost the target exclusive, it will receive `MSG_META_LOST_TARGET_EXCL`.

11.5.4.3 Sending Classed Events to the Target

Frequently, you may wish to send messages to objects in the active Target path. The easiest way to deliver a message to an object in the target hierarchy is to use `MSG_META_SEND_CLASSSED_EVENT` with the **TravelOption** `TO_TARGET`. To send a message to the leaf node of the active path, send this message with a null class. The message will be sent down the hierarchy until it reaches the leaf object, where it will be processed. (See Code Display 11-3.)

This approach is desirable over using MSG_META_GET_TARGET_EXCL to return an optr for later use, as the system may have corrupted the optr in the meantime.

Code Display 11-3 Delivering Messages to the Target

```
@method MyProcessClass, MSG_SEND_MESSAGE_TO_TARGET_INTERACTION {
    EventHandle event;

    /* The classed event is recorded to be handled by the first object of class
     * GenInteractionClass. */
    event = @record GenTextClass::MSG_TEST_MESSAGE();

    /* This message is then sent to the GenApplication object, from which it
     * will travel down the target hierarchy until handled at the
     * GenInteraction level. */
    @call MyApplication::MSG_META_SEND_CLASSSED_EVENT(event, TO_TARGET);
}

/* You may also declare an object with the TravelOption TO_APP_TARGET. This will
 * send the message to the application object and then down the target hierarchy.
 */

@object GenTriggerClass MyTargetTrigger = {
    GI_visMoniker = "Send test message to the target";
    GTI_actionMsg = MSG_TEST_MESSAGE;
    GTI_destination = (TO_APP_TARGET);
    ATTR_GEN_DESTINATION_CLASS = { (ClassStruc *)&GenTextClass };
}

/* To send a message to the target leaf node, use a null class. */

@method MyProcessClass, MSG_SEND_TEST_TO_TARGET_LEAF {
    EventHandle event;

    /* A classed event with the class of null is recorded. */
    event = @record null::MSG_TEST_MESSAGE();

    /* This event is sent to the GenApplication, where it will travel down the
     * target hierarchy to the leaf object, where it will be handled. */
    @call MyApplication::MSG_META_SEND_CLASSSED_EVENT(event, TO_TARGET);
}

/* As an alternative, an object could be set to deliver its message to the
 * leaf target node of the application. */
@object GenTriggerClass MyTargetLeafTrigger = {
    GI_visMoniker = "Send test message to the target leaf object";
```

11.5



```
GTI_actionMsg = MSG_TEST_MESSAGE;
GTI_destination = (TO_APP_TARGET);
/* An ATTR_GEN_DESTINATION_CLASS of zero specifies a null class. */
ATTR_GEN_DESTINATION_CLASS = 0;
}
```

11.5.5 Using Model

In some cases, you may need to send or receive information from some object other than the focus or target. The Model hierarchy is provided as an alternative. In most cases, the model provides a means of pointing to data selections other than that pointed to by the Target hierarchy. Only use it if the target hierarchy cannot serve your needs.

The model of an application is used to identify objects that represent the actual data that can be selected by the user. Typically, these selections are either secondary selections or are not visible. The selections are frequently operated on with the use of menus, dialog boxes, or other UI gadgetry. Only one object at each model level may have the model “exclusive,” and for an object to “have the Model” of an application, that object must have a continuous path of model exclusives up to the application object.

Only GenSystem, GenApplication, and the document control objects can be directly used in the Model hierarchy without any modifications. To use other objects in the Model hierarchy, you must first subclass those objects’ classes and add an instance field and several message handlers pertaining to the Model hierarchy. Any object which is a valid target node can be subclassed in this manner.

By default, an application will come up with certain objects having the model exclusive within their model levels. This allows the user to immediately begin operating on data. For an object to grab the default exclusive, it must first be modelable. Only objects which are modelable may have the model exclusive. A modelable object can be any child of a valid model node. Valid nodes for the Model hierarchy are the same as for the Target hierarchy. However, a document control object must also be set GDCA_MODELABLE to grab the model exclusive. Remember that just because an object is a valid

node does not mean it can exhibit the behavior of the Model hierarchy; you must subclass most objects for this to occur.

For objects within a model level, the model will be granted first to any object with `HINT_DEFAULT_MODEL` in its instance data. If no object has this hint at a particular model level, then the model will be granted to the first modelable object at that level.

The easiest way to deliver a message to an object in the active Model hierarchy is to use `MSG_META_SEND_CLASSSED_EVENT` with the **TravelOption** `TO_MODEL`. To send a message to the leaf node in the Model hierarchy, send this message with a class of null. The message will be sent down the hierarchy until it reaches the leaf object, where it will be processed.

11.5

This approach is desirable over using `MSG_META_GET_MODEL_EXCL` to return an `optr` for later use as the system may have corrupted the `optr` in the meantime.

11.5.5.1 Changing the Model Exclusive

To modify the Model hierarchy, use `MSG_META_GRAB_MODEL_EXCL`. To remove the model exclusive from an object without setting it to another object, use `MSG_META_RELEASE_MODEL_EXCL`. Note that these messages only modify the model exclusive of a single object. Therefore, they only affect the model pointer of the parent; further up the Model hierarchy there is no effect. Changing the model exclusive of an object will only change the direction of the model path if all ancestors of the new model exclusive are also model exclusives.

To check which model exclusive a composite points to, send it a `MSG_META_GET_MODEL_EXCL`. This message will return the `optr` of the model child. The returned object may not have the model properties, though, for the reasons above. Never send a message directly to the object returned by this method as it may have been changed in the meantime. Use `MSG_META_SEND_CLASSSED_EVENT` with the **TravelOption** `TO_MODEL` instead.

11.5.5.2 Intercepting the Model

Your application may also wish to be notified when an object either gains or loses its model properties. Whenever an object in GEOS gains the model of the application—not just the model exclusive—it sends a `MSG_META_GAINED_MODEL_EXCL` to that object. Similarly, if an object loses the model of an application—not just the model exclusive—the system sends a `MSG_META_LOST_MODEL_EXCL` to that object.

11.5

If your application needs to know when an object either gains or loses its model properties, intercept `MSG_META_GAINED_MODEL_EXCL` and `MSG_META_LOST_MODEL_EXCL`. Be sure to pass these messages on to the superclass, however, to perform necessary default functionality. Within your message handlers for these messages, you may add whatever additional behavior you require.

11.5.6 Extending the Hierarchies

You may also extend or modify the hierarchies to add other objects that are not active nodes by default. If custom objects wish to use the hierarchies, you will need to add instance data and message handlers for their classes. The following steps must be taken to add a new node in a hierarchy:

- 1 Add a hierarchical instance field of type **HierarchicalGrab**.
- 2 Add a handler for `MSG_META_MUP_ALTER_FTVMC_EXCL`. This method alters the custom instance field for the grab and release messages for all hierarchies. You must call the superclass in your handler.
- 3 Add methods to handle the `GAINED` and `LOST` messages pertaining to the particular hierarchy you are extending. These methods should call **FlowUpdateHierarchicalGrab()**. You can also add behavior exhibited when the object gains or loses the exclusive. Depending on what you are subclassing, your default behavior may already take care of this. For instance subclasses of **VisClass** automatically handle losing and gaining the target exclusive.
- 4 Add a method to handle the `GET` message for the particular hierarchy you are extending. This method returns the `optr` of the object set in the new instance field. The method should return *true* to signal that it is capable of gaining the exclusive.

- 5 Add the capability to handle the MSG_META_SEND_CLASSSED_EVENT with the **TravelOption** of the particular hierarchy you are extending. The way you handle this message depends on the hierarchy affected, as discussed below.

TO_FOCUS and TO_TARGET requests should only travel down hierarchies of the same name. If no further travel is possible, the method should pass the event on to the superclass for handling. TO_MODEL requests should only travel down the Model hierarchy if it exists and the next optr is non-null.

11.5

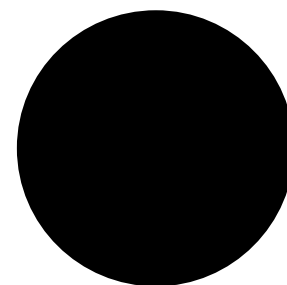


Input

464

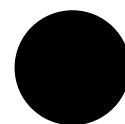
11.5

Managing UI Geometry

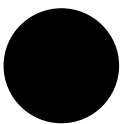


12

12.1	Geometry Manager Overview	467
12.1.1	Geometry Manager Features	468
12.1.2	How Geometry Is Managed	469
12.2	Arranging Your Generic Objects	472
12.2.1	General Geometry Rules	472
12.2.1.1	Generic Tree Structure	473
12.2.1.2	How Hints Work.....	473
12.2.2	Orienting Children	474
12.2.3	Justifying and Centering Children.....	478
12.2.3.1	Edge Justification	478
12.2.3.2	Edge Alignment with Other Objects.....	479
12.2.3.3	Centering Children.....	479
12.2.3.4	Full Justification.....	479
12.2.4	Sizing Objects	480
12.2.4.1	Sizing the Parent	481
12.2.4.2	Sizing the Children.....	481
12.2.4.3	Maximum, Minimum, Fixed, and Initial Sizes	485
12.2.5	Outlining the Composite	487
12.2.6	Using Monikers.....	487
12.2.6.1	Placing Monikers in Relation to Children	487
12.2.6.2	Justifying or Centering on Monikers.....	489
12.2.6.3	Removing Moniker Borders.....	489
12.2.7	Using Custom Child Spacing.....	490
12.2.8	Allowing Children to Wrap.....	492
12.2.9	Object Placement.....	493
12.2.9.1	Reply Bars	493
12.2.9.2	Placing Objects in Menu Bars.....	496
12.2.9.3	Placing Objects in Scroller Areas.....	496
12.2.9.4	Placing Objects in Window Title Bars	496
12.2.9.5	Placement of Objects Popped Up	497
12.3	Positioning and Sizing Windows	497



12.3.1	Window Positioning.....	499
12.3.2	Determining Initial Size.....	500
12.3.3	On-Screen Behavior	501
12.3.4	Window Look and Feel.....	502



One of the more difficult aspects of programming a graphically oriented application is positioning the user interface components on the screen. Developers often have to manually place buttons, size dialog boxes, and draw frames and titles. The GEOS Geometry Manager lets you ignore most of these issues.

The geometry manager allows you to ignore position, size, and spacing of your generic UI components; if you use visual objects, the geometry manager can calculate object sizes and positions, dynamically changing them as necessary. Because the geometry manager works at run-time, you have to do very little in the way of screen management. Instead, you can set various attributes and hints to fine-tune your UI's appearance.

12.1

Before reading this chapter, you should be familiar with the GEOS User Interface in general and the generic UI in specific. For most of this chapter, "First Steps: Hello World," Chapter 4, will be sufficient background information; for other sections, you should at least read "The GEOS User Interface," Chapter 10.

12.1 Geometry Manager Overview

The GEOS geometry manager is an algorithmic mechanism which interacts directly with the UI to position and size objects of all types. The geometry manager automatically manages all generic objects through interactions with the specific UI library; you can also use it to manage your visible object trees.

The Geometry Manager is what makes the generic UI possible: Since geometry of UI objects (size and position) is determined by the UI at run-time, generic UI objects do not have to be explicitly defined with certain geometry. With this feature, a specific UI library can set up the UI geometry as it wants.

Your first interaction with the geometry manager will be when you build a generic object tree for your UI. Most hints applied to an object are signals to the geometry manager about how the object should be positioned, sized, or otherwise placed on the screen. These hints are all defined in **GenClass**, the

topmost generic object class, so they are available (though not necessarily useful) to all types of generic objects.

This chapter first describes the features of the geometry manager and how the mechanism fits into the GEOS system as a whole (later in this section). Then it discusses how to manage generic UI objects (using a dialog box as an example) in section 12.2 on page 472. For the hints you can apply to window objects, see section 12.3 on page 497. For descriptions of managing the geometry of visible object trees and composites, see “VisClass,” Chapter 23 of the Object Reference Book.

12.1.1 Geometry Manager Features

The geometry manager, used effectively, can take nearly all the work out of displaying, sizing, and positioning objects on the screen. Many of the tasks you normally would have to do to create even a simple dialog box are taken care of automatically by the geometry manager and the specific UI library. Among the functions the geometry manager performs are the following:

- ◆ **Window Positioning and Sizing**
The geometry manager interacts directly with the UI to determine where windows should be placed and what their dimensions should be. You have some control over this with hints, messages, and object instance data, and often you will have to choose a hint to determine the window position or size.
- ◆ **Object Sizing**
The size of an object can be determined by the combined sizes of its children, by the limits of its parent, or specifically by other constraints on the object. The geometry manager allows all three types of sizing behavior. The first two are implemented automatically in geometry updates; the third depends on how the application wants to manage the “other constraints.”
- ◆ **Object Positioning**
The position of an object can be determined either by the object or by the geometry manager. The geometry manager can position objects based on several criteria including the spacing of siblings in a composite, the size and/or location of the parent composite, and the justification hints applied to the object. Visible objects can control very closely how and

where they are positioned; generic objects can determine how positioning occurs with various hints.

◆ **Automatic Geometry Updating**

Typically, when a particular object in a tree is resized, repositioned, added, or removed, the object mechanisms will mark the geometry of both that object and its parent invalid. This will invoke the geometry manager, which will traverse the object tree, determine which objects are affected by the potential geometry change, and then update the size and position of all affected objects up to the top of the tree. This entire process is automatic for generic objects; for visual objects, you may or may not have to mark the object invalid yourself, depending on the depth of custom geometry control you use.

12.1

With geometry functions covered by the geometry manager and drawing functions covered by the specific UI library, you have little else to do but specify the structure of your object tree and the hints each object should have. The system does all the dirty work of drawing and managing the objects on the screen, leaving you free to work on your application rather than on its graphical screen representation.

12.1.2 How Geometry Is Managed

The geometry manager is one part of the system that draws objects on the screen and redraws them when necessary. This system is called the “visual update mechanism.”

The visual update mechanism is invoked any time the geometry or image of an object is marked invalid. If only the image is invalid, typically no geometry calculations will be done. If, however, the geometry of an object is marked invalid, the geometry manager is called in to calculate the new size and position of all the affected objects in the object tree.

Normally, you won't have to do any more than set certain hints (for generic objects) or attribute flags (for visible objects) to get the desired geometry behavior. If, however, you plan on modifying how the geometry of a particular set of objects is set, you will want to understand the sequence of events that happen during a geometry update. Typically, only programmers who are building their own specific UI libraries and those who have complex visual object trees will need to understand the geometry calculation sequence.

The geometry manager's calculation can be thought of as a multiple-pass, recursive algorithm. It traverses the object tree, managing geometry within each composite until it reaches the top-most object that is unaffected by the geometry calculations.

The geometry update process is invoked when an object has its geometry marked invalid. This is determined by the object's visual flags (its *VI_optFlags* field), `VOF_GEOMETRY_INVALID` and `VOF_WINDOW_INVALID`. If either of these flags is set, the object's geometry requires recalculation.

The geometry manager sends `MSG_VIS_UPDATE_GEOMETRY` to the visible object with invalid geometry. This message determines which objects in the tree are potentially affected by the geometry update, and then it recalculates the tree's geometry.

If the object initially marked invalid is a composite, all its children are automatically included in the update. If the object is a child of a composite, the geometry manager will travel up the tree, marking any parents and siblings that also require recalculation. (Recalculation does not cross `VTF_IS_WIN_GROUP` boundaries, however. A `VTF_IS_WIN_GROUP` is any windowed object; geometry calculations generally do not cross windowed objects (move into another window layer) unless explicitly instructed to do so.)

The geometry manager then returns to the bottom-most affected object in the tree and sends it a `MSG_VIS_RECALC_SIZE`. This message passes the object a suggested size, which will be the same as the object's original size. (The original size is passed as an optimization in case the object does not need its geometry changed; many changes will affect many objects but will actually change only a few.)

The object then returns its new desired size based on the geometry change. If this is the original object marked invalid, the returned size will be the direct result of the geometry change. If this object is a descendent of the invalid object, it may not want to change its geometry at all and may therefore return its current size.

The geometry manager collects the returned sizes of all siblings at a given level and calculates the suggested size of their parent composite. The parent composite is then sent `MSG_VIS_RECALC_SIZE` with the newly calculated size and returns the size it wants to be as a result. (Some composites will

have fixed, maximum, or minimum sizes or may have their sizing tied to other factors; these objects may return a different size.) If the returned size is different from the passed size, the geometry manager arbitrates between the objects. Typically, the composite will be grown large enough (even if fixed or maximum size) to fit all its children.

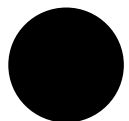
The geometry manager will go up the tree in this manner, calculating the new geometry of each child and then each composite until it reaches an object that is unaffected by the geometry change. (For example, if a child shrinks but other children cause the composite to remain the same size, the geometry update will stop at the composite because objects higher up are not affected.) If all objects in the tree are affected by the change, the geometry update will stop at the topmost object in the branch (the first `VTF_IS_WIN_GROUP` object).

12.1

If any parent objects disagree with their children on the new size, the geometry manager will arbitrate. The geometry manager may make several passes to suggest width and height changes to both the child and parent before deciding on an ultimate geometry. The geometry manager may alter child spacing, wrap the children, grow the composite, or even clip the children if necessary. A result will quickly be reached, however—the geometry manager will not thrash between the parent and children (unless you subclass `MSG_VIS_RECALC_SIZE` or `MSG_VIS_GET_CENTER` and do not pay close attention to how the message is handled).

After the geometry is recalculated up to the first `VTF_IS_WIN_GROUP` object (or up to the top affected object), a `MSG_VIS_NOTIFY_GEOMETRY_VALID` will be sent to each of the invalid objects that have `VGA_NOTIFY_GEOMETRY_VALID` set in their `VI_attrs` field. A visual update will then be started to draw the new geometry to the screen. Once a branch's geometry has been calculated, the branch will remain unaffected until the next time the geometry of one of its objects is marked invalid.

If a recalculation of a branch's geometry causes a `VTF_IS_WIN_GROUP` object to change its size or position, the window system will determine the effects. For example, if a `VisContent` object grows as a result of a geometry change, it will send a `MSG_GEN_VIEW_SET_DOC_BOUNDS` to the `GenView`; the geometry recalculation mechanism will not scroll the content-view connection. What happens to the view and its parent window then depends on the attributes of the view. For example, if the view is set to follow the size



of its content, it will grow, bumping the size of its parent in the process. If the view is scrollable, it likely will stay the same size.

12.2 Arranging Your Generic Objects

12.2

The process of arranging user interface elements for a GEOS application differs drastically from that of traditional user interface programming methods. In most graphical systems, you must manually determine the location of each and every user interface element. For instance, to design a dialog box, you have to figure the location of each text string, button, and list entry. Normally, these items would have to be hard-coded into your application, and often different video resolutions would have to be taken into account.

Not so in GEOS. The generic and specific UI libraries work with the geometry manager to automatically arrange your generic UI objects in the most appropriate way. To describe a dialog box in GEOS, for instance, you simply have to define the objects contained in the dialog box and their relative positions in the generic object tree; GEOS does the rest, determining the position and visual effects of each object. You can even fine-tune the appearance of each object with the use of hints.

12.2.1 General Geometry Rules

You may feel at first that the use of generic UI components limits your control over the UI of your application. Actually, just the opposite is true: Your application becomes immediately available at all supported video resolutions in all available specific UIs. Additionally, you can concentrate on more important issues of your application; the generic UI objects take care of a tremendous amount of the work in positioning and displaying the user interface.

To determine your generic UI tree, you must remember that UI geometry is determined by three things: First and foremost, the overall structure of the

UI is governed by the structure and organization of the generic object tree. For example, if GenTriggers A, B, and C are designated as children of a dialog box, they will likely be ordered in the same order they're listed in the dialog's *GI_comp* field.

Second, an object's attributes can determine its implementation. For example, the *GIGI_visibility* field of a GenInteraction can determine whether the interaction is implemented as a dialog box, a menu, or a grouping object. The attributes of a generic object determine the basic functionality of the object; that is, even though a menu might be implemented differently in different specific UIs, the functionality of a menu will be implemented in the object. The manner of display of an object is based on its functionality. In this way, the attributes of a generic object take precedence over hints.

12.2

Third, an object's behavior can be fine-tuned through the use of hints. Hints may or may not be implemented for each object that has them, and they control less strictly how the object works.

12.2.1.1 Generic Tree Structure

As stated above, the structure of your application's generic object tree determines how the generic UI objects will be organized. A simple example of a generic tree including a menu, a dialog box, and a GenView can be found in "First Steps: Hello World," Chapter 4.

12.2.1.2 How Hints Work

Nearly all geometry of generic UI objects is determined by hints. You can position, size, and limit generic objects with different hints. All these hints are defined in **GenClass**; inheritance allows all generic objects to have them, though not all hints are applicable to all generic objects.

Hints are described in detail in "GenClass," Chapter 2 of the Object Reference Book, but the basics are reviewed here for convenience. Hints are implemented as variable data entries; each hint is a different variable-data type. Hints therefore take up instance data space in an object only when the object has that hint.

As stated earlier, not all hints are appropriate for all objects. For example, `HINT_CUSTOM_CHILD_SPACING` is probably not useful when applied to a

GenTrigger object because a GenTrigger can have no children. Hints, by definition, are also not guaranteed to be supported by all specific UI libraries. For example, a specific UI might require all elements of a dialog box to be oriented horizontally; in this case, `HINT_ORIENT_CHILDREN_VERTICALLY` might be useful but might not be heeded by the specific UI.

Because geometry management of generic objects can be confusing at times, this chapter follows a series of examples accompanied with diagrams. For the most part, you should be able to “plug and play” the examples in the following sections.

12.2.2 Orienting Children

```
HINT_ORIENT_CHILDREN_HORIZONTALLY,  
HINT_ORIENT_CHILDREN_VERTICALLY,  
HINT_ORIENT_CHILDREN_ALONG_LARGER_DIMENSION,  
HINT_SAME_ORIENTATION_AS_PARENT
```

The two hints `HINT_ORIENT_CHILDREN_HORIZONTALLY` and `HINT_ORIENT_CHILDREN_VERTICALLY`, when used in a composite object, determine the orientation of the composite’s children. A composite object can orient its children either horizontally or vertically. The geometry manager will determine the combined size of the children and any margins or spacing set up (none by default) and will size the composite to the minimum required by its children.

Children oriented horizontally will typically be top-justified, and children oriented vertically will be left-justified. (This is default behavior changeable either with the use of hints or by the specific UI.) The vertical composite will

size to the width of the widest child; the horizontal composite will size to the height of the tallest child.

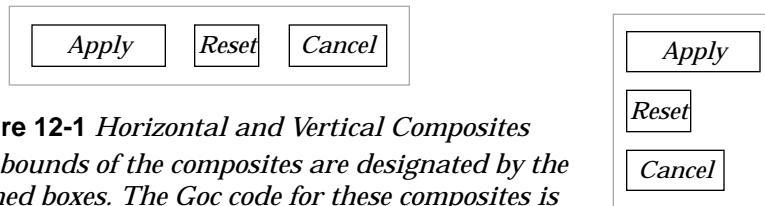


Figure 12-1 *Horizontal and Vertical Composites*
The bounds of the composites are designated by the dashed boxes. The Goc code for these composites is shown in the following display.

12.2

Code Display 12-1 Orienting a Composite

```
/* Composite oriented horizontally */
@object GenInteractionClass MyInteraction = {
    GI_comp = @ApplyTrigger, @ResetTrigger, @CancelTrigger;
    HINT_ORIENT_CHILDREN_HORIZONTALLY;
}

/* Composite oriented vertically */
@object GenInteractionClass MyOtherInteraction = {
    GI_comp = @ApplyTrigger, @ResetTrigger, @CancelTrigger;
    HINT_ORIENT_CHILDREN_VERTICALLY;
}
```

A large example of a complex dialog box is shown in Figure 12-2. This dialog box uses nested GenInteraction objects to achieve a complex grouping of children. Using the many justification, spacing, and orientation hints described throughout the chapter, you can change and fine-tune the appearance of the dialog box. The code that implements this dialog box is shown in Code Display 12-2.

Two other hints may be useful for arranging children, especially in toolboxes. These two hints are not shown in the examples, though they are used prominently with tool groups.

HINT_SAME_ORIENTATION_AS_PARENT is used by one windowed object (GenInteraction) when it should orient its children in the same way its parent does. This, too, is useful for toolboxes because you can set an

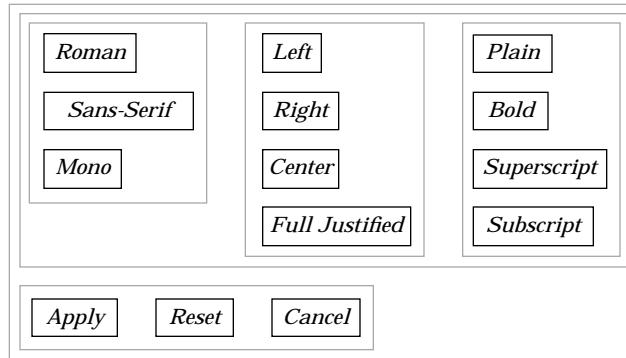


Figure 12-2 A Complex Dialog Box

GenInteraction objects may be nested, and they may be oriented just as other children. The code for this example is shown in the following display.

orientation in the toolbox, then set `HINT_SAME_ORIENTATION_AS_PARENT` in the tool group.

`HINT_ORIENT_CHILDREN_ALONG_LARGER_DIMENSION` allows you to orient a composite's children along the screen's larger dimension. The hint will work like `HINT_ORIENT_CHILDREN_HORIZONTALLY` if the screen is wider than it is tall; it will work like `HINT_ORIENT_CHILDREN_VERTICALLY` if the screen is taller than it is wide.

Code Display 12-2 A Complex Dialog Box

```
/* This code display shows the basic Goc code that will get the configuration
 * shown in Figure 12-2 on the screen. Other attributes are left out of the
 * definitions of the objects. */

/* The topmost interaction. It groups the two large interactions vertically. */
@object GenInteractionClass TopInteraction = {
    GI_comp = @ParaInteraction, @ReplyInteraction;
    HINT_ORIENT_CHILDREN_VERTICALLY;
}
```

```

/* The top of the two large interactions. It groups the three smaller interactions
 * on the top, horizontally. */
@object GenInteractionClass ParaInteraction = {
    GI_comp = @FontInteraction, @JustInteraction, @StyleInteraction;
    HINT_ORIENT_CHILDREN_HORIZONTALLY;
}

/* The interaction containing the Apply, Reset, and Cancel triggers. The
 * triggers are oriented horizontally. */
@object GenInteractionClass ReplyInteraction = {
    GI_comp = @ApplyTrigger, @ResetTrigger, @CancelTrigger;
    HINT_ORIENT_CHILDREN_HORIZONTALLY;
}

/* The three vertically-oriented interactions at the top are similar. Each
 * contains several triggers. All three are defined below. */
@object GenInteractionClass FontInteraction = {
    GI_comp = @RomanTrigger, @SansTrigger, @MonoTrigger;
    HINT_ORIENT_CHILDREN_VERTICALLY;
}

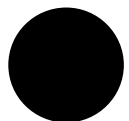
@object GenInteractionClass JustInteraction = {
    GI_comp = @LeftTrigger, @RightTrigger, @CenterTrigger, @FullTrigger;
    HINT_ORIENT_CHILDREN_VERTICALLY;
}

@object GenInteractionClass StyleInteraction = {
    GI_comp = @PlainTrigger, @BoldTrigger, @SuperTrigger, @SubTrigger;
    HINT_ORIENT_CHILDREN_VERTICALLY;
}

/* Triggers. All the triggers are essentially the same. Each is named as it
 * appears in the GI_comp fields above, and each has the corresponding
 * GI_visMoniker field. The RomanTrigger object is shown as an example. */
@object GenTriggerClass RomanTrigger = {
    GI_visMoniker = "Roman";
}

```

12.2



12.2.3 Justifying and Centering Children

Although child justification defaults to whatever the specific UI normally uses, you can set the justification of a composite's children with several different hints.

Typically, horizontal composites will have their children top-justified and vertical composites will have left-justified children (as shown in the previous examples).

12.2

12.2.3.1 Edge Justification

`HINT_TOP_JUSTIFY_CHILDREN`, `HINT_BOTTOM_JUSTIFY_CHILDREN`,
`HINT_LEFT_JUSTIFY_CHILDREN`, `HINT_RIGHT_JUSTIFY_CHILDREN`

If you want to line all of a composite's children up on one of their edges, you should use one of these hints. They are self-explanatory; the edge specified in the name of the hint (top, left, etc.) is the edge of the composite along which all the children will appear.

Add the justification hints along with an orientation hint. Examples of justification are shown in Figure 12-3.

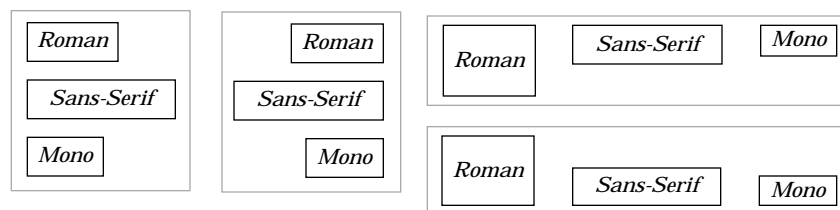


Figure 12-3 *Examples of Edge Justification*

From left to right (clockwise): Left justification, right justification, top justification, bottom justification.

12.2.3.2 Edge Alignment with Other Objects

```
HINT_ALIGN_LEFT_EDGE_WITH_OBJECT,
HINT_ALIGN_TOP_EDGE_WITH_OBJECT,
HINT_ALIGN_RIGHT_EDGE_WITH_OBJECT,
HINT_ALIGN_BOTTOM_EDGE_WITH_OBJECT
```

These hints align an object's respective edge with the same edge of another object; the `optr` of the other object is the hint's argument. The other object does not necessarily have to be a direct sibling or parent, but the result must not cause the object to stray outside its parent's bounds.

12.2

12.2.3.3 Centering Children

```
HINT_CENTER_CHILDREN_VERTICALLY,
HINT_CENTER_CHILDREN_HORIZONTALLY
```

Often, rather than justifying a composite's children, you will want to center them. These two hints allow you to center the children as shown in Figure 12-4.

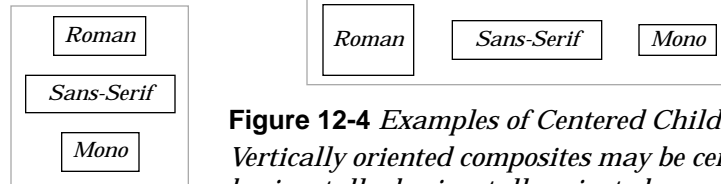
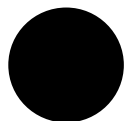


Figure 12-4 *Examples of Centered Children*
Vertically oriented composites may be centered horizontally; horizontally-oriented composites may be centered vertically.

12.2.3.4 Full Justification

```
HINT_FULL_JUSTIFY_CHILDREN_HORIZONTALLY,
HINT_FULL_JUSTIFY_CHILDREN_VERTICALLY,
HINT_DONT_FULL_JUSTIFY_CHILDREN,
```



```
HINT_INCLUDE_ENDS_IN_CHILD_SPACING,  
HINT_DONT_INCLUDE_ENDS_IN_CHILD_SPACING
```

12.2

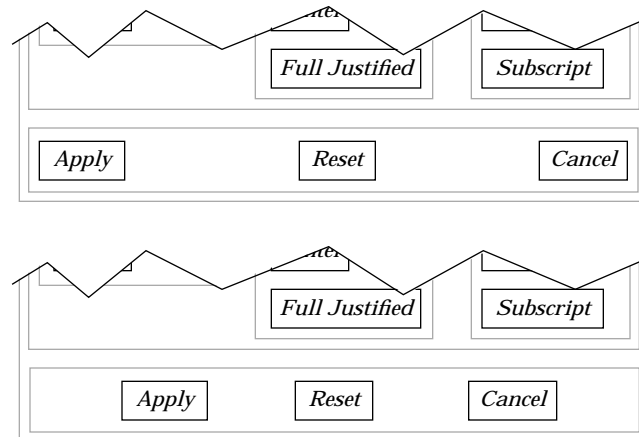
Besides justifying the composite's children either on a single edge or on the center line of the composite, you might want to full-justify the children. To do this, you will want to give the composite an orientation first (either vertical or horizontal). The composite may also require a special sizing hint to expand its bounds—see `HINT_EXPAND_WIDTH_TO_FIT_PARENT` and the other sizing hints in section 12.2.4 on page 480.

If the full justification hint is in the same dimension as the orientation of the composite (e.g., `HINT_FULL_JUSTIFY_CHILDREN_HORIZONTALLY` and `HINT_ORIENT_CHILDREN_HORIZONTALLY`), then the geometry manager will divide up the entire space of the composite and spread the children out equally along it. If you also specify `HINT_INCLUDE_ENDS_IN_CHILD_SPACING`, the geometry manager will calculate the spacing as if there were one more child than there actually is; it will then put half the extra space on either end of the children. `HINT_DONT_INCLUDE_ENDS_IN_CHILD_SPACING` performs the default calculations. Examples of various full justification are shown in Figure 12-5. See Code Display 12-3 on page 483 for the code required to implement this full justification behavior.

Typically, full justification is not implemented by default. Some specific UIs may try to full-justify children always; if you want the children non-justified (when possible), apply `HINT_DONT_FULL_JUSTIFY_CHILDREN`. Added spacing is also typically not turned on by default; you can ensure that added spacing will not be included (where possible) by applying the hint `HINT_DONT_INCLUDE_ENDS_IN_CHILD_SPACING` to your composite.

12.2.4 Sizing Objects

Sizing can occur in essentially one of two ways: First, the composite can size itself based on its children. Second, the children can size themselves based on their parent. You can specify additional sizing restrictions in the form of minimum, maximum, and fixed sizes.



12.2

Figure 12-5 *Full-Justification of Children*

The reply bar interaction object is specified as horizontally full-justified. In the upper example, it does not include ends in child spacing; in the lower example, it does. Unchanged portions of the dialog box are cut out.

12.2.4.1 Sizing the Parent

```
HINT_NO_TALLER_THAN_CHILDREN_REQUIRE ,
HINT_NO_WIDER_THAN_CHILDREN_REQUIRE
```

A composite can size itself to be only as large as its children require. This may be applied to your primary window or to other organizational composites to keep them from growing into any extra space in their parents. You can set the width and height restrictions independently; to set them both, use both hints in the composite.

12.2.4.2 Sizing the Children

```
HINT_EXPAND_HEIGHT_TO_FIT_PARENT ,
HINT_EXPAND_WIDTH_TO_FIT_PARENT ,
HINT_DIVIDE_WIDTH_EQUALLY , HINT_DIVIDE_HEIGHT_EQUALLY
```

Often a composite will have a set size and its children will want to expand themselves to take up all the available space. For example, the interaction



with the Apply, Reset, and Cancel triggers in the dialog box shown in

12.2

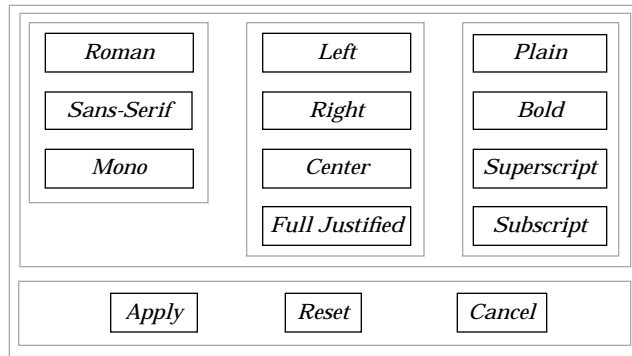


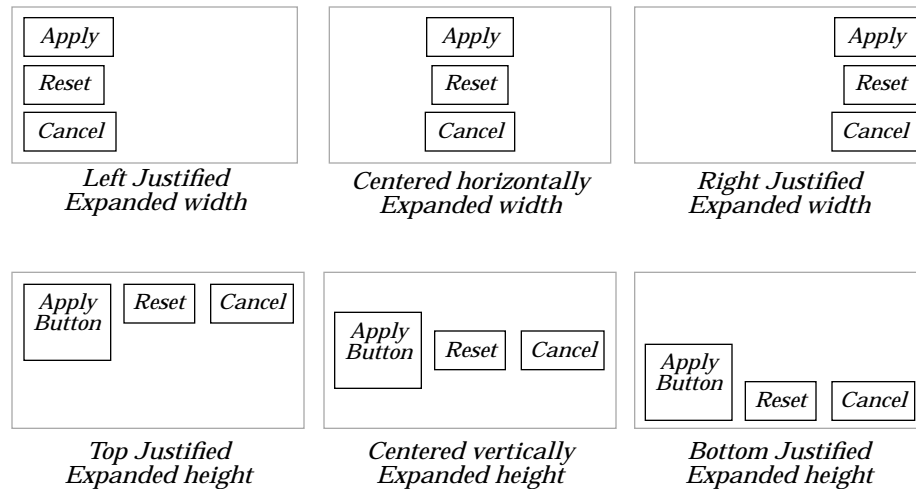
Figure 12-6 *Expanding Children to Fit Parents*

All the triggers in the vertical composites are expanded to fit their parents' widths. The interaction at the bottom is as well in order to allow the triggers to be full-justified with space added.

Figure 12-6 must have `HINT_EXPAND_WIDTH_TO_FIT_PARENT` set for it to expand itself to the far right edge of the dialog box. Another potential situation is a `GenView` within a `GenPrimary`; when the user resizes the primary, the view should probably expand itself to take up any extra space in the window.

A third example would be a series of triggers that want to be the same width in a vertical composite. This appears in many menus; each of the triggers expands itself to the width of the parent; the parent composite sizes its width to the largest of the triggers. An example taken from the complex dialog box is shown in Figure 12-6. Code for this example is shown in Code Display 12-3.

Some more complex examples of expanding a group and justifying the elements are shown in Figure 12-7. All of the examples use a justification hint combined with an expand-to-fit hint.



12.2

Figure 12-7 *Sample Justifications*

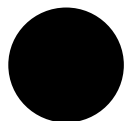
Any geometry combination can be obtained with various geometry hints and levels of organizational GenInteractions. The above six combinations apply both the expansion and the justification hints to the composite object.

Code Display 12-3 Using Full Justification of Children

```
/* This code display shows the basic Goc code that will get the configuration
 * shown in Figure 12-6 on the screen. Other attributes are left out of the
 * definitions of the objects. This code display is supplemental to
 * Code Display 12-2; most comments have been removed. */

@object GenInteractionClass TopInteraction = {
    GI_comp = @ParaInteraction, @ReplyInteraction;
    HINT_ORIENT_CHILDREN_VERTICALLY;
}

@object GenInteractionClass ParaInteraction = {
    GI_comp = @FontInteraction, @JustInteraction, @StyleInteraction;
    HINT_ORIENT_CHILDREN_HORIZONTALLY;
}
```



```
12.2 @object GenInteractionClass ReplyInteraction = {
    GI_comp = @ApplyTrigger, @ResetTrigger, @CancelTrigger;
    HINT_ORIENT_CHILDREN_HORIZONTALLY;
    HINT_EXPAND_WIDTH_TO_FIT_PARENT;
    HINT_FULL_JUSTIFY_CHILDREN_HORIZONTALLY;
    HINT_INCLUDE_ENDS_IN_CHILD_SPACING;
}

@object GenInteractionClass FontInteraction = {
    GI_comp = @RomanTrigger, @SansTrigger, @MonoTrigger;
    HINT_ORIENT_CHILDREN_VERTICALLY;
}

@object GenInteractionClass JustInteraction = {
    GI_comp = @LeftTrigger, @RightTrigger, @CenterTrigger, @FullTrigger;
    HINT_ORIENT_CHILDREN_VERTICALLY;
}

@object GenInteractionClass StyleInteraction = {
    GI_comp = @PlainTrigger, @BoldTrigger, @SuperTrigger, @SubTrigger;
    HINT_ORIENT_CHILDREN_VERTICALLY;
}

/* Triggers. All the upper triggers are essentially the same. Each is named as it
 * appears in the GI_comp fields above, and each has the corresponding
 * GI_visMoniker field. The RomanTrigger object is shown as an example. */

@object GenTriggerClass RomanTrigger = {
    GI_visMoniker = "Roman";
    HINT_EXPAND_WIDTH_TO_FIT_PARENT;
}

/* The Apply, Reset, and Cancel triggers do not have the hint shown above. */
@object GenTriggerClass ApplyTrigger = {
    GI_visMoniker = "Apply";
}
```

HINT_DIVIDE_WIDTH_EQUALLY and **HINT_DIVIDE_HEIGHT_EQUALLY** instruct a composite object to divide up its space along the affected dimension among the usable children. This hint can only suggest sizes for the children. The children themselves will decide if they can expand or contract to fit the requested space. (In general, children will not contract to fit a requested size, but they may expand.) Note: these hints will not work within **GenValue** objects; if you wish to use them on such an object, place them within a dummy **GenInteraction** object and use these hints on that object.

Typically, these hints work well in conjunction with `HINT_EXPAND_WIDTH_TO_FIT_PARENT` and `HINT_EXPAND_HEIGHT_TO_FIT_PARENT`. If these hints are not also in place (on a `GenInteraction`, for example) the division of width and/or height may be based on the children's default size; i.e., the parent will size itself based on total size of the children and then attempt to divide up its children's space, rather than the maximum allotted size for the parent, which is probably not desired.

12.2

12.2.4.3 Maximum, Minimum, Fixed, and Initial Sizes

`HINT_INITIAL_SIZE`, `HINT_MAXIMIMUM_SIZE`, `HINT_MINIMUM_SIZE`, `HINT_FIXED_SIZE`

If you know certain sizing restrictions for window or interaction objects, you can set them with the following four hints. Each takes a parameter that details the appropriate height and width in points. The parameters are described for the hints.

`HINT_INITIAL_SIZE`

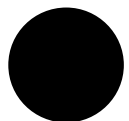
This hint sets the initial size of the object, if allowed. Other sizing restrictions may override this hint (such as the children being too large). This is primarily used with window objects (you will likely never use it for anything other than a `GenPrimary`, `GenDisplay`, or `GenView`), and it is effective only for when the object is first displayed. This hint takes an argument of type **CompSizeHintArgs**, a structure that defines a suggested width, height, and the number of children the object has.

`HINT_FIXED_SIZE`

This hint sets a fixed size for the object if one is allowed. Other sizing restrictions (such as children growing too large for the window) may override this hint occasionally. This hint can often be used for optimization to inhibit excessive geometry recalculations while on-screen. This hint takes the same parameters as `HINT_INITIAL_SIZE`, of type **CompSizeHintArgs**.

`HINT_MAXIMUM_SIZE`

This hint sets a desired maximum size for the object. Other geometry restrictions may override this hint when necessary.



This hint is useful for GenText and GenView objects. It takes the same parameters as HINT_INITIAL_SIZE, of type **CompSizeHintArgs**.

HINT_MINIMUM_SIZE

This hint sets a desired minimum size for the object. Other geometry restrictions may override this hint when necessary. The hint takes the same parameters as HINT_INITIAL_SIZE, of type **CompSizeHintArgs**.

12.2

The following examples show different ways to set the initial size of an object. The third argument of the hint is the number of children in a particular line of a composite, when the composite is set up to wrap its children.

This example sets the initial size of the composite (probably a GenPrimary) to be half the screen's height and half the screen's width.

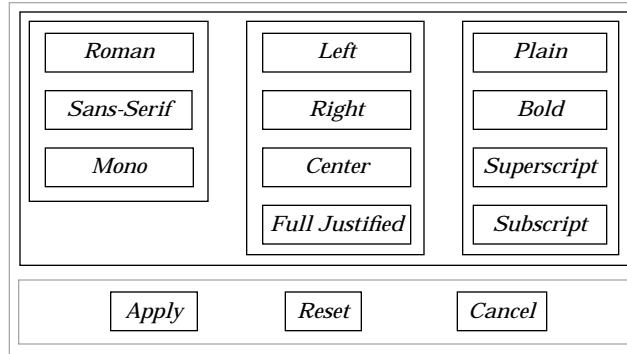
```
HINT_INITIAL_SIZE = {  
    SST_PCT_OF_FIELD_WIDTH | PCT_50,  
    SST_PCT_OF_FIELD_HEIGHT | PCT_50,  
    0 };
```

This example sets the size of the composite to be 100 pixels high and 200 pixels wide.

```
HINT_FIXED_SIZE = {  
    SST_PIXELS | 200,  
    SST_PIXELS | 100,  
    0 };
```

This example sets the composite's minimum size to be 10 average characters wide and 20 percent of the screen height tall.

```
HINT_MINIMUM_SIZE = {  
    SST_AVG_CHAR_WIDTHS | 10,  
    SST_PCT_OF_FIELD_HEIGHT | PCT_20,  
    0 };
```



12.2

Figure 12-8 *Drawing Boxes Around Composites*

When `HINT_DRAW_IN_BOX` is used for the upper composites, the object groupings become much easier to distinguish.

12.2.5 Outlining the Composite

`HINT_DRAW_IN_BOX`

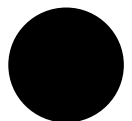
Often it is useful to outline a composite, especially if it is used for organizational purposes within another composite. You can use the hint `HINT_DRAW_IN_BOX` to have the UI automatically draw a box at the object's bounds as shown in Figure 12-8.

12.2.6 Using Monikers

Nearly all generic objects will have visual monikers (usually text labels) that get drawn somewhere in or near the object. The moniker helps the user distinguish between different objects; for example, each trigger should have a text (or graphical) moniker giving some indication of what happens when the trigger is pressed.

12.2.6.1 Placing Monikers in Relation to Children

`HINT_PLACE_MONIKER_ABOVE`, `HINT_PLACE_MONIKER_BELOW`,
`HINT_PLACE_MONIKER_TO_LEFT`, `HINT_PLACE_MONIKER_TO_RIGHT`,



```
HINT_DO_NOT_USE_MONIKER, HINT_CENTER_MONIKER,  
HINT_PLACE_MONIKER_ALONG_LARGER_DIMENSION,  
HINT_ALIGN_LEFT_MONIKER_EDGE_WITH_CHILD
```

Monikers are extremely useful and dynamic—see “GenClass,” Chapter 2 of the Object Reference Book, for full information on creating, manipulating, and using visual monikers. Monikers may be created or changed at run-time (often causing geometry updates), and all generic objects may be given monikers.

12.2

If you place a composite object’s moniker to one side of its children, the geometry manager will ensure extra space in the composite gets allotted for the moniker. You can hint that the moniker should be placed above, to the right of, to the left of, or below the children. You can also hint that the moniker should not be used.

Most specific UIs will put the moniker to the left of the object’s children by default. You can also center the moniker on the composite’s children with `HINT_CENTER_MONIKER`. Figure 12-9 shows several different moniker placements.

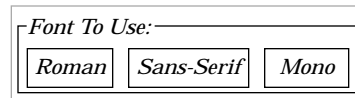
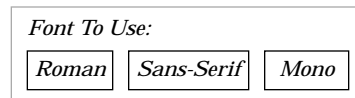


Figure 12-9 *Placing Monikers*

The top-left has the moniker placed to the left by default. The second uses the hint `HINT_PLACE_MONIKER_ABOVE`; notice that by default in this specific UI, the left moniker edge is aligned with the first child. The third also has `HINT_DRAW_IN_BOX`. The fourth also has the hint `HINT_CENTER_MONIKER` set to center the moniker.

You can also have the UI place the moniker along the object’s larger dimension—to the left if the screen is wider than it is tall, or above if the screen is taller than it is wide. To do this, set the hint `HINT_PLACE_MONIKER_ALONG_LARGER_DIMENSION`.

If a moniker is placed above a group of children, `HINT_ALIGN_LEFT_MONIKER_EDGE_WITH_CHILD` ensures that left moniker edge coincides with the left edge of the first child below that moniker. This is usually the default behavior anyway, but different specific UIs may have different default behavior.

12.2.6.2 Justifying or Centering on Monikers

`HINT_CENTER_CHILDREN_ON_MONIKERS`,
`HINT_LEFT_JUSTIFY_MONIKERS`

12.2

When you have several composites within a single composite, you can center the composites based on the position and size of the monikers. Using the hint `HINT_CENTER_CHILDREN_ON_MONIKERS` has the effect shown in Figure 12-10: It aligns the child composites so their monikers are right-justified at the center. Adding `HINT_LEFT_JUSTIFY_MONIKERS` has a similar effect, except the monikers are left-justified.

Note that the two hints `HINT_CENTER_CHILDREN_ON_MONIKERS` and `HINT_LEFT_JUSTIFY_MONIKERS` are valid only for composites with one or more child composites, all of which have monikers placed to their left.

12.2.6.3 Removing Moniker Borders

`HINT_NO_BORDERS_ON_MONIKERS`

This hint removes borders that may appear by default around an object's moniker. For example, GenTriggers under most specific UIs have a rectangular border drawn around their moniker's borders. This hint removes these borders. It may be useful in cases where a custom border should be implemented, but in general should not be used to override the specific UI, as this may confuse the user.

12.2.7 Using Custom Child Spacing

12.2

```
HINT_CUSTOM_CHILD_SPACING ,  
HINT_CUSTOM_CHILD_SPACING_IF_LIMITED_SPACE ,  
HINT_MINIMIZE_CHILD_SPACING
```

Normally, spacing of children within a composite is left entirely up to the specific UI. You can customize spacing, however, by using the hint `HINT_CUSTOM_CHILD_SPACING`. This hint takes a single argument of type **SpecSizeSpec**, described below.

The **SpecSizeSpec** structure defines both the spacing between children and how the spacing is determined. It exists as a record with two fields: The first field is a constant of type **SpecSizeTypes**, used to interpret the second field. The second field is data based on the type specified in the first field.

The different types allowable in the first field are

`SST_PIXELS` Data specified in pixels.

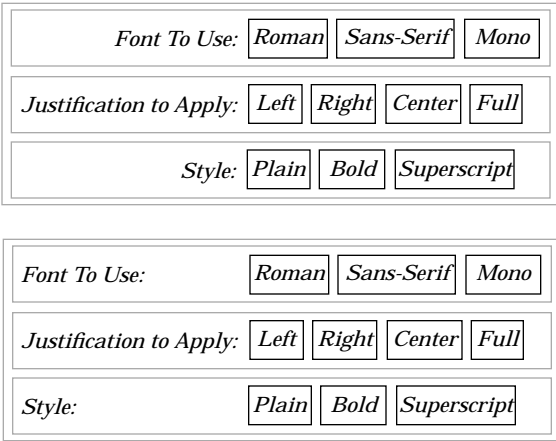


Figure 12-10 *Centering on Monikers*
The top composite has `HINT_CENTER_CHILDREN_ON_MONIKERS` set, causing the child composites to be lined up on the separation between their monikers and their children. The bottom composite also has the hint `HINT_LEFT_JUSTIFY_MONIKERS` as well, causing the same geometry with the addition of left-justifying the monikers.

SST_PCT_OF_FIELD_WIDTH

Data specified as a percentage of screen width. The data value is a fraction that gets multiplied by the width of the screen. Predefined fraction values to use in the data field are named PCT_XX, where the XX represents any multiple of five between zero and 100 inclusive. You can get better accuracy by using your own value if desired, however.

SST_PCT_OF_FIELD_HEIGHT

Data specified as a percentage of screen height. As above except uses screen height rather than width. 12.2

SST_AVG_CHAR_WIDTHS

Data specified as a number that gets multiplied by the average width of a character in the font being used. The data may be between zero and 1023 inclusive.

SST_WIDE_CHAR_WIDTHS

Data is specified as a number that gets multiplied by the maximum width of the widest character in the font being used. The data may be between zero and 1023 inclusive.

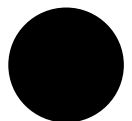
SST_LINES_OF_TEXT

Data is specified as a number of text lines. The number will be multiplied by the height of a line of text in the font being used.

Both fields of the record are defined by using the bitwise-or operator. If you had a composite that wanted to line all its children up exactly next to each other, for example, you would specify the hint as follows:

```
@object GenInteractionClass @MyComp = {
    GI_comp = @child1, @child2, @child3;
    HINT_CUSTOM_CHILD_SPACING = SST_PIXELS | 0;
}
```

HINT_CUSTOM_CHILD_SPACING_IF_LIMITED_SPACE also takes a **SpecSizeSpec** to suggest a custom amount of spacing, but only implements this custom spacing if the specific UI determines that the children are already too tightly arranged. This hint may or may not be helpful, as the specific UI often allocates a minimum amount of spacing independent of this hint.



`HINT_MINIMIZE_CHILD_SPACING` ensures that child spacing is kept to an absolute minimum, even if this means that object's edges will touch (in color systems) or even overlap (in black and white systems).

12.2.8 Allowing Children to Wrap

12.2

```
HINT_ALLOW_CHILDREN_TO_WRAP, HINT_WRAP_AFTER_CHILD_COUNT,  
HINT_DONT_ALLOW_CHILDREN_TO_WRAP,  
HINT_WRAP_AFTER_CHILD_COUNT_IF_VERTICAL_SCREEN
```

When a composite grows to fit its children and also must remain small enough to fit within its generic parent, it may wrap the children. This typically happens as a result of the user resizing a window smaller so the children of the composite no longer fit the window's width. Wrapping can occur in both the horizontal and the vertical, depending on the orientation of the composite object.

To allow a composite to wrap its children when necessary, apply the hint `HINT_ALLOW_CHILDREN_TO_WRAP` to the composite. If you want the children not to wrap, apply `HINT_DONT_ALLOW_CHILDREN_TO_WRAP`. An example of wrapping children appears in Figure 12-11.

When `HINT_WRAP_AFTER_CHILD_COUNT` is applied and children are allowed to wrap, the composite will cause its children to wrap after a certain number are displayed. This hint takes a single integer as a parameter. For example, if you wanted three children on every line in the composite no matter how many children there were total, you would specify the composite as follows:

```
@object GenInteractionClass @MyComp = {  
    GI_comp = @child1, @child2, @child3, @child4;  
    HINT_ALLOW_CHILDREN_TO_WRAP;  
    HINT_WRAP_AFTER_CHILD_COUNT = 3;  
}
```

The configuration shown above is demonstrated in Figure 12-11.

A variation, `HINT_WRAP_AFTER_CHILD_COUNT_IF_VERTICAL_SCREEN`, allows you to allow child wrapping based on whether or not the screen is

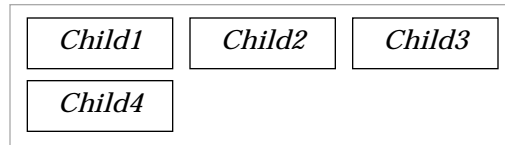


Figure 12-11 *Allowing Children to Wrap*

If the composite has `HINT_WRAP_AFTER_CHILD_COUNT = 3`, the composite will automatically wrap children after three on any given line.

12.2

“vertical.” If the screen is taller than it is wide, this hint will have an effect like `HINT_WRAP_AFTER_CHILD_COUNT`; if not, the hint will have no effect.

12.2.9 Object Placement

As we have seen, generic objects can be arranged and distributed in several ways. You may also place generic objects within certain areas of your user interface that are not explicitly defined. For example, in many dialog boxes, a “reply bar” is typically present where reply triggers such as “OK,” “Cancel,” or “Apply” are located. You may specify objects to generically place themselves within such a reply bar, even though you may not know the exact location of that reply bar as the specific UI creates it.

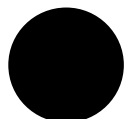
12.2.9.1 Reply Bars

`HINT_MAKE_REPLY_BAR`, `HINT_SEEK_REPLY_BAR`

One special feature that many dialog boxes have is a *reply bar*. Typically, any dialog box that allows the user to set a number of options before applying them will have a reply bar. A reply bar usually has triggers in it for “Apply,” “Cancel,” and perhaps other functions (such as “Reset”).

Note that many special dialog box types can have their reply bars built automatically according to the specific UI. See “GenInteraction,” Chapter 7 of the Object Reference Book, for full information on reply bars. If you want to create your own, however, you can with the hints in this section.

Although you have to declare each of the triggers that will appear in the reply bar, you can use `HINT_MAKE_REPLY_BAR` to set up the geometry appropriate



for the specific UI. For example, Code Display 12-4 gives the code that will create a reply bar similar to that shown in Figure 12-12.

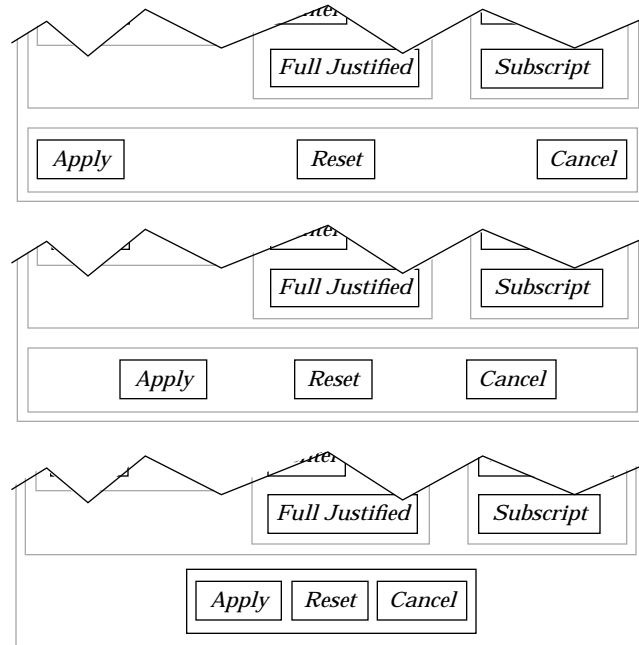
Code Display 12-4 Creating a Reply Bar

```
/* This reply bar has three triggers, Apply, Reset, and Cancel. A few possible
 * formats determined by the specific UI are shown in Figure 12-12. The
 * appropriate geometry is not determined until run-time. */
12.2 @object GenInteractionClass MyReplyBar = {
    GI_comp = @ApplyTrigger, @ResetTrigger, @CancelTrigger;
    HINT_MAKE_REPLY_BAR;
}

@object GenTriggerClass ApplyTrigger = {
    GI_visMoniker = "Apply"; }

@object GenTriggerClass ResetTrigger = {
    GI_visMoniker = "Reset"; }
```

```
@object GenTriggerClass CancelTrigger = {
    GI_visMoniker = "Cancel"; }
```

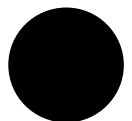


12.2

Figure 12-12 *Implementations of a Reply Bar*

The specific UI determines the actual geometry of the reply bar including justification, sizing, and other features.

HINT_SEEK_REPLY_BAR instructs the generic object (usually a GenTrigger) to seek placement in the dialog box's reply bar. The specific UI will try to place the trigger in the reply bar of the first dialog box GenInteraction it finds that is not GIT_ORGANIZATIONAL.



12.2.9.2 Placing Objects in Menu Bars

`HINT_SEEK_MENU_BAR`, `HINT_AVOID_MENU_BAR`

Usually, it is left up to the specific UI whether it places any given object within a menu bar. You can suggest that objects be placed within the window's menu bar by placing `HINT_SEEK_MENU_BAR` on that object. Similarly, you can place `HINT_AVOID_MENU_BAR` on an object to suggest that it not be placed within the window's menu bar.

12.2

12.2.9.3 Placing Objects in Scroller Areas

`HINT_SEEK_X_SCROLLER_AREA`, `HINT_SEEK_Y_SCROLLER_AREA`,
`HINT_SEEK_LEFT_OF_VIEW`, `HINT_SEEK_TOP_OF_VIEW`,
`HINT_SEEK_RIGHT_OF_VIEW`, `HINT_SEEK_BOTTOM_OF_VIEW`

These hints affect the placement of generic objects within GenViews. The objects must be children of a GenView for the hints to take effect.

`HINT_SEEK_X_SCROLLER_AREA` and `HINT_SEEK_Y_SCROLLER_AREA` suggest that a generic object be placed alongside the scrollbar area—either the horizontal scrollbar or the vertical scrollbar, respectively—of the GenView object.

`HINT_SEEK_LEFT_OF_VIEW`, `HINT_SEEK_TOP_OF_VIEW`,
`HINT_SEEK_RIGHT_OF_VIEW` and `HINT_SEEK_BOTTOM_OF_VIEW` suggest that the generic object be placed alongside the respective side of the GenView.

12.2.9.4 Placing Objects in Window Title Bars

`HINT_SEEK_TITLE_BAR_LEFT`, `HINT_SEEK_TITLE_BAR_RIGHT`

These hints suggest the placement of a generic object within a window's title bar. These hints are usually used on GenTriggers or GenInteractions; the objects involved should fit within the title bar area (i.e., be equivalent to tool bar icons). For each window with a title bar, only one object may have each of these hints, and the object must be a *direct* child of the windowed object.

12.2.9.5 Placement of Objects Popped Up

HINT_POPS_UP_TO_RIGHT, HINT_POPS_UP_BELOW

HINT_POPS_UP_TO_RIGHT

This hint instructs the specific UI to bring up the object to the right of its activating gadget, if the object is normally popped up below the activating gadget.

HINT_POPS_UP_BELOW

This hint instructs the specific UI to bring up the object below its activating gadget. This is usually the default behavior for objects that are popped up.

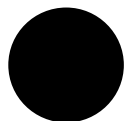
12.3

12.3 Positioning and Sizing Windows

A generic object that appears as a window, such as an independently-displayable GenInteraction or a GenPrimary, has special needs for positioning and sizing itself, especially when appearing initially. Specific user interfaces may differ in how windows appear on the screen. In Motif, for instance, windows initially appear in one of several staggered positions. Primary and display objects extend by default to the right edge of the screen and to the top edge of the icon area at the bottom of the screen. Other windows are usually only as large as required by their children.

The **SpecSizeWinPair** structure is often used when specifying window sizes and positions. There are several hints which signal that a window should be positioned or sized depending on the position or size of another window, the size of the screen, or other conditions. The **SpecSizeWinSpec** structure allows you to specify an offset from another window, either by a number of points (remember that there are 72 points per inch), or by means of a ratio.

```
typedef struct {
    SpecWinSizeSpec    SWSP_x;
    SpecWinSizeSpec    SWSP_y;
} SpecWinSizePair;
```



```
typedef WordFlags SpecWinSizeSpec;  
#define SWSS_RATIO      0x8000  
#define SWSS_SIGN       0x4000  
#define SWSS_MANTISSA    0x3c00  
#define SWSS_FRACTION    0x03ff
```

12.3

The **SpecSizeWinPair** structure allows you to set up separate parameters for the *x* and *y* coordinates. For each coordinate, use the **SWSS_RATIO** flag to signal whether you are giving coordinates by means of a ratio instead of a constant. If you are using a ratio, use the **SWSS_FRACTION**, **SWSS_MANTISSA**, and **SWSS_SIGN** fields to specify the value of the ratio or use one of the following pre-defined constants to specify a ratio between zero and one:

```
#define PCT_0      0x000  
#define PCT_5      0x033  
#define PCT_10     0x066  
#define PCT_15     0x099  
#define PCT_20     0x0cc  
#define PCT_25     0x100  
#define PCT_30     0x133  
#define PCT_35     0x166  
#define PCT_40     0x199  
#define PCT_45     0x1cc  
#define PCT_50     0x200  
#define PCT_55     0x233  
#define PCT_60     0x266  
#define PCT_65     0x299  
#define PCT_70     0x2cc  
#define PCT_75     0x300  
#define PCT_80     0x333  
#define PCT_85     0x366  
#define PCT_90     0x399  
#define PCT_95     0x3cc  
#define PCT_100    0x3ff
```

To specify a constant offset, just use that offset, not setting the **SWSS_RATIO** flag. Not all windows will use the **SpecWinSizePair** structure.

12.3.1 Window Positioning

```
HINT_POSITION_WINDOW_AT_RATIO_OF_PARENT,
HINT_POSITION_WINDOW_AT_MOUSE, HINT_STAGGER_WINDOW,
HINT_CENTER_WINDOW, HINT_TILE_WINDOW,
HINT_WINDOW_NO_CONSTRAINTS
```

GenClass has a number of hints used by the window classes to determine where the window appears initially and how the window is allowed to move about the screen. Specific UIs can override these hints when necessary or when they conflict with the UI's specifications. These hints are listed below. 12.3

HINT_POSITION_WINDOW_AT_RATIO_OF_PARENT

This hint suggests an initial position for the window based on its parent window's size and relative to the upper-left corner of the parent window. The suggested position is specified as a fraction of the parent window's size in a **SpecWinSizePair** structure. This structure consists of two fields, the X position and the Y position of the new window's origin. Predetermined fraction values are names **PCT_xx**, where **xx** is a multiple of five between zero and 100 inclusive. For example, to have the window object's origin start out in the center of its parent, specify the hint as follows:

```
HINT_POSITION_WINDOW_AT_RATIO_OF_PARENT = {
    SWSS_RATIO | PCT_25, SWSS_RATIO | PCT_60
};
```

HINT_POSITION_WINDOW_AT_MOUSE

This hint will position the origin of the new window at the mouse pointer.

HINT_STAGGER_WINDOW

This hint indicates that the window's origin should be placed just below and to the right of the origin of the last staggered window.

HINT_CENTER_WINDOW

This hint centers the window in its parent window.

HINT_TILE_WINDOW

This hint tiles the window with other windows that are also

marked `HINT_TILE_WINDOW` within the parent window. This hint is not currently supported.

`HINT_WINDOW_NO_CONSTRAINTS`

This hint removes all constraints upon the positioning of a window. It should only be used as the final attempt to position the window if other methods do not work to your satisfaction.

12.3

12.3.2 Determining Initial Size

`HINT_EXTEND_WINDOW_TO_BOTTOM_RIGHT`,
`HINT_EXTEND_WINDOW_NEAR_BOTTOM_RIGHT`,
`HINT_SIZE_WINDOW_AS_DESIRED`,
`HINT_SIZE_WINDOW_AS_RATIO_OF_PARENT`,
`HINT_SIZE_WINDOW_AS_RATIO_OF_FIELD`,
`HINT_USE_INITIAL_BOUNDS_WHEN_RESTORED`

GenClass has a number of hints used by the window classes to determine the size of the window object initially and how the window is allowed to be resized on the screen. Specific UIs can override these hints when necessary or when they conflict with the UI's specifications. These hints are listed below.

`HINT_EXTEND_WINDOW_TO_BOTTOM_RIGHT`

This hint extends the window's bounds so its lower right corner is located at the lower right corner of its parent window.

`HINT_EXTEND_WINDOW_NEAR_BOTTOM_RIGHT`

This hint extends the window's bounds so its lower right corner is located near the lower right corner of its parent window. Enough room will be left for any icon area that may be present. Most major applications will apply this hint to their `GenPrimary` objects.

`HINT_SIZE_WINDOW_AS_DESIRED`

This hint allows the window to size itself according to the combined size of its children.

`HINT_SIZE_WINDOW_AS_RATIO_OF_PARENT`

This hint sizes the window to a ratio of its parent window. It takes a single parameter of type **SpecWinSizePair**, described under `HINT_POSITION_WINDOW_AT_RATIO_OF_PARENT`.

Instead of coordinates, however, the ratios in the structure are width and height.

HINT_SIZE_WINDOW_AS_RATIO_OF_FIELD

This hint sizes the window to a ratio of the field window's size. It takes a **SpecWinSizePair** parameter indicating the ratio. See above.

HINT_USE_INITIAL_BOUND_WHEN_RESTORED

This hint instructs the window to come on-screen with its original values rather than the state of the window when it was saved to state.

12.3

12.3.3 On-Screen Behavior

HINT_KEEP_INITIALLY_ONSCREEN,
HINT_DONT_KEEP_INITIALLY_ONSCREEN,
HINT_KEEP_PARTIALLY_ONSCREEN,
HINT_DONT_KEEP_PARTIALLY_ONSCREEN,
HINT_KEEP_ENTIRELY_ONSCREEN,
HINT_KEEP_ENTIRELY_ONSCREEN_WITH_MARGIN, HINT_NOT_MOVABLE

HINT_KEEP_INITIALLY_ONSCREEN

This hint will make the window resize itself to fit entirely within its parent window. Most windows will inherently know how to come up fully on-screen; some specific UIs do not implement it by default for optimization purposes.

HINT_DONT_KEEP_INITIALLY_ONSCREEN

This hint will allow the window to come up whether it will be fully on-screen or not. If the default is to keep the window entirely on-screen and the UI allows this hint, the window will be allowed to open partially off-screen.

HINT_KEEP_PARTIALLY_ONSCREEN

This hint indicates that when the user moves the window, it will be forced to remain partially on-screen.

HINT_DONT_KEEP_PARTIALLY_ONSCREEN

This hint indicates that the user can move the window around the screen at will, even if the window becomes completely obscured under its parent.

`HINT_KEEP_ENTIRELY_ONSCREEN`

This hint indicates that when the user moves the window, it will be forced to remain entirely on-screen.

`HINT_KEEP_ENTIRELY_ONSCREEN_WITH_MARGIN`

This hint indicates that when the user moves the window, it will be forced to remain on-screen. An additional margin is added to the edge of the screen to keep the window from touching the edge as well; this margin is defined by the specific UI.

12.3

`HINT_NOT_MOVABLE`

This hint instructs the specific UI to make the windowed object not movable. This hint should be avoided if at all possible.

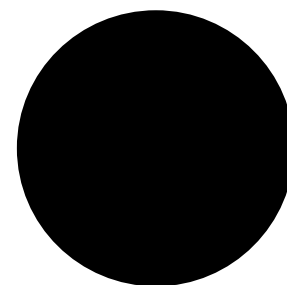
12.3.4 Window Look and Feel

`HINT_WINDOW_NO_TITLE_BAR`, `HINT_WINDOW_NO_SYS_MENU`

`HINT_WINDOW_NO_TITLE_BAR` instructs the specific UI to remove the title bar from a windowed object (dialog box, `GenPrimary` or `GenDisplay`). Use this hint with extreme prejudice as most of a window's functionality will not be accessible without a title bar.

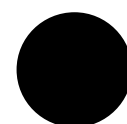
`HINT_WINDOW_NO_SYS_MENU` instructs the specific UI to remove the system menu (or close button) from a windowed object. Again, do not use this hint unless necessary, as it overrides important features of windowed objects.

Sound Library



13

13.1	Goals and Motives	505
13.2	Playing UI Sounds	505
13.3	Representing Tones	506
13.4	Single Notes	510
13.5	Declaring Music Buffers.....	513
13.6	Playing Music Buffers	516
13.7	Playing Very Large Music Buffers.....	518
13.8	Playing Sampled Sounds.....	518
13.9	Grabbing the Sound Exclusive.....	520
13.10	Simulating Musical Instruments.....	521
13.10.1	Acoustics In Brief.....	521
13.10.2	Simple Instrument Description	522
13.10.3	Advanced Description.....	523





GEOS includes a powerful sound library. The library allows users to play individual notes, sequences of notes, and sampled sounds. It also provides support for those sound devices which allow the sampling and playback of digitized sounds.

13.1

13.1 Goals and Motives

The sound library provides multiple tiers of support for a variety of devices. You may program multiple-voice compositions (musical pieces which require playing more than one note at a time), and the sound driver will determine how to best play the sounds if the device does not have enough voices.

The sound library also shows the advantage of a preemptive multithreading system like GEOS. The library sets up queues of sounds and arranges for the kernel's thread to call the sound driver functions at appropriate times. An application can thus arrange ten seconds of music then continue operation as the music plays in the background.

The system provides a routine to play standard sounds. By using these, the programmer can keep his interface consistent with that of existing programs. When the standard system sounds are inappropriate, the programmer can use other routines to completely specify the sound's pitch and duration.

13.2 Playing UI Sounds

`UserStandardSound()`

When a program needs to create a UI-related sound, it should call the routine **UserStandardSound()**. This routine plays a sound, silencing any other sounds which may have been playing. The program may specify either a system-standard sound, a custom sound, or a sequence of custom sounds.

Note that if the user has disabled UI sound (via the Preference Manager application), **UserStandardSound()** will not play the sound. The sound will

be played at a high priority—the UI sound will take over the speakers in spite of any sound that some other program is generating.

To play one of the standard UI sounds, call **UserStandardSound()** with a single argument—a member of the **StandardSoundType** enumerated type. **StandardSoundType** has the following members:

SST_ERROR The sound produced when an error box appears.

SST_WARNING
General-purpose warning tone.

SST_NOTIFY General-purpose tone to notify the user of some occurrence.

SST_NO_INPUT
Sound produced when the user is making mouse-clicks or keypresses that are not going anywhere—the user is clicking outside of a modal dialog box, etc.

SST_KEY_CLICK
Sound produced when the user types on a real or virtual keyboard.

SST_CUSTOM_SOUND

SST_CUSTOM_NOTE

SST_CUSTOM_BUFFER
These are not actually standard sound values, but in fact act as signals that you wish to provide a custom sound buffer. **SST_CUSTOM_BUFFER** means that you will provide a pointer to a buffer containing the music data. **SST_CUSTOM_SOUND** signals that you are providing a handle to a pre-allocated sampled sound. **SST_CUSTOM_NOTE** signals that you wish to play a single note. To learn more about creating custom notes and music buffers, see below.

13.3 Representing Tones

Tones are defined by the following characteristics:

- ◆ Frequency (i.e. pitch), expressed in terms of cycles/second (Hertz or Hz). The higher the frequency, the higher the pitch.

- ◆ Duration, which may be measured in milliseconds, sixtieth of a second ticks, or by means of a “tempo.”
- ◆ Volume, a relative measure of loudness.
- ◆ Instrument, which instructs sound cards which are capable of imitating certain musical instruments to do so.

The sound library expects sounds to be specified in terms of cycles/second. However, constants have been defined which associate three octaves of tones with their values in Hz. As a benchmark, middle A is defined as 880 Hz. Doubling the frequency of a pitch is equivalent to raising it by one octave. Since there are twelve equal half-steps in an octave, multiplying the frequency of a tone by $^{12}\sqrt{2}$ is equivalent to raising the pitch by a single half-step. The constants are defined with the formats

13.3

```
octave_note
octave_note_accidental
```

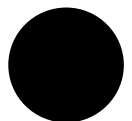
where *octave* is “LOW,” “MIDDLE,” or “HIGH”; *note* is one of “A” through “G”; and *accidental*, if present, is either “SH” (for sharp) or “b” (for flat). The lowest constant defined is LOW_C, approximated as 262 Hz; the highest is HIGH_B, approximately 1,976 Hz.

You may use a frequency that is beyond the range of human hearing (which extends from about 20Hz-20,000Hz). Thus, keep in mind that if you can’t hear a note, the problem may not be that you’ve set the volume too low.

Duration may be specified in a number of units, by means of the **SoundStreamDeltaTimeType** enumerated type.

```
#define SSDTT_MSEC 8
#define SSDTT_TICKS 10
#define SSDTT_TEMPO 12
```

Duration may be expressed in milliseconds, timer ticks (each tick is one sixtieth of a second), or fractions of notes. Of course, there is no set length for a note, but when you actually play the note, you will provide a *tempo* value which will be used to compute the actual sound length. This tempo is just the



number of milliseconds to play a one-hundred-and-twenty-eighth note. A number of constants have been set up to aid you in computing note lengths:

```
#define WHOLE          128
#define HALF           64
#define QUARTER        32
#define EIGHTH         16
#define SIXTEENTH       8
#define THIRTYSECOND    4
#define SIXTYFOURTH     2
#define ONE_HUNDRED_TWENTY_EIGHTH 1
#define DOUBLE_DOT_WHOLE (WHOLE + HALF + QUARTER)
#define DOUBLE_DOT_HALF (HALF + QUARTER + EIGHTH)
#define DOUBLE_DOT_QUARTER \
    (QUARTER + EIGHTH + SIXTEENTH)
#define DOUBLE_DOT_EIGHTH \
    (EIGHTH + SIXTEENTH + THIRTYSECOND)
#define DOUBLE_DOT_SIXTEENTH \
    (SIXTEENTH + THIRTYSECOND + SIXTYFOURTH)
#define DOTTED_WHOLE    (WHOLE + HALF)
#define DOTTED_HALF     (HALF + QUARTER)
#define DOTTED_QUARTER  (QUARTER + EIGHTH)
#define DOTTED_EIGHTH   (EIGHTH + SIXTEENTH)
#define DOTTED_SIXTEENTH (SIXTEENTH + THIRTYSECOND)
#define DOTTED_THIRTYSECOND \
    (THIRTYSECOND + SIXTYFOURTH)
```

If you're not familiar with the terms “double-dotted eighth note” and “tempo,” don't worry. Basically, tempo allows you specify your notes' durations in units when defining your sound buffer, but delay specifying the length of those units until playing time. Thus, you might specify that a note should be 16 units long when composing your sound buffer. You could then play that note with a tempo of 100 milliseconds and the note would sound for 1.600 seconds. You might play the sound again later with tempo 50, and that note would last only 0.800 seconds. You would not have to do anything to the musical composition itself to change this time-scale.

Volume is a relative value, depending on the strength of the machine's speakers. You may specify the volume with a word-length value. The following constants have been set up to aid you in specifying a volume:

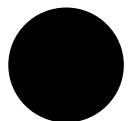
```
#define DYNAMIC_FFFF    0xffff /* very loud */
#define DYNAMIC_FFF     0xdfff /* fortississimo */
#define DYNAMIC_FF      0xbfff /* fortissimo */
#define DYNAMIC_F       0x9fff /* forte */
#define DYNAMIC_MF      0x8fff /* mezzo forte */
#define DYNAMIC_MP      0x6fff /* mezzo piano */
#define DYNAMIC_P       0x5fff /* piano */
#define DYNAMIC_PP      0x3fff /* pianissimo */
#define DYNAMIC_PPP     0x1fff /* pianississimo */
#define DYNAMIC_PPPP    0x01ff /* is it playing? */
```

13.3

By way of comparison, a typical alert beep plays with DYNAMIC_FFF (sometimes called “fortissimo,” or even “fortississimo”) volume.

Some sound devices emulate musical instruments. You may ask that notes be played using one of these instruments or even define your own instrument. You will specify what sort of instrument you wish to use by means of a segment:offset pointer. If the segment is non-null, the pointer should point to the instrument's data (for information about setting up this data, see section 13.10.3 on page 523); if the segment is NULL, then the “offset” will be interpreted as a standard instrument value. The standard instrument “patches” are listed in sound.h; they have names like IP_ACOUSTIC_GRAND_PIANO and IP_ELECTRIC_SNARE.

Note that many simulated percussion instruments have been set up so that they sound most realistic when playing notes of a particular frequency. If there is a frequency constant with a name derived from the name of the instrument, you should probably play notes of that frequency. For example, to simulate an “electric snare,” you would use IP_ELECTRIC_SNARE to play a note of frequency FR_ELECTRIC_SNARE.



13.4 Single Notes

```
SoundAllocMusicNote(), SoundPlayMusicNote(),
SoundStopMusicNote(), SoundReallocMusicNote(),
SoundFreeMusicNote()
```

13.4

The simplest way to play a note is by using **UserStandardSound()** with the **SST_CUSTOM_NOTE** sound type. In this mode, the routine takes a frequency and a duration in ticks. The sound will be played immediately, played using the instrument patch **IP_REED_ORGAN** if the sound device supports simulated instruments. The note will play with medium loudness.

To play a note at a lower priority (so it won't necessarily interrupt other sounds), or with a different loudness, or as if by an instrument other than a reed organ, you will create and play a custom note.

The task of playing a custom note breaks down as follows:

- 1 Allocate the note with **SoundAllocMusicNote()**.
- 2 Play it with **SoundPlayMusicNote()** or **UserStandardSound()**.
- 3 Free it with **SoundStopMusicNote()** and **SoundFreeMusicNote()**.

First, you must set up the data structure containing the note's data. **SoundAllocMusicNote()** takes a frequency, a duration, a volume, and an instrument and returns the handle to a data structure which the sound library will understand. See Code Display 13-1 for an example.

Code Display 13-1 Allocating Single Music Notes

```
/* BigBlatt is a Cb (C flat) Half-note, which will sound as if played by a trumpet.
 * This means a tone of 494Hz with duration (128 * playing tempo). */
SoundErr = SoundAllocMusicNote ( IP_TRUMPET, 0, /* Play as if with a trumpet */
                                MIDDLE_C_b, /* C flat, 494 Hz. */
                                DYNAMIC_FFF, /* Play it rather loudly */
                                SSDTT_TEMPO, /* Length based on 128th notes */
                                WHOLE, /* Duration = 128/128 notes */
                                &BigBlatt);
```

```

/* SubliminalTone is an A note lasting 0.3 sec which will sound as if it was being
 *   played very quietly on an electric piano. It will play at 880Hz. */
SoundAllocMusicNote(
    IP_ELECTRIC_PIANO_1, 0, /* As if on piano */
    MIDDLE_A,           /* Middle A, 880 Hz. */
    DYNAMIC_PP,         /* Play it rather quietly. */
    SSDTT_MSEC,         /* Length based on milliseconds */
    300,                /* Duration = 0.300 seconds */
    &SubliminalTone);

```

13.4

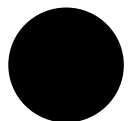
There are two ways to play a note. The first is to call **UserStandardSound()** with the **SST_CUSTOM_SOUNDS** sound type, passing the handle returned by **SoundAllocMusicNote()**. The other is to call **SoundPlayMusicNote()**, supplying the proper tempo and priority level. **UserStandardSound()** uses a tempo of 8 millisecond per one-hundred-and-twenty-eighth note; this is somewhat fast; tempo-based notes are not normally used with **UserStandardSound()**.

When playing a note or sound buffer, you must provide a sound priority level. Sounds may interrupt one another, so if you ask to play a sound more important than that which is presently playing, your sound will take precedence and will play immediately. The following sound priority levels are available:

SP_SYSTEM_LEVEL	SP_IMMEDIATE
SP_ALARM	SP_THEME
SP_STANDARD	
SP_GAME	
SP_BACKGROUND	

The lower the priority's numerical value, the more important the sound. The **SP_SYSTEM_LEVEL_IMMEDIATE** constant is the most important of those listed above and has the lowest numerical value.

The **SP_IMMEDIATE** and **SP_THEME** values are a bit unusual in that they are meant to be combined with the other priorities, as modifiers of base priority. Thus a sound whose priority was (**SP_STANDARD** + **SP_IMMEDIATE**) would edge out a sound which was just **SP_STANDARD**. A sound which had priority (**SP_STANDARD** + **SP_THEME**) would be edged out by the **SP_STANDARD_SOUND**. And all of these sounds would take precedence over a sound whose base priority was **SP_GAME** or **SP_BACKGROUND**.



Code Display 13-2 Playing Single Music Notes

```
/* UserStandardSound() with note:
 * We will play a quick high E note for 0.020 seconds. */
UserStandardSound(SST_CUSTOM_NOTE, HIGH_E, 20);

/* UserStandardSound() with allocated note:
 * We will play the Subliminal tone immediately. */
UserStandardSound(SST_CUSTOM_SOUND, SubliminalTone, 0);

13.4 /* SoundPlayNote():
 * Total playing time of BigBlatt will be (WHOLE*62)=(128*8)=1024 msec = 1+ sec. */
SoundPlayMusicNote(BigBlatt, SP_STANDARD + SP_IMMEDIATE, 8, EOSF_UNLOCK);
```

You may stop a note from playing at any time by invoking **SoundStopMusicNote()**.

To free the note's data structure, call **SoundFreeMusicNote()**. You should make sure that the note has stopped playing first (call **SoundStopMusicNote()** if you are not sure).

To change any of the characteristics of a note, call **SoundReallocMusicNote()**. You should make sure that the note has stopped playing first (call **SoundStopMusicNote()** if you are not sure).

Code Display 13-3 Stopping and Freeing a Note

```
/* Stop the SubliminalTone if we're playing it, then change the stored note */
SoundStopMusicNote(SubliminalTone);
SoundReallocMusicNote(SubliminalTone, LOW_C, DYNAMIC_PPP,
                      SSDTT_MSEC, 500, IP_TINKLE_BELL, 0);

/* Stop the BigBlatt if we're playing it, then discard it */
SoundStopMusicNote(BigBlatt);
SoundFreeMusicNote(BigBlatt);
```

13.5 Declaring Music Buffers

You may set up a buffer of notes to be played. This buffer may include information for several voices. If you are familiar with constructing MIDI streams, you will find that setting up GEOS music buffers is a similar process.

Sound buffers are made up of events and timer ticks. The various event types are defined in the enumerated type **SoundStreamEvent**—each event is signalled by one of the values listed below and includes one or more words of data describing the event specifics:

13.5

SSE_VOICE_ON

Start a note. The next words of data contain the note's characteristics:

- voice** The voice on which the note should play. If your composition has notes playing on more voices at a time than the device can handle, then some will not be played.
- frequency** The note's frequency (e.g. MIDDLE_C).
- volume** The note's loudness (e.g. DYNAMIC_MF).

SSE_VOICE_OFF

End a note. This does not necessarily mean that the voice will stop making sound—this corresponds to releasing the key of a piano, and some sound may linger on. Ending the note also signifies that the voice is “free”; if another sound on the queue wants to appropriate that voice, then it can. There is one word of data:

- voice** The voice which should stop playing.

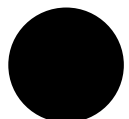
SSE_CHANGE

Change a voice's instrument. There are three words of data:

- voice** The voice to be changed.
- pointer** Pointer (two words) to new instrument structure to use (or standard **InstrumentPatch** value such as IP_CLARINET followed by the IP_STANDARD_TABLE word constant to signal that a standard instrument is being used).

SSE_GENERAL

A miscellaneous event (some possible values are discussed below). The number and meaning of data words will depend on the event.



Time is expressed as a unit and a duration. The unit is one of the **SoundStreamDeltaTimeType** values; the duration specifies how many of these units should be allowed to pass.

Code Display 13-4 Simple Sound Buffer Example

```
#define MELODY 0
#define PERC 1
13.5 static const word themeSongBuf[] = {
    SSE_CHANGE,      MELODY, IP_FLUTE,          /* We want voice 0 to
                                                    * play like a flute. */

    SSE_CHANGE,      PERC, IP_ACOUSTIC_SNARE,    /* voice 1 should play
                                                    * like a drum. */

    SSE_GENERAL,     GE_SET_PRIORITY, SP_GAME,   /*Set priority of sound */

    SSE_VOICE_ON,    PERC, FR_ACOUSTIC_SNARE, DYNAMIC_F, /* Hit the drum. We are
                                                    * hitting it hard (forte)
                                                    * with a C note. */

    SSDTT_TEMPO,     QUARTER,                    /*A quarter note passes...*/

    SSE_VOICE_OFF,   PERC,                      /* Pick up drumstick... */

    SSE_VOICE_ON,    PERC, FR_ACOUSTIC_SNARE, DYNAMIC_F, /* hit the drum */

    SSDTT_TEMPO,     EIGHTH,
    SSE_VOICE_OFF,   PERC,                      /* Pick up drumstick */

    SSDTT_TEMPO,     EIGHTH,                    /* For one eighth-note, no
                                                    * instruments playing */

    SSE_VOICE_ON,    PERC, FR_ACOUSTIC_SNARE, DYNAMIC_F, /* Tap the drum... */

    SSE_VOICE_ON,    MELODY, LOW_C, DYNAMIC_MP,  /* ...and start the flute */

    SSDTT_TEMPO,     QUARTER,                    /* Advance a quarter-note
                                                    * of time. */

    SSE_VOICE_OFF,   PERC,                      /* Pick up the drum stick
                                                    * (but we'll continue
                                                    * the flute) */

    SSE_CHANGE,      PERC, IP_COWBELL,          /* Switch instruments */
}
```

```

SSE_VOICE_ON,   PERC, FR_COWBELL, DYNAMIC_F,    /* Hit bell */
SSDTT_TEMPO,    EIGHTH,                      /* Advance eighth note */
SSE_VOICE_OFF,  PERC,                          /* Stop hitting bell */
SSDTT_TEMPO,    EIGHTH,                      /* Advance one more eighth... */

SSE_VOICE_OFF,  MELODY,                       /* ...and stop blowing on flute */
SSDTT_TICKS,    120,                         /* Let sound die out
                                           * for two seconds */

SSE_GENERAL,    GE_END_OF_SONG};             /* We're done */

```

13.5

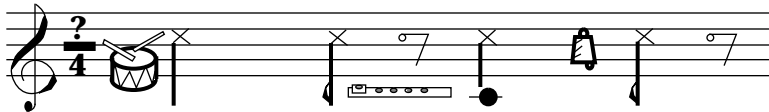


Figure 13-1 Sheet Music Excerpt for Example

Percussion will play two notes then rest. Next, melody will play a long note, while percussion plays a note, switches instruments, plays another note, and rests.

The song buffer must end with a GE_END_OF_SONG event.

The following general events may be incorporated into any sound buffer:

GE_NO_EVENT

No event at this time. This event may be sandwiched between other events when a long duration is necessary.

GE_END_OF_SONG

Signals the end of the sound buffer. Every song buffer must end with this event.

GE_SET_PRIORITY

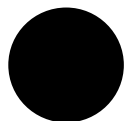
Change the sound's priority. It has one word of data:

priority New priority level to use.

GE_SET_TEMPO

Change the sound's tempo. It has one word of data:

tempo New tempo to use, as measured by the number of milliseconds to apportion a one-hundred-and-twenty-eighth note.



GE_SEND_NOTIFICATION

Send a message to an object upon reaching this point in the sound buffer. It has three words of data, consisting of a Message and an optr:

message This is a Message value, the message to send to the object. Note that this message should not require any arguments. This message will be sent as if with the *forceQueue* message flag.

object This is an optr, the message's intended recipient. This object may be disabled.

13.6

GE_V_SEMAPHORE

Release a semaphore. There is one word of data:

semaphore The handle of the semaphore to release ("V").

13.6 Playing Music Buffers

`SoundAllocMusic()`, `SoundInitMusic()`, `SoundPlayMusic()`,
`SoundStopMusic()`, `SoundReallocMusic()`, `SoundFreeMusic()`

If your sound buffer is stored in a movable resource, and you only want that resource to be locked down while the music is playing, call

SoundInitMusic() and pass the handle of a block containing the music. The block should contain a `SimpleSoundHeader` structure which specifies the number of voices in the buffer, followed by a music buffer as described above.

If your sound buffer is in fixed or locked memory, you must allocate space in the area accessible to the sound library by calling **SoundAllocMusic()**. This returns a handle to a sound block which you will be passing to other sound routines. You may pass a flag requesting that the data from which the sound was allocated be automatically unlocked and/or freed. Note that since sound buffers often become rather large, it is normally a bad idea to leave them in fixed memory, and normally inadvisable to make them fixed for very long. If you use **SoundAllocMusic()**, the sound buffer will have to remain locked for as long as the music is playing.

To play the sound, call **SoundPlayMusic()**. You must provide the handle returned from the sound allocation, a tempo, and a priority.

Code Display 13-5 Preparing and Playing Sound Buffers

```
SoundInitMusic(MySongResource, 1);

SoundErr = SoundAllocMusic(    &themeSongBuf,
                               2,                               /* themeSongBuf has two voices,
                                                                * MELODY and PERC. */
                               &theSong);

SoundPlayMusic(theSong, SP_STANDARD, 125, EOSF_UNLOCK);

UserStandardSound(SST_CUSTOM_SOUND, MySongResource, 33);
```

13.6

To stop playing the sound, call **SoundStopMusic()**. This routine will not affect songs which are not playing.

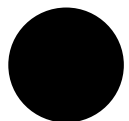
To free up the sound handle and data associated with the sound, call **SoundFreeMusic()**. You should never free a sound which is playing; if you think that a sound may be playing, call **SoundStopMusic()** before freeing it.

To reuse a sound handle with a new sound buffer, call **SoundReallocMusic()**. The number of voices may not be changed by reallocation. You should never reallocate a sound which is playing; if you think that a sound may be playing, call **SoundStopMusic()**.

Code Display 13-6 Stopping and Freeing a Sound

```
/* We are done with the theme song and want to play our ending song. Instead of
 * allocating a new song handle, we'll just reallocate the one we were using to
 * store the data taken from the themeSongBuf, and use the data from
 * endingSongBuf. The endingSongBuf had better only contain data for two
 * voices (or fewer). */
SoundStopMusic(theSong);
SoundReallocMusic(theSong, (_far)&endingSongBuffer, FREE_ON_EOS);
SoundPlayMusic(theSong, SP_STANDARD, 125);

/* If we're done with "theSong" handle, then we can free it, after making sure
 * it has stopped playing. */
SoundStopMusic(theSong);
SoundFreeMusic(theSong);
```



13.7 Playing Very Large Music Buffers

```
SoundAllocMusicStream(), SoundPlayToMusicStream(),  
SoundStopMusicStream(), SoundFreeMusicStream()
```

13.7

You may wish to play a sound which takes up a great deal of memory. If you wish to play an extremely long music buffer, you can play parts of it to a music stream. For instance, to play a 1 Megabyte music buffer (requiring too large a block of memory for the routines described in the above sections), you would play 4K of it at a time to a music stream, which would concatenate the music together. Normally, these “music stream” commands are the best to use when playing a music buffer whose sound resides on a disk. To allocate a special stream which will accept music data, call

SoundAllocMusicStream(). To play music into the stream, call **SoundPlayToMusicStream()**, passing a pointer to the buffer of music to play. To stop the passage of music through the stream and flush the buffer, call **SoundStopMusicStream()**. To free up the music stream, call **SoundFreeMusicStream()**. Make sure that there is no music playing through the stream when you free it; be sure to call **SoundStopMusicStream()** if you are not otherwise sure.

13.8 Playing Sampled Sounds

Once you have your sound data structure set up, the basic steps to playing the sound are

- 1 Allocate a DAC stream.
- 2 Enable the DAC stream.
- 3 Play your sampled sound data into the enabled stream. Play the sound as many times as you like. You may play other sampled sounds to the same stream.
- 4 Disable the DAC stream when you're finished with it.

To allocate a DAC stream, call **SoundAllocSampleStream()**. This will return a memory handle which you will use to identify the sound stream as you manipulate it with other routines.

When you are ready to start playing sounds to the stream, call **SoundEnableSampleStream()**. This routine grabs access to a physical DAC player device. Note that you will specify a **SoundPriority** for the stream, and that you will not be able to change this priority from sample to sample. Also note that if a sound is playing on the stream and you queue another sound to the stream, then the second sound will be queued behind the first. Thus, if your application wishes to play simultaneous sounds capable of interrupting one another, it should allocate more than one sound stream.

13.8

If there are more DAC streams than the sound device has DAC outputs, then the sound system will have to decide which sounds to play when too many play at once.

Sounds with a higher priority will oust those with lower priority. Note that the lower priority sound will not come back when the more important sound is finished—thus, a short alert noise might cause a few seconds of sampled speech not to play at all. If your application will be playing sampled sounds at more than one priority, you may thus want to make sure that the lower-priority sounds are broken up into small pieces.

If a sound wants to play, but all the available DAC outputs are playing higher-priority sounds, then the sound will not play.

To actually play a sound to your enabled sound stream, call **SoundPlayToSampleStream()**. This routine takes a pointer to the sound data, (which should be in a locked area of memory). The sound data will be copied to the DAC stream's buffer, and you may thus free the sound data after you have played it to the DAC stream. The sound will be queued at the end of the stream's data. The sound device's sound driver will read from the stream and will eventually play the sound.

When you are done playing sounds to a stream, you should free up the stream by calling **SoundDisableSampleStream()**.

If you wish to enable a new DAC stream with the same sound token, but with a different **SoundPriority**, you may do so using **SoundEnableSampleStream()** (You should call

SoundDisableSampleStream() on the token before enabling it in this way). You should eventually call **SoundDisableSampleStream()** again in this case.

Once the DAC stream has been disabled, you may free up the sound token by calling **SoundFreeSampleStream()**. You should call this with all sound tokens before your application exits.

13.9

13.9 Grabbing the Sound Exclusive

```
SoundGetExclusive(), SoundGetExclusiveNB(),  
SoundReleaseExclusive()
```

If your sound device has multiple voices and/or DAC outputs, then more than one application may play sound at a time. If the highest priority sound does not use all of the card's voices, then the next priority sound will be allowed to use them.

Most of the time, this is the behavior you want. This allows you to have sounds going on in the background and foreground at the same time. However, if your application will be creating some sound so important that no other application should be allowed to interrupt it, you may grab exclusive access to the sound routines.

SoundGetExclusive() grabs a semaphore associated with the sound library (performs a "P" operation). Your thread thus makes it known that it wants sole access to the sound library, and will wait, if necessary, for any other thread which has grabbed sole access to finish. It will then grab exclusive access to the sound library's low level routines. Until you release the exclusive, no other thread will be able to use the **SoundAllocMusic()**, **SoundAllocMusicStream()**, **SoundAllocMusicNote()**, **SoundPlayMusic()**, **SoundPlayToMusicStream()**, **SoundStopMusic()**, **SoundStopMusicStream()**, **SoundReallocMusic()**, **SoundReallocMusicNote()**, **SoundAllocSampleStream()**, **SoundEnableSampleStream()**, or **SoundPlayToSampleStream()** routines.

The **SoundGetExclusiveNB()** routine also locks the sound library for exclusive access. However, if any other thread has already grabbed the

exclusive, **SoundGetExclusiveNB()** won't wait; it will just grab the exclusive away anyhow.

To release the exclusive, call **SoundReleaseExclusive()**. This will allow sounds to enter the queue and will allow another thread which has called **SoundGetExclusive()** to grab the exclusive.

13.10 Simulating Musical Instruments

13.10

In addition to the standard instruments provided by the system, you may define instruments of your own. To do so, you must first find out what sort of instrument format the user's sound driver expects. You will then pass a pointer to the appropriate data structure.

To find out what sort of data structure the sound driver expects, call **SoundSynthDriverInfo()** and note the value returned at the *format* pointer argument.

13.10.1 Acoustics In Brief

Sound may be thought of in terms of waves. A pure tone manifests as a sine wave. The higher the frequency of the wave (as measured in Hertz), the higher the pitch of the generated sound. A piccolo would generate a wave with a high frequency; a tuba would generate a low frequency. The amplitude of the wave translates to the sound's volume, or loudness. Cannons generate sound waves with great amplitude.

A tone consists of a sound of one frequency. Full, rich sounds are created by making sound at more than one frequency. If you generate sound at multiples of the base frequency, you get a good broad, rich sound and the frequencies beyond that of the note itself are known as "harmonic" frequencies. However, the volume levels should be lower at these harmonic frequencies than at that of the note itself. Together, these frequencies can form the sounds of chords. A rich instrument such as an organ will make sounds on many frequencies

other than that of the main note. A violin, with a very pure tone, will generate almost no sound on frequencies other than that of the note being played.

13.10.2 Simple Instrument Description

13.10

The simple instrument description allows you to specify a way for less powerful devices to emulate different musical instruments. You describe three volume levels, each of which can be between 0 and 255. The fundamental tone will play at the first volume level; the second volume will be used for playing the first harmonic frequency; the third volume for playing the second harmonic.

Simple instruments have another characteristic: you can specify a noise component. By means of noise, you impart sibilance to the sound, giving it a “breathy,” sometimes hissing quality. A sound without noise will seem crisp by comparison. At this time, the simple sound format supports three kinds of noise:

NO_NOISE The lack of noise. Use for crisp, acute sounds. This value is guaranteed to be zero.

WHITE_NOISE
Noise evenly distributed across all frequencies.

METAL_NOISE
Noise only at the lower frequencies—you may think of this as standard white noise which has gone through a low-pass filter. Noise generated will have a lower tone, and have a less hiss-like quality than those with white noise. If cymbals have white noise, then drums have metal noise.

To describe an instrument with no noise component, specify the three volume levels and leave the noise component as zero.

```
CTIEnvelopeFormat BrokenTuba =  
    {255,          /* Primary: full strength */  
     64,           /* First harmonic: 25% strength */  
     10,           /* 2nd harmonic: 10% strength */  
     0};          /* No noise */
```

To describe an instrument with noise, you must specify a type and a degree of noise (encoded as two bits describing what sort of noise OR'd together with six bits describing the degree of noise, if any):

```
CTIEnvelopeFormat FunkyPiano =
    { 255, 32, 17, ((NT_WHITE_NOISE << 6) | 10) };
CTIEnvelopeFormat NoisyDrum =
    { 0, 0, 0, ((NT_METAL_NOISE << 6) | 63) };
```

13.10

13.10.3 Advanced Description

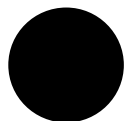
The advanced instrument description data structures were set up with the SoundBlaster card in mind; for detailed information (i.e. timing numbers) about these fields, you should purchase a copy of the SoundBlaster Development Kit and read the chapter about Programming the FM Chip. However, you may first want to read the documentation below for an introduction to some of the involved concepts:

- ◆ As with the simple instrument format, you'll specify the richness of your instrument, how its sound shows up at frequencies other than the base frequency. This is done using a technique known as "Frequency Modulation."
- ◆ You will also work with the instrument's loudness, or sound level. You may specify that some instruments are louder than others. You may also specify how the sound level changes over time.

Frequency Modulation

Frequency modulation allows you to specify how an instrument's sounds will appear at its harmonic frequencies. You can even use it to set up sound at frequencies other than the harmonics. The sound device does this by means of a "modulating" frequency.

The sound device actually keeps track of two frequencies. The *carrier* frequency, corresponding to the note being played, can be thought of as constant for a given note being played. This frequency may be input to a sound wave generator, and the resulting wave to a speaker, resulting in a pure tone. A frequency modulating (FM) system does not just use a constant frequency, but actually adds a wave signal to the carrier frequency to achieve



a frequency which changes over time. Consider a siren—its frequency changes over time. However, in an FM system, these changes come much more quickly.

The wave signal which gets added to the carrier frequency is known as the *modulating signal*, and you may change the nature of your instrument's harmonics by changing the characteristics (amplitude, frequency) of this modulating signal.

13.10

The input to the speaker—this wave which is constantly changing frequency, thanks to the modulating signal—is called the *output signal*.

The net effect of this is that the carrier signal will contain several frequencies. In fact, those frequencies will be spaced apart by a constant distance—the frequency of the modulating signal. If the modulating frequency is the same as the frequency of the note being played, then this means that the carrier will contain sounds at the note and its harmonic frequencies.

Thus, you may use the modulating signal to set up sound on the harmonic frequencies, just as you did with the simple instrument description. The FM system is a bit more adaptable in that by making the modulating signal some multiple of the main note, you can create sound at every other harmonic frequency, or possibly at each harmonic and any number of frequencies between each harmonic.

Furthermore, you can change the strength of the sound at the various frequencies by changing the amplitude, or level, of the modulating signal.

Changing Sound Levels

Notes in a sound buffer have a single volume, or loudness. However, when you play a note on a musical instrument, the volume level is not constant. When you hit a drum, the sound is instantaneous—it doesn't matter if you pull the drumstick back a moment or a minute later, all the sound is generated in a single instant. When you press a piano key, a string inside the piano will start vibrating, producing sound. This string will continue to produce sound (though it will grow gradually quieter). Thus, to truly differentiate instruments, we need a model for an instrument's loudness over time. This change over time is known as an instrument's *envelope*.

The advanced instrument description uses a standard ADSR model of an instrument's envelope. That means it breaks a sound's life time into four parts: an Attack, Decay, Sustain, and Release. Each instrument has a value for each stage of its envelope, and will determine the length of that stage of the life-span. Also, you may use an envelope type to specify whether the sound should continue for as long as the voice is on (like an organ), or should die off automatically (like a drum). In the GEOS model, attack, decay, and release are all slopes; sustain is a sound level, expressed as a fraction of the note's volume.

13.10

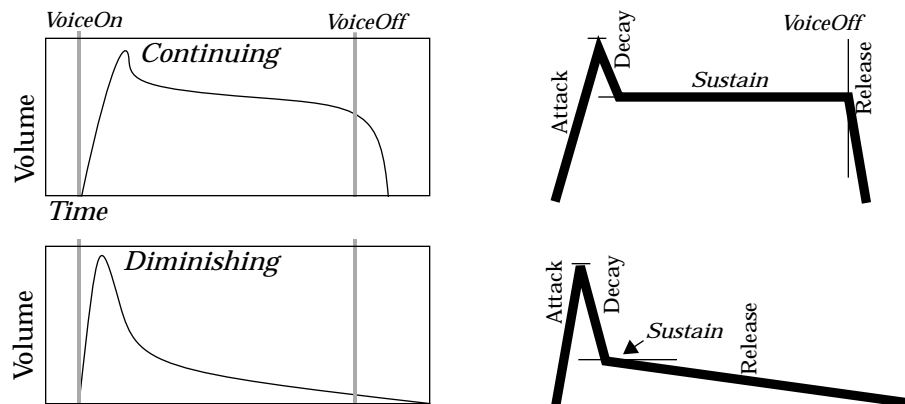


Figure 13-2 Modeling with the ADSR envelope.

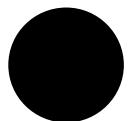
For a continuing (organ-like) instrument, the sound level will progress thus:

Attack The sound level will build up to that specified for the note (e.g. DYNAMIC_MF). The higher the attack value, the more quickly this will occur. Note that if the voice is turned off during this stage, the sound will go immediately to the release stage (This simulates that the instrument was never allowed to reach peak volume, as if an organist had tapped an organ key instead of keeping it depressed).

Decay The sound level will decrease at a rate determined by the decay value. The higher the decay, the more quickly the sound level will fall off. If the voice is turned off during this stage, the sound will immediately enter the release stage.

Sustain Level

The sound level will fall off at the decay rate until it has fallen



to the sustain level. This sustain level is expressed as a fraction of the note level, so you might, for example, specify that the sound level should stop decaying once it has fallen 25 percent. The sound level will remain at the sustain level until that note ends (i.e. the voice is turned off).

Release

The voice will then decrease at a rate determined by the release value. The higher the release, the more quickly the sound level will fall off. The sound will continue to fall until inaudible.

13.10

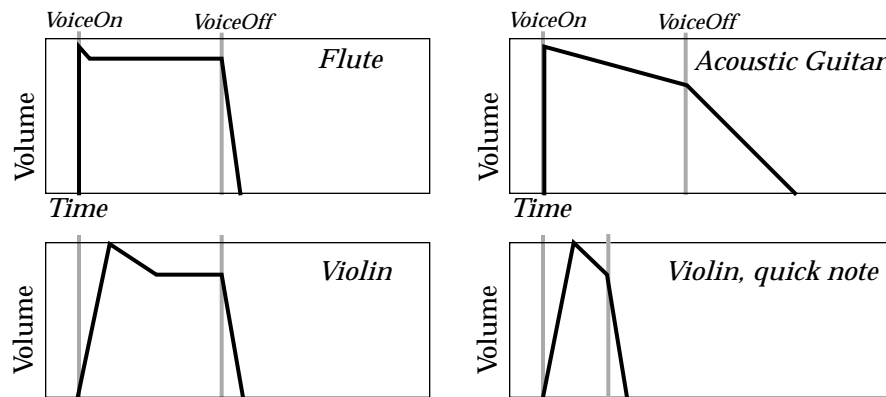


Figure 13-3 Continuing Instrument Envelopes

The flute shows a typical continuing instrument envelope, made up of four segments. The guitar has a small decay value, and thus the volume has not fallen to the sustain level when the voice goes off, and the volume starts falling at the release rate. You can see how the Violin can be similar to either of these patterns, depending on when the voice is turned off.

For a diminishing (drum-like) instrument, the sound level progresses:

Attack

The sound level will build up to that specified for the note (e.g. DYNAMIC_FF). The higher the attack value, the more quickly this will occur. If the voice is turned off partway through the attack the sound will immediately enter release, never reaching full volume, and skipping the decay stage.

Decay

The sound level will decrease at a rate determined by the decay value. The higher the decay, the more quickly the sound level will fall off. If the voice is turned off during this time, the sound will go into release without waiting to reach the sustain level.

Sustain Level

The sound level will fall off at the decay rate until it has fallen to the sustain level. This sustain level is expressed as a fraction of the note level, so you might, for example, specify that the sound level should stop decaying once it has fallen 25 percent.

Unlike a continuing instrument, the sound level will immediately move on to the release stage, regardless of whether the voice has been turned off.

Release

The voice will decrease at a rate determined by the release value. The higher the release, the more quickly the sound level will fall off. The sound will continue to fall until inaudible.

13.10

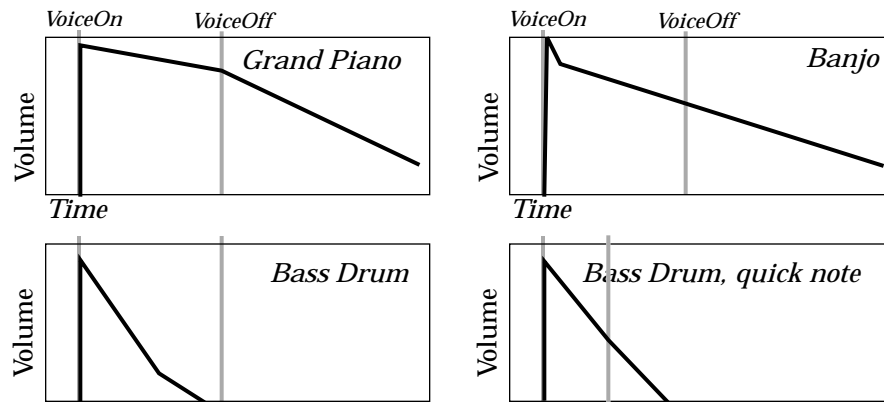


Figure 13-4 *Diminishing Instrument Envelopes*

The piano demonstrates the envelope of a typical diminishing instrument envelope.

In case that isn't enough to keep track of, consider the following: You may also specify an ADSR envelope for your modulating signal. This actually comes in handy—many instruments attack with a rather broad sound, but will sharpen to a single tone while sustaining or releasing. By setting up a modulator envelope so that the modulator signal is strong while the main signal is attacking but weak when the modulator is passed this point, you may simulate this behavior.



Pieces of the Advanced Instrument Format

Now that you understand the main forces governing how your sound will be modeled, you are ready to use the **SBIEnvelopeFormat** data structure. This structure has fields for the following values:

SBIEF_carTimbre: Carrier Timbre

This field contains several effects which may be applied to the carrier wave.

13.10

Amplitude Tremolo

This flag requests that a wave signal be added to the carrier's sound level, producing a sort of "tremolo" effect.

Frequency Vibrato

This flag requests that a wave signal be added to the carrier's frequency, producing a "vibrato" effect. Note that the modulating signal adds a wave to the carrier's base frequency, but vibrato is separate and different: The modulating frequency is dependent on the note frequency, but the vibrato wave's frequency is a constant; that is, you may control the modulating amplitude, but the vibrato's amplitude is fixed.

Envelope Type

This flag specifies the envelope type of the instrument—continuing or diminishing.

Key Scaling Rate

Some instruments, especially stringed instruments, tend to produce shorter notes at the higher frequencies. This flag allows you to simulate this type of instrument—at the higher frequencies, the carrier's ADSR values will be boosted, so that the note's life span will be shorter.

Frequency Multiplier

You may apply a multiplying factor to the carrier frequency. If this factor is greater than one, then the instrument will tend to play notes at a higher pitch—if the factor is two, for example, the instrument will play one octave higher than normal. If the factor is one-half, then the instrument will play one octave lower than normal.

SBIEF_carScaling: Carrier Scaling

This field contains information about the carrier's basic scaling—its basic loudness, before the effects of the ADSR envelope are applied.

Key Scaling Level

Some instruments tend to play high-pitch notes quietly. You may set this value to specify how much the sound level should decrease when playing high frequencies: the higher the number, the faster the decrease. Zero means you want no decrease.

13.10

Total Level This is the base loudness of your instrument, but be careful—the higher the number, the quieter the instrument. If this value is zero, then you want your instrument to sound as loud as possible.

SBIEF_carAttack: Carrier Attack/Decay

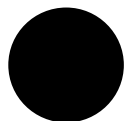
This field contains the attack and decay portions of the carrier's ADSR envelope.

Attack This is the instrument's "attack," its rate of sound level increase from none to the note's sound level. This time could range from something measured in seconds to nearly instantaneous. Increasing the attack level by one effectively doubles the attack rate. Percussive instruments tend to have high attacks, bowed and wind instruments tend to have medium to low attacks. Not many instruments have attack levels towards the bottom of the scale, and the higher attack levels are almost exclusively the province of special effects.

Decay This is the instrument's "decay," its rate of decrease from the note's sound level to the sustain level. This rate of decay is about one fifteenth of an Attack of the same level (i.e. decays are on a slower scale than attacks). Instruments which get quiet soon after being activated will have high attacks (especially diminishing-type instruments of this nature). A chime would have almost no decay, while a drum would have a medium-level decay—almost no instruments will have a decay more than halfway up the scale.

SBIEF_carSustain: Carrier Sustain/Release

This field contains the sustain and release portions of the carrier's ADSR envelope.



13.10

Sustain This is the instrument's "sustain." It is a fraction, which will be multiplied by the note level to find the true sustain level. Continuing-type instruments will stay at this sound level until they encounter a voice-off command. Diminishing instruments will not (proceeding immediately to release).

Release This is the instrument's "release," its rate of decrease from the sustain level to silence. This rate will be the same as a decay of the same level (i.e. decays and releases happen at the same rate). In a piano, a damper hits the string when you release the key, stopping sound quickly; a piano has a high release. Bells, which continue vibrating for a long time after being hit, have a low release rate.

SBIEF_carWave: Carrier Wave

Most natural sounds correspond to a sine wave. However, you may request another type of wave.

Wave Type There are four choices of wave:

- ◆ Sine wave. The standard $y = \sin(x)$ curve.
- ◆ Truncated sine wave: $y = \text{MAX}(0, \sin(x))$. This results in something like a square wave.
- ◆ Absolute value sine wave: $y = |\sin(x)|$. This results in something approaching a triangle wave. This results in a somewhat "tinny" sound.
- ◆ Chopped sine wave: $\{ 0 \leq x \leq 90, 180 \leq x \leq 270: y = |\sin(x)| ; 90 \leq x \leq 180, 270 \leq x \leq 360: y = 0 \}$. This wave is similar to a sawtooth wave.

SBIEF_modTimbre: Modulator Timbre

This field contains a number of effects which you may apply to the modulator signal. Thus, applying a tremolo effect results in a sound which wobbles in breadth over time—the amplitude of the modulator is being changed, not that of the final signal.

Amplitude Tremolo

This flag adds a sine wave signal to the modulator's amplitude. You will recall that the modulator's amplitude determines the "breadth" of the sound—how far it will extend into other frequencies. Applying a tremolo effect to this value will cause the sound to vary in breadth over time.

Frequency Vibrato

This flag adds a sine wave signal to the modulator's frequency.

You will recall that the frequency determines the distance between frequencies at which the carrier signal will have sound. If you have set up the modulating signal to have sound appear at the harmonics, then applying this effect will cause the supplement sounds to warble around the harmony level.

Envelope Type

This flag specifies the envelope type of the modulator signal's envelope (continuing or diminishing).

13.10

Key Scaling Rate

This flag boosts the ADSR values of the modulator signal when the note is at a high pitch. You would apply this factor if an instrument's breadth dissipated faster at high frequencies.

Frequency Multiplier

This is the field which gives you your main control over the modulating signal's frequency. You may apply a multiplying factor to the modulating signal's frequency. If the modulating signal's frequency is the same as the carrier frequency, then it will create harmonic frequencies. If the modulating frequency is a multiple of the carrier frequency, then it will generate harmonic frequencies, and others in between. If the modulating frequency is a fraction of the carrier frequency, then it will generate some but not all harmonic frequencies. Percussion instruments, whose sounds tend to manifest as quick bursts of noise, tend to have very high modulator frequency multipliers.

SBIEF_modScaling: Modulator Scaling

These fields allow you to control the amplitude of the modulating signal, and thus the breadth of the carrier signal.

Key Scaling Level

This field specifies that the amplitude should be lower when a high-pitch note is played. This means that higher notes will seem sharper, more piercing. The greater the value of this field, the more quickly the amplitude will fall off; if this value is zero, then the amplitude will be unaffected.

Total Level This field allows you to directly change the amplitude of the modulator signal—but be careful: the higher the value of this field, the lower the amplitude. The lower the value of this field, the broader the sound.



SBIEF_modAttack: Modulator Attack/Decay

These fields allow you to control the attack and decay portions of the modulating signal's ADSR envelope.

13.10

Attack This field determines how quickly the modulating signal will go to full amplitude—creating the broadest sound. This is often one or two levels higher than the carrier signal's attack level (resulting in an actual attack rate two to four times faster), resulting in a sound broadest during the carrier's attack period, characteristic of many instruments. Wind instruments tend to have a slow modulating attack.

Decay This field determines the rate at which the modulating amplitude will decrease until it reaches the sustain level. This value is normally kept low to maintain the instrument's breadth during and beyond the instrument's attack.

SBIEF_modSustain: Modulator Sustain/Release

These fields control the sustain and release portions of the modulating signal's ADSR envelope.

Sustain This determines the level to which the amplitude will be allowed to decay. If the modulating signal is continuing-type, it will remain at this sustain level until the voice is turned off.

Release This determines the rate at which the amplitude will decrease from the sustain level. The higher the value, the faster the sound will reduce to a pure tone.

SBIEF_modWave: Modulator Wave

You may apply any supported wave type in place of a sine wave for the modulator signal. This allows you to change the distribution of sounds over other frequencies.

Wave Type There are four choices of wave:

- ◆ Sine wave. The standard $y = \sin(x)$ curve.
- ◆ Truncated sine wave: $y = \text{MAX}(0, \sin(x))$. This results in something like a square wave.
- ◆ Absolute value sine wave: $y = |\sin(x)|$. This results in something approaching a triangle wave.
- ◆ Chopped sine wave: $\{ 0 \leq x \leq 90, 180 \leq x \leq 270: y = |\sin(x)| ; 90 \leq x \leq 180, 270 \leq x \leq 360: y = 0 \}$. This wave is similar to a sawtooth wave.

SBIEF_feedback: Feedback

This field allows you to change the way the modulator and carrier signals are combined.

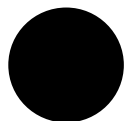
Feedback Type

Instead of adding the modulating signal to the note frequency, you can bypass all frequency modulation and just add the outputs of the “modulator” and “carrier” signals. If you set this flag, then both signals become, in effect “carrier” signals—both will generate a note, not a breadth.

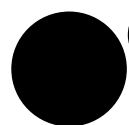
13.10

Feedback Level

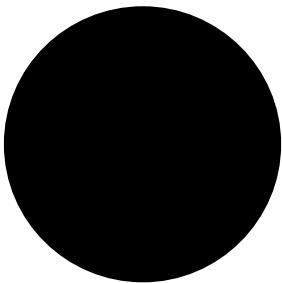
Whether it's being used as a modulator signal, or just as another “carrier,” the modulator can feed back to itself. This means that some fraction of its output in a given tick may be added to its frequency input.



13.10



Handles



14.1	Design Philosophy	537
14.2	The Global Handle Table	539
14.3	Local Handles	539





One of the most important data types in GEOS is the *handle*. There are many different types of handles serving many different purposes. However, they all have one or both of these traits: they provide an unchanging reference to a changing (or moving) thing, and they allow geodes to access and manipulate indirectly certain structures which are maintained as internal to the kernel.



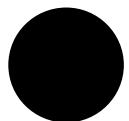
This chapter is an overview of the use of handles in GEOS. Before you read this chapter, you should be familiar with PC architecture; in particular, you should understand the segment:offset scheme of addressing memory (discussed in section A.2.1 of Appendix A). This section sketches some basic principles of memory usage and data manipulation. These topics are covered in greater detail in other chapters in this book, particularly in the chapters on Memory Management (“Memory Management,” Chapter 15) and Local Memory management (“Local Memory,” Chapter 16).

14.1

14.1 Design Philosophy

GEOS handles play many different roles. The most fundamental of these is memory access. GEOS moves memory in the global heap and swaps memory to the disk. This makes memory available as needed and means that the total memory being used by geodes can be many times larger than the physical memory in the computer; however, it means also that geodes need some way to keep track of their memory. In addition, the system needs to perform memory “housekeeping,” doing such things as freeing memory when the geode which owns it exits, making sure that memory is only locked by appropriate geodes, noting which memory sections can be swapped to the hard disk, etc. This means that the kernel needs to maintain a data area for each block of memory. Although the memory itself may move around, the data about the memory should remain in the same place so the kernel can access it.

The solution to both difficulties is the *handle table*. The table contains entries for each block of memory. These entries keep track of the block, noting such information as where the block is in the global heap, whether it has been swapped to the disk, who its owner is, etc. The handle table does not move in



memory. Each entry is referenced by a *handle*, which is simply the address of the table entry. When a geode needs access to a block, it can call a memory-management routine and pass the handle associated with that block. The memory manager dereferences the handle to find the data about the block and then takes any appropriate steps.

The handle table is useful for many other things. Often a geode will need to perform an action on something which is managed by the kernel. For example, the kernel keeps track of the different disk volumes it has seen. A geode may need to access a disk but not want to keep track of the volume name; or it might want to find out details (such as the size available) of a given disk. Similarly, it might want to find out how large a file which it has opened has grown. All of this information is managed by the kernel, but applications may need to access it or change it. Therefore, each of these things is given an entry in the same handle table as the memory blocks; geodes can reference these things by their handles, the same way they do memory blocks.

The handle mechanism is useful in other places. Sometimes a geode will want to subdivide a block of memory into smaller parcels (or *chunks*). GEOS provides a local memory library which implements this functionality. These chunks can be moved around within a block, so applications access them by handles. The handle in this case is an offset to a handle table within that block; dereferencing the handle gives the address of the chunk. Similar techniques are used to divide Virtual Memory files. The principle is the same as with global memory handles: an unchanging handle is used to find the address of a movable thing.

There are certain things all handles have in common. They are all sixteen bits long, allowing them to fit in an 80x86 register. They also all specify addresses. The address is of an entry in a handle table; this entry contains information about the thing whose handle this is. However, handles can be divided into two basic groups:

- ◆ Those in the global handle table.
The global handle table is kept in main memory below the global heap. The table is never accessed directly by geodes. Instead, geodes pass the handle as an argument to kernel routines that access the handle table.
- ◆ Those in local handle tables.
Such handles are offsets into handle tables which may be stored in

memory blocks or VM files. These handles persist as long as the entity containing the handle table does.

14.2 The Global Handle Table

Many handles refer to things which are managed by the kernel. These things include global memory blocks, disk volumes, files, and many other things. Each of these things has an entry in the global handle table. Only the GEOS kernel may directly access the global handle table. If a geode wants to access one of these things, it must call a system routine and pass the thing's handle.

14.2

Global handles may not be saved across sessions of GEOS. The global handle table has to be rebuilt each time GEOS is launched. For example, when GEOS shuts down, all open files are closed. When an application restores from state, it may choose to reopen a file; it will then be given the file's handle, which will almost certainly be different from the file's handle during the previous session.

Sometimes several threads will be using the same global handle, and they will need to synchronize their access to it. They can arrange this by using the **HandleP()** and **HandleV()** routines. These routines are most often used to synchronize access to a global memory block; for this reason, they are documented in section 15.3.6 of "Memory Management," Chapter 15.

14.3 Local Handles

Many handles are not kept in the global handle table. For example, a block of memory may be declared as a "local-memory (LMem) heap." An LMem heap can contain many different *chunks* of data; the LMem manager moves these chunks around to make room for new ones. Each chunk is referenced via a *chunk handle*. (For more details about LMem heaps, see "Local Memory," Chapter 16.)

Local handle tables are useful when a handle needs to persist across GEOS shutdowns. For example, data in VM files is accessed by its block handle.

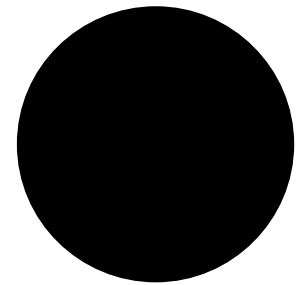
These handles therefore need to remain the same every time the file is opened.

Some local handle tables are intended to be directly accessible to geodes. For example, LMem Chunk handles are simply offsets into the block containing the LMem heap; the address specified by that offset contains another offset value, which is the offset to the chunk. This scheme is part of the LMem API. Geodes which are written in assembly generally look up chunk handles themselves, instead of calling a system routine to translate the chunk handle into an address. (Goc code generally dereferences a chunk handle with a call to **LMemDeref()**.) Some other local handle tables are intended to be opaque to geodes. For example, applications treat VM block handles as opaque 16-bit tokens; to dereference a VM handle, a geode must pass it to an appropriate system routine (such as **VMLock()**).

14.3

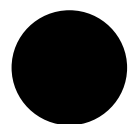
Local handles are used by many different libraries; among them are the LMem, VM, Database, and Cell libraries. The details differ with each library; for more information, see the appropriate chapter. An object-descriptor is a combination of two handles: the handle of the global memory block containing the object, and the chunk handle of the object itself. For more information, see “GEOS Programming,” Chapter 5.

Memory Management



15

15.1	Design Philosophy	543
15.2	The Structure of Memory	544
15.2.1	Expanded/Extended Memory.....	544
15.2.2	Main Memory.....	545
15.2.2.1	Blocks and Handles	545
15.2.2.2	Enabling Block Access	546
15.2.2.3	Types of Blocks.....	546
15.2.2.4	Maximizing Free Space in Memory	548
15.2.2.5	Block Attributes.....	550
15.3	Using Global Memory	552
15.3.1	Memory Etiquette	552
15.3.2	Requesting Memory.....	554
15.3.3	Freeing Memory.....	555
15.3.4	Accessing Data in a Block	555
15.3.5	Accessing Data: An Example	556
15.3.6	Data-Access Synchronization	558
15.3.7	Retrieving Block Information	561
15.3.8	The Reference Count.....	563
15.4	malloc()	564





Managing memory in a multitasking system is complex because many different entities—the kernel, libraries, and applications—are all trying to use the same memory space. To ensure that each gets all the memory it needs, without having to worry about what other memory users are doing, GEOS provides a memory manager.



This chapter describes the memory management system used by GEOS. It also describes the routines applications can use to get raw memory. Before you read this chapter, you should be familiar with the use of handles (see “Handles,” Chapter 14).

15.1

15.1 Design Philosophy

Some systems make each application maintain and manage its own memory. In a multitasking system, this is not only difficult and time-consuming but also potentially dangerous, as applications can easily start trashing memory used by other processes. GEOS protects and relieves applications by providing comprehensive memory management.

The GEOS memory management system is designed to meet rigorous demands. Some of the requirements are

- ◆ Independence of memory location
Ideally, an application should not have to keep track of the address of each of its data items; rather, the memory management system should allow the application to reference its memory virtually.
- ◆ Hardware independence
Applications are much easier to write and maintain if they can ignore hardware specifics. An application should be able to specify its memory requirements in generic terms (“I need a package this big which behaves in this way”) and let the memory manager worry about where it comes from.
- ◆ Efficient use of memory
A good operating system should be able to rearrange memory in order to gather as much space as possible. It should also be able to discard certain



data or copy it to mass-storage devices (like a hard disk) if more memory is needed.

- ◆ **Management of Shared Data**

Applications should be able to share common data or code. In a multitasking system, proper synchronization of shared resources is essential to maintain data integrity.

15.2

The GEOS memory manager meets all of these needs. Applications often take advantage of the memory manager without trying; for example, the memory manager swaps methods into memory when messages need to be handled. Applications can also request memory at run-time, either by requesting raw memory from the memory manager, or by creating Virtual Memory files through the VM library (see “Virtual Memory,” Chapter 18).

15.2 The Structure of Memory

The GEOS memory manager currently uses real mode memory addressing, and is designed to run on PCs with as little as 512K RAM. Typically, between 30K and 160K of that is occupied by DOS. The remaining RAM is used for the global handle table and the global heap (described below). The GEOS kernel is kept in the global heap.

Because of the constraints of real-mode memory addressing, GEOS uses memory in segments of at most 64K. Each segment may be subdivided and parceled out to fill memory requests from applications. Sometimes an application will need a seemingly contiguous section of memory which is larger than 64K; in these situations, it should use the HugeArray routines (see section 18.4 of chapter 18).

15.2.1 Expanded/Extended Memory

While GEOS is designed to run on a standard system, it makes efficient use of expanded and extended memory, if it is available. GEOS treats extended memory as a fast and convenient RAM disk, swapping blocks into the extended memory rather than out to a slower disk drive. This incurs none of

the usual overhead of a RAM disk such as a directory because it is treated as normal memory. Expanded memory is used in a similar manner.

15.2.2 Main Memory

Memory available to applications is organized in a data structure called the Global Heap. The size of the global heap may vary from machine to machine and can change from session to session, but during a single execution of GEOS, the heap size remains constant. Usually, the global heap occupies approximately 450K bytes of a 640K system.

15.2

When an application requests memory, it is allocated a “block” on the heap. Blocks may be of almost any size but may not be larger than 64K. (However, the heap is most efficient when blocks are 2K-6K in size; see “Memory Etiquette” on page 552.) Every block is allocated a unique handle (described below) from the Handle Table; by calling a memory manager routine, the application can translate a handle to a conventional pointer.

When GEOS shuts down, all the blocks on the global heap are freed, even if they are locked or fixed. If an application will need to store the data over a shutdown, it should make the block part of a VM file, which it can then reopen when it restores from state. The GEOS kernel attaches object blocks to system VM files and takes care of storing them to state and restoring them when GEOS restarts.

15.2.2.1 Blocks and Handles

GEOS segments memory into blocks. A block is simply a number of contiguous bytes on the global heap in which code or data may be stored. Any given block may be of any size but must be less than 64K, due to the segmented addressing scheme used by the 80x86 processors. A block’s size is determined when it is allocated—the process that requests the memory must specify the desired size.

To facilitate efficient memory usage and to present all applications with enough memory to function, blocks are dynamic in nature. This means that unless a block is fixed or has been locked (see section 15.3.4 on page 555), there is no telling its precise address on the heap, or indeed whether it is on

the heap at all. For this reason, applications are not given the address of an allocated block; rather, they are given the block's handle.

Memory handles are indexes into the global handle table. Geodes may not access the table directly. When they want to access the memory indicated by a handle, they pass the handle to the memory manager and are returned the segment address of the block's current location on the heap.

15.2

In addition to storing the actual address of the block, the handle table entry records the block's attributes, such as whether the block is discardable, swappable, or fixed. The memory manager uses all these attributes to manipulate the block.

15.2.2.2 Enabling Block Access

Dynamic memory, while providing significant benefits, poses one major problem: What happens if a block is accessed while being moved, swapped, or discarded? GEOS responds to this problem with the implementation of a system for locking and unlocking blocks. When a block is locked on the heap, the Memory Manager may not move, swap, or discard it until it is unlocked. This locking mechanism allows applications to gain control over their memory during periods of access and relinquish it when it is not in active use. Applications, however, should not leave blocks locked for extended periods as this may interfere with heap compaction.

When a process wants to use a block, it instructs the memory manager to lock the block by calling **MemLock()** (see page 556). The application passes the handle of the block, and the memory manager locks the block and returns a pointer to the block's area in the global heap. While a block is unlocked, the memory manager can, depending on the block's category, move the block on the heap, swap it to disk or extended/expanded memory, or discard it altogether.

15.2.2.3 Types of Blocks

When a geode requests memory, it may specify how that memory is to be treated by the memory manager. The memory request includes a set of **HeapFlags** (see page 551) which specifies how and when the block can be moved. Broadly speaking, memory blocks can be divided into four categories:

◆ Fixed

A fixed block has the flag `HF_FIXED` set to one, and `HF_DISCARDABLE` and `HF_SWAPABLE` set to zero. A fixed block is allocated on the global heap and stays exactly where it was created until it is freed. An application can therefore reference data within a fixed block with normal pointers; the application does not need to use the memory manager to translate a handle into a pointer. Accessing fixed blocks is very quick since the memory never needs to be read from disk and the handle does not need to be dereferenced. However, since the memory manager cannot relocate fixed blocks, they tend to degrade the memory manager's performance; therefore, applications should generally use small amounts of fixed memory. Note that if a fixed block is resized, it may be moved on the global heap; therefore, after resizing a fixed block, an application should reset its pointers within the block by dereferencing the block handle.

15.2

◆ Moveable

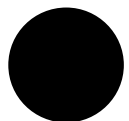
A moveable block has the flag `HF_FIXED` set to zero. When a moveable block is not in use (i.e. unlocked), the memory manager may move it within the global heap. This lets the memory manager keep the global heap compacted (see section 15.2.2.4 on page 548). Accessing moveable blocks is slightly slower than accessing fixed blocks since the application needs to call the memory manager to lock the block and translate the memory handle to a pointer. In addition to being moveable, a block may also be swapable and/or discardable (as described below).

◆ Swapable

A swapable block has the flag `HF_FIXED` set to zero and `HF_SWAPABLE` set to one. If a block is marked "swapable," the memory manager has the option of swapping it out of main memory (either to the hard disk, or to extended or expanded memory) when it is unlocked. This keeps space on the global heap free for other requests. If an application tries to lock a block which has been swapped out of main memory, the memory manager will copy the entire block back to the global heap, then lock it normally. By using swapable memory, applications can potentially use much more memory than the roughly 450K available in the global heap. However, accessing an unlocked swapable block can potentially be much slower than accessing a non-swapable block, as the block may have to be copied in from the disk.

◆ Discardable

A discardable block has `HF_FIXED` set to zero and `HF_DISCARDABLE` set to one. Sometimes it doesn't make sense to copy a block to the disk. For example, many memory blocks contain code or unchanging data which is



read in from the disk. It would make no sense to swap these blocks back to disk, since the data is already there. Instead, one can mark a block “discardable.” This indicates that when the block is unlocked, the memory manager can, at will, discard the block. If a process attempts to lock a block which has been discarded, the memory manager will return an error. The application can then “re-allocate” memory for that block (see **MemReAlloc()** on page 554) and copy the data back from the disk. (The system takes care of reloading discarded code resources as necessary.) A block can be both discardable and swappable (indeed, discardable blocks are usually swappable). If a block has both **HF_DISCARDABLE** and **HF_SWAPABLE** set, the memory manager can either swap the block to extended/expanded memory or discard it; it will not swap the block to the disk.

Fixed blocks must be declared as such when they are allocated, and they remain so until they are freed. However, non-fixed blocks may become or cease to be discardable or swappable after they are created. To enable or disable these characteristics, call the routine **MemModifyFlags()** (see page 562).

15.2.2.4 Maximizing Free Space in Memory

Moveable, swappable, and discardable blocks are allocated from the top of the heap using a first-fit method. Fixed blocks are allocated from the bottom of the heap. If there is not enough contiguous free memory to satisfy an allocation request, the memory manager attempts to shuffle moveable blocks in order to place all free memory together in one large mass.

This shuffling is called heap compaction. If the free space resulting from compaction still is not enough, blocks are discarded or swapped to liberate more free space, and the heap is again compacted. Because of the multitasking nature of GEOS, compaction occurs in the background and is invisible to both the user and applications. (See Figure 15-1 on page ● 549.) The memory manager will also periodically compact memory during periods of low activity; this helps insure that there will be memory instantly available for a sudden large demand (e.g. when an application is launched).

The compaction is not arbitrary. The kernel decides which blocks to swap or discard based on recent usage patterns. This means, for example, that if you haven't used routines in a code resource for a while, that resource is more likely to be discarded than the resources you've accessed recently. (For this

reason, geodes normally isolate their initialization code in one resource, which can be discarded later.)

One can see from Figure 15-1 that a block left locked for extended periods could interfere with heap compaction. Suppose, for example, that the moveable locked block in the middle of the heap were left locked during an application's entire execution. Essentially, this would cause the heap to be fractured into two subheaps, making compaction more difficult and possibly slowing the system down.

15.2

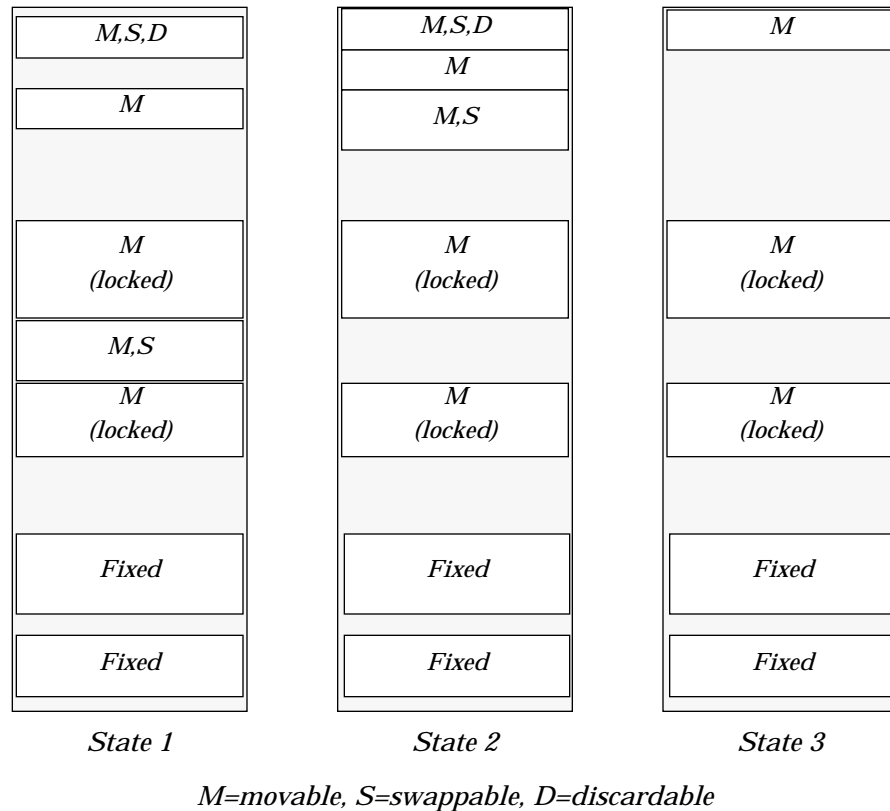
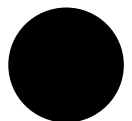


Figure 15-1 *Heap Compaction*

Compaction of the Global Heap: State 1 is prior to compaction, State 2 is after a single pass, and State 3 is after discardable blocks have been discarded and swappable blocks have been swapped. Note that the fixed blocks and locked moveable blocks are not relocated.



All compaction, swapping, and discarding are functions of the Memory Manager. Applications need only indicate how much space is needed and when space can be freed. Applications may also resize blocks at will; if necessary, the memory manager will compact the heap to accommodate the request.

15.2.2.5 Block Attributes

15.2

`HeapAllocFlags`, `HeapFlags`

Blocks are allocated with certain flags that help the Memory Manager manipulate memory efficiently. These flags can be found in the GEOS file **heap.h**, which should be included in all applications that plan to allocate memory dynamically with the memory manager routines.

The flags fall into two categories: those used when the block is allocated (stored in a record called **HeapAllocFlags**) and those used to describe the block as it is manipulated (stored in a record called **HeapFlags**).

The **HeapAllocFlags** record is used to determine what qualities the memory manager should give the block when it is first allocated. Some of these flags are also relevant when memory is being reallocated. These qualities include:

HAF_ZERO_INIT

Upon allocation, initialize data in block to zeros.

HAF_LOCK Upon allocation, the block should be locked on the global heap. Use **MemDeref()** (page 561) to get a pointer to the block.

HAF_NO_ERR

Do not return error codes; system error if block cannot be allocated. Use of this flag is strongly discouraged.

HAF_OBJECT_RESOURCE

This block is an object-block. This is set by the system *only*.

HAF_UI

If both **HAF_OBJECT_RESOURCE** and **HAF_UI** are set, the memory manager will set the block to allow the application's UI thread to manipulate objects in the block. This is set by the system *only*.

HAF_READ_ONLY

This block's data will not be modified.

HAF_CODE This block contains executable code.

HAF_CONFORMING

If this block contains code, the code may be run by a less privileged entity. If the block contains data, the data may be accessed or altered by a less privileged entity.

Once a block is allocated, it has certain properties that govern how the Memory Manager manipulates it. These properties are determined by the **HeapFlags**. The **HeapFlags** also contain data about whether the block has been swapped or discarded. These flags are stored in the block's handle-table entry, so they can be retrieved without locking the block. To retrieve the flags, call the routine **MemGetInfo()** with the flag

15.2

MGIT_FLAGS_AND_LOCK_COUNT. (See **MemGetInfo()** on page 562.) Some of the flags can be changed after the block has been allocated; for details, see **MemModifyFlags()** on page 562. The flags include

HF_FIXED The block will not move from its place in the global heap until it is freed. If this flag is *off*, the memory manager may move the block when it is unlocked. If it is *on*, the block may not be locked. This flag *cannot* be changed after the block has been allocated.

HF_SHARABLE

The block may be locked by geodes other than the owner. This flag can be changed with **MemModifyFlags()**.

HF_DISCARDABLE

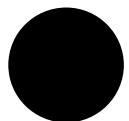
If the block is unlocked and space is needed, the memory manager may discard it. This flag can be changed with **MemModifyFlags()**.

HF_SWAPABLE

If the block is unlocked and space is needed, it may be swapped to expanded or extended memory or to the hard disk. This flag can be changed with **MemModifyFlags()**.

HF_LMEM

The block is a local-memory block, managed by the LMem module (see "Local Memory," Chapter 16). The flag is set automatically by **LMemInitHeap()**. It can be changed with **MemModifyFlags()**; however, an application should not change this flag.



HF_DISCARDED

The block has been discarded by the memory manager. Only the system can set or clear this flag.

HF_SWAPPED

The block has been swapped to extended or expanded memory or to the hard disk. Only the system can set or clear this flag.

15.3

15.3 Using Global Memory

When an application needs more raw memory at run-time, it can use the memory manager kernel routines to allocate, use, and free new blocks. An application can also create a Virtual Memory file to get memory with more advanced functionality, such as the ability to save to a disk file (see “Virtual Memory,” Chapter 18). However, one must understand the raw memory routines in order to use VM files.

If you will be storing small pieces of information in a block, you should create a local memory heap (a special kind of memory block). LMem heaps are also useful for storing arrays of information or for storing objects. For more information, see “Local Memory,” Chapter 16.

15.3.1 Memory Etiquette

The GEOS memory manager tries to fulfill every memory request. It does not enforce many rules on how the geodes should use memory. This gives each geode maximum flexibility, but it also means that the geodes have to take responsibility for their behavior; they must see to it that their memory usage does not degrade the system's performance.

First and foremost, applications should never use memory which has not been assigned to them by the memory manager. Unlike some operating systems (e.g. UNIX), GEOS does not enforce divisions between memory spaces. This allows GEOS to run on PCs which do not provide protected-mode access. It means, however, that the system will not stop an application from loading a bad address into a pointer, accessing that address, and thus writing to another memory space. An application should access only that memory

which the memory manager assigns to it directly (via calls to memory routines or data-space assigned at startup) or indirectly (through other libraries, such as the VM library). This also means that you should not save pointers to movable memory after unlocking the memory block. If you are not careful, you can accidentally access memory which no longer contains your data, with sometimes-fatal results.

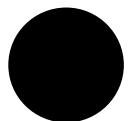
Another rule is to minimize the amount of memory you keep locked at a time. When memory is locked, the memory manager cannot move it on the global heap or swap it out of memory. This can result in the heap becoming fragmented, which in turn makes it more likely that memory requests will fail. Keep memory unlocked as much as possible. It's worthwhile to unlock memory even if you'll be locking it again very soon; this gives the memory manager a chance to compact the heap. If you have some data which you will need locked for long periods of time, it is best to put it in a fixed block, since fixed blocks are allocated from the bottom of the global heap and thus cause less fragmentation. Of course, you should free the fixed block as soon as possible.

15.3

Try to have as few blocks locked at a time as possible. Every locked block increases the danger that the heap will be unable to comply with a memory request. Try to organize your use of memory to minimize the number of blocks you will have to keep locked at a time.

In the same vein, try to keep the amount of memory you have on the global heap to a minimum. You should declare all of your non-fixed memory as swappable and/or discardable. Remember, you can use **MemModifyFlags0** to change these characteristics; during a timing-critical period, you could have a block set as non-swappable, and then make it swappable again as soon as timing becomes less important. If you use Virtual Memory files, the VM Manager does much of this for you.

Try to keep your memory blocks small. Although memory blocks can, in principle, grow to 64K, the memory manager is best suited for dealing with blocks in the 2K-6K range. If you need a contiguous data space which grows beyond 8K in size, you should use the Huge Array mechanism (see section 18.4 of chapter 18), which automatically (and almost transparently) divides a large data space across several smaller memory blocks. If you use many small data items, you should use the Database library, which automatically



distributes data items among different memory blocks, keeping each block near the optimum size. (See “Database Library,” Chapter 19).

15.3.2 Requesting Memory

`MemAlloc()`, `MemAllocSetOwner()`, `MemReAlloc()`

15.3

When you need a block of raw memory, you must use one of the kernel's memory allocation routines. You also must use kernel memory routines to change the size of a block or to reallocate space for a discarded block.

MemAlloc() creates a block and assigns a handle to it. The routine must be passed the size (in bytes) of the block to be allocated, along with the **HeapAllocFlags** and **HeapFlags** for that block. **MemAlloc()** will set the block's owner as the owner of the thread that called it. It will return the handle of the newly-allocated block.

MemAllocSetOwner() is the same as **MemAlloc()**, except that the caller explicitly sets the owner of the new block by passing the handle of the owning geode. Like **MemAlloc()**, it returns the handle of the new block. This is commonly used by drivers and shared libraries, which allocate memory owned by the geode which calls them. When the block's owner exits, the block will be freed, even if the block's creator is still running.

If you request a fixed block or pass the flag **HAF_LOCK**, the block will be allocated locked on the heap. However, the routine will still return just the memory handle. To translate this handle to a pointer, call the routine **MemDeref()**. **MemDeref()** is passed a block's handle and returns a pointer to the block (or a null pointer if the block has been discarded).

To change the size of a block, call the routine **MemReAlloc()**. This routine is also used to allocate memory for a block that has been discarded. The routine is passed the memory handle, the new size, and the **HeapAllocFlags**; it returns the block's memory handle. You can reallocate a fixed or locked block; however, the block may be moved on the global heap to satisfy the request. (This is the only way a fixed block can move.) As with **MemAlloc()**, you can request that the memory manager lock the block after reallocating it; you will then have to call **MemDeref()** to get the address of the block. Note that if the new size is *smaller* than the original size, the routine is guaranteed to succeed, and the block will not move from its current position. Reallocating a

block to zero bytes discards the block but preserves its handle; the block can then be reallocated later.

If the memory manager is unable to accommodate a request, it will return an error condition. The requestor can prevent error messages by passing the flag `HAF_NO_ERR`; this will result in a system error if the memory cannot be allocated. Passing `HAF_NO_ERR` is therefore strongly discouraged.

15.3.3 Freeing Memory

15.3

`MemFree()`

When you are done using a block, you should free it. Every block takes up space in the global handle table; if blocks are not freed, the handle table may fill up, causing a system error. Furthermore, non-swappable, non-discardable blocks take up space in the global heap until they are freed.

To free a block, call **`MemFree()`** and pass the handle of the block to be freed. The block will be freed even if it is locked. Therefore, if the block can be used by other threads, you should make sure no other thread has locked the block before you free it.

You can also set a reference count for a block. When a block's reference count drops to zero, the memory manager will automatically free it. This is useful if several threads will be accessing the same block. For more information, see "The Reference Count" on page 563.

When a geode exits, all blocks owned by it are automatically freed.

15.3.4 Accessing Data in a Block

`MemLock()`, `MemUnlock()`

Because the memory manager is constantly reorganizing the global heap, applications must have a way of making sure a block stays put when they want to use it. Applications must also have a way of recalling swapped blocks when they are needed.

These requirements are met by the memory manager's locking scheme. Whenever you need to access data in a non-fixed block, you must first lock it.

15.3

**Warning**

Do not lock a fixed block; an error will result.

This will cause the memory manager to copy the block back into the global heap if it had been swapped; the memory manager will not move, swap, or discard a block while the block is locked.

Any block may be locked several times. Each lock increments the block's lock count (to a maximum of 255 locks per block), and each unlock decrements it. The memory manager can only move the block when the lock count is zero.

One warning about locking blocks: Do not try to lock a block which was allocated as fixed. Attempting to do so will result in a system error. If you need to translate a fixed-block handle to a pointer, call **MemDeref()**.

MemLock() locks a block on the heap. It is passed the handle of the block; it returns a pointer to the start of the block on the heap. If the block has been discarded, **MemLock()** returns a null pointer.

Immediately after you are done using a block, you should unlock it by calling **MemUnlock()**. It is better to lock and unlock the same block several times than to retain control of it for an extended period, as locked blocks degrade the performance of the heap compaction mechanism. To unlock a block, call **MemUnlock()**, passing the handle of the block to be unlocked.

MemUnlock() decrements the lock count.

A block may be locked by any of the threads run by its creator; if the block is sharable, it may be run by any other thread as well. There is nothing in the **MemLock()** routine to prevent different threads from locking a block at the same time, causing potential synchronization problems. For this reason, if threads will be sharing a block, they should use the synchronization routines (see section 15.3.6 on page 558).

15.3.5 Accessing Data: An Example

Code Display 15-1 shows how to allocate a block, lock it, access a word of data in the block, and unlock the block. This example shows the basic principles of using blocks.

Code Display 15-1 Allocating and Using a Block

```

/*
 * Variable Declarations
 */

MemHandle myBlockHandle;
char      charArray[50], *blockBaseAddress;

/* First, we allocate a block of the desired size. Since we'll use the block right
 * away, we allocate the block already locked.
 */
myBlockHandle = MemAlloc(          /* MemAlloc returns the block handle */
    2048,                          /* Allocate 2K of memory */
    HF_SWAPABLE,                   /* HeapFlags: Make block swapable */
    HAF_ZERO_INIT|HAF_LOCK); /* HeapAllocFlags: Initialize
                                * the memory to zero & lock it */

/* The block is already locked on the global heap. However, we do not have the
 * block's address; we just have its handle. Therefore, we need to call a routine
 * to dereference the handle. */
blockBaseAddress = (char *) MemDeref(myBlockHandle); /* Returns a ptr to base of
                                                       * block */

/* Enter some data in the block */
strcpy(blockBaseAddress,
    "I can resist anything except temptation.\n    --Wilde"

/* We're done with the block for the moment, so we unlock it. */
MemUnlock(myBlockHandle); /* blockBaseAddress is now meaningless */

/* Here we do some other stuff . . . */

/* Now we want to use the block again. First we have to lock it. */
blockBaseAddress = (byte *) MemLock(myBlockHandle); /* Returns a ptr to locked
                                                       * block */

/* Read a string from the block: */
strcpy(charArray, blockBaseAddress);

/* We're done with the block, so we free it. Note that we can free the block
 * without unlocking it first.
 */
MemFree(myBlockHandle); /* myBlockHandle is now meaningless */

```

15.3



15.3.6 Data-Access Synchronization

```
MemThreadGrab(), MemThreadGrabNB(), MemThreadRelease(),  
MemLockShared(), MemUnlockShared(), MemLockExcl(),  
MemDowngradeExclLock(), MemUpgradeSharedLock(),  
MemUnlockExcl(), HandleP(), HandleV(), MemPLock(),  
MemUnlockV()
```

15.3

Blocks can have the property of being sharable—that is, the same block may be locked by threads owned by several different geodes. However, this can cause data synchronization problems; one application can be changing data while another application is trying to use it. To prevent this, GEOS provides semaphore routines. Only one thread can have the block's semaphore at a time. When an application wants to use a shared block, it should call a routine to set the semaphore. Once the routine returns, the application can use the block; when it is done, it should release the block's semaphore so other applications can use the block.



Warning

Use of semaphores is voluntary; a thread can lock a block without having its semaphore.

Note that use of semaphores is entirely voluntary by each application. Even if thread A has the semaphore on a block, thread B can call **MemLock()** on the block and start changing it. However, all threads using shared blocks ought to use the semaphore routines to prevent confusion.

There are several different sets of routines which can be used to control a block's semaphore. The different sets of routines make different trade-offs between faster operation and increased flexibility. Any one block should be accessed with only one set of routines; different threads should all be using the same routines to access a given block, and a thread should not switch from one set of routines to another for a particular block. If this rule isn't followed, results are undefined. All of these routines access the *HM_otherInfo* word of the handle table entry; if the block will be locked by any of these routines, you must not alter that word. None of these routines is used to access object blocks; instead, special object-block locking routines are provided.

Most geodes should use **MemThreadGrab()**, **MemThreadGrabNB()**, and **MemThreadRelease()** to access sharable blocks. These routines provide the maximum protection against deadlock in exchange for a slightly slower execution.

MemThreadGrab() gives the thread the semaphore for the block in question and locks the block. It is passed the handle of the block and returns the block's address on the global heap. If no thread has the block's semaphore, it gives the semaphore to the calling thread. If the calling thread already has the semaphore, a "semaphore count" is incremented; the thread will not release the semaphore until it has been released as many times as it has been grabbed. (For example, two different objects run by the same thread could each grab the semaphore; the semaphore would not be released until each object called **MemThreadRelease()**.) If another thread has the semaphore, **MemThreadGrab()** blocks until it can get the semaphore; it then increments the semaphore, locks the block, and returns the address.

15.3

MemThreadGrabNB() is the same as **MemThreadGrab()**, except that it never blocks. If you call **MemThreadGrabNB()** while another thread has the semaphore, the routine will immediately return an error.

MemThreadGrabNB() takes the handle of the block; it increments the semaphore, locks the block, and returns the block's address on the heap.

MemThreadRelease() releases a block grabbed by either **MemThreadGrab()** or **MemThreadGrabNB()**. It is passed the block's handle. It unlocks the block and decrements the block's semaphore.

One common situation is that several threads may need to read a block but only once in a while will an application need to write to the block. In this case, there is no synchronization problem in having several readers at once; however, if any thread is writing, no other thread should be reading or writing. For this situation, GEOS provides this set of **MemLock** routines: **MemLockShared()**, **MemUnlockShared()**, **MemLockExcl()**, and **MemUnlockExcl()**.

These routines, like the others, maintain a queue of threads which have requested thread access. The difference is that any number of readers can have access at once. When a thread wants read access, it calls **MemLockShared()**. If the queue is empty and the block is unlocked or locked for reading, the routine returns and the thread is given shared access; otherwise, the thread is blocked, and the request goes on the queue. When a routine is finished reading the block, it should call **MemUnlockShared()**.

When a routine needs to write to a block, it should call **MemLockExcl()**. If nobody has locked the block (and thus the queue is empty), the thread will immediately be given exclusive access; otherwise, the thread will block, and



the request will go on the queue. When the thread no longer needs write access, it should call **MemUnlockExcl()**.

When all threads with access to a block have released their locks, the queued thread with the highest priority will be awakened and given the lock on the block. If that thread had requested shared access, all other threads on the queue that had requested shared access will also be awakened and given locks.

15.3

A thread can change its lock from shared to exclusive or vice versa. If a thread has an exclusive lock on a block, it can change the lock to shared by calling **MemDowngradeExclLock()**. This routine takes one argument, namely the block's global handle. It changes the lock to "shared" and wakes up all sleeping threads which are waiting for shared access. For convenience, **MemDowngradeExclLock()** returns the address of the block; however, the block is guaranteed not to move.

If a thread has shared access and wants exclusive access, it can call **MemUpgradeSharedLock()**. If the thread has the only lock on the block, its lock will be changed to "exclusive" (even if there are writers on the queue). If any other threads have the block locked, the routine will release the thread's lock and put the thread on the queue. When the thread comes to the head of the queue, the routine will wake the thread and give it an exclusive lock. The routine returns the block's address on the global heap. Note that the block may be altered or moved during this call if the call blocked.

Once a thread has been given a shared lock, there is nothing to prevent it from altering (or even freeing) the block. The routines rely on good citizenship by the threads using them. Also, if a thread isn't careful, there is a great danger of deadlock. If (for example) a thread requests exclusive access to a block when it already has access, the thread will deadlock: it will block until the threads with access all release the block, but it can't release its own lock because it is blocked. If you may need to have multiple locks on a block, use the **MemThread** routines, which check for these situations.

There are other sets of routines which can be used to access a block's semaphore. As noted, a block should be accessed via just one set of routines. These routines provide less protection against deadlock than the **MemThread** routines do; however, they have a slightly faster response time.

**Deadlock Warning**

These routines will deadlock if a thread is not very careful. Most of the time you should use the **MemThread** routines, which avoid some deadlock situations.

A more primitive group of routines is **HandleP()**, **HandleV()**, **MemPLock()**, and **MemUnlockV()**. These routines function much like the **MemThread** routines. **HandleP()** grabs the block's semaphore and returns; it does not lock the block. This makes it very useful for working with fixed blocks (which cannot be locked). **HandleV()** releases the block's semaphore and returns; it does not unlock the block. Note, however, that **HandleP()** will block if *any* thread controls the semaphore, even the thread that called **HandleP()**. If a thread calls **HandleP()** while it controls the semaphore, it will block until the semaphore is released, but it can't release the semaphore because it has blocked. Thus, the thread will deadlock, and no other thread will be able to get the semaphore. Therefore, a thread should use **HandleP()** only if it is very confident that it will never try to double-set the semaphore.

15.3

Usually, when a thread grabs a block's semaphore, it needs to have the block locked on the heap. For this reason, GEOS provides the routines **MemPLock()** and **MemUnlockV()**. **MemPLock()** simply calls **HandleP()** and then calls **MemLock()**. **MemUnlockV()**, correspondingly, calls **MemUnlock()** and then calls **HandleV()**. These routines are completely compatible with **HandleP()** and **HandleV()**; for example, a thread could grab and lock a block with **MemPLock()**, then unlock it with **MemUnlock()** and release it with **HandleV()**.

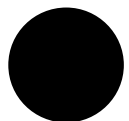
HandleP() and **HandleV()** are general-purpose handle routines. They can be called on any type of global handle. For example, if two threads need to synchronize their access to a file, they can lock and unlock the file handle with **HandleP()** and **HandleV()**. However, they are most commonly used with memory blocks.

15.3.7 Retrieving Block Information

```
MemDeref(), MemGetInfo(), MemModifyFlags(),
HandleModifyOwner(), MemModifyOtherInfo()
```

GEOS offers several routines to retrieve and change information about a block. Each of these routines has a full entry in the Routine Reference Book.

MemDeref() is passed the handle of a block on the global heap; it returns the block's address on the global heap. As noted above, this routine is useful



when you allocate a fixed or locked block. If the block has been discarded, it returns a null pointer.

MemGetInfo() is a general-purpose block information routine. It is passed two arguments: the handle of the block, and a member of the **MemGetInfoType** enumerated type. The return value is always word-length; however, its significance depends on the **MemGetInfoType** value passed:

15.3

MGIT_SIZE Returns the size of the memory block (in bytes).

MGIT_FLAGS_AND_LOCK_COUNT

Upper byte is the number of locks on the block; lower eight bits are the block's **HeapFlags** record (see page 551).

MGIT_OWNER_OR_VM_FILE_HANDLE

If the block is attached to a GEOS Virtual Memory file, **MemGetInfo()** returns the VM file handle. Otherwise, it returns the **GeodeHandle** of the owning thread.

MGIT_ADDRESS

Returns the block's segment address, if it is on the global heap; otherwise, it returns zero. If the block is resized or is not locked, the address may change without warning. Note that the segment address is returned as a word, not as a pointer; this is of limited utility in C.

MGIT_OTHER_INFO

Returns the *HM_otherInfo* word from the block's handle table entry. The usage of this word varies depending on the block's nature; for example, semaphore routines use this word.

MGIT_EXEC_THREAD

This is useful for object blocks only. Returns the handle of the thread executing methods for objects in the block.

MemModifyFlags() is used to change a block's **HeapFlags** record. It takes three arguments: The handle of the block, the **HeapFlags** to turn on, and the **HeapFlags** to clear. It returns nothing. Not all **HeapFlags** can be changed after a block is created; only **HF_SHARABLE**, **HF_DISCARDABLE**, **HF_SWAPABLE**, and **HF_LMEM** can be so changed.

HandleModifyOwner() changes the geode owning a given block. It takes two arguments, namely the handle of the block and the handle of the new

owner. It returns nothing. If the block is not sharable, only the owner of a block can change the block's owner.

MemModifyOtherInfo() changes the *HM_otherInfo* word of the block's handle table entry. It takes two arguments: The handle of the block, and one word of data to store in the *HM_otherInfo* field. It returns nothing.

15.3.8 The Reference Count

15.3

`MemInitRefCount()`, `MemIncRefCount()`, `MemDecRefCount()`

Sometimes several different threads will need to look at the same block of memory. For example, a single thread might need to send the same information to objects in several different threads. The simplest way to do that would be to write the information in a global memory block and pass the block's handle to the objects. However, it's a bad idea to allocate global blocks for longer than necessary, since this uses up handles. It therefore becomes important to free these blocks when everyone's done with them.

GEOS provides a simple mechanism for managing this. Every block can have a *reference count*. When a reference count for a block reaches zero, the block is automatically freed. That way, for example, if an object needed to send the same information to five objects, it could give the block a reference count of five and send its handle to the objects. Each object, when it finished accessing the data, would decrement the reference count. When all five objects had decremented the reference count, the block would be freed.

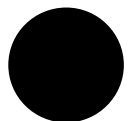


Warning

These routines use the *HM_otherInfo* field. Therefore, you cannot use them with any of the semaphore routines.

You must be careful about a few things while using the reference count mechanism. First of all, the reference count is kept in the *HM_otherInfo* field of the block's handle table entry. For this reason, you must not use the reference count routines if you will be using *HM_otherInfo* for any other purpose. In particular, you may not use any of the data-access synchronization routines described in section 15.3.6, since all of those routines store the semaphore in *HM_otherInfo*. You should generally use the reference count only for blocks that will not be changed until they are freed, so that data synchronization will not be an issue.

Second, once the reference count is decremented to zero, the block is *immediately* freed. Once a block is freed, its handle may be used for something else. If you try to increment or decrement the reference count of



the block, the results are undefined. For this reason, you should make sure that the reference count does not reach zero until all threads are done with the block. One way to arrange for this is to have a single thread do all the decrementing. For example, an object might set the reference count to five, and send the handle to five other objects. Each of these objects, when finished with the block, would send a message back to the first object, which would decrement the reference count. As an alternative, you could have each of the objects decrement the count itself when it was finished. In this case, the first object would have to assume that the block was freed as soon as it sent out all of the messages, since it would have no way of knowing when the other objects would be finished with the block.

Finally, since the reference count is stored in *HM_otherInfo*, it has a maximum value of $(2^{16} - 1)$. If you try to increment the reference count past this value, the results are undefined. This will not be a problem for most applications.

To set up a reference count for a block, call **MemInitRefCount()**. This routine takes two arguments: the handle of a global memory block, and the reference count for that block. The reference count must be greater than zero. **MemInitRefCount()** sets the block's *HM_otherInfo* field to the specified reference count. **MemInitRefCount()** does not return anything.

To increment the reference count, call **MemIncRefCount()**. This routine is passed a single argument, namely the handle of the global memory block. The routine simply increments *HM_otherInfo*. It does not return anything.

To decrement the reference count, call **MemDecRefCount()**. This routine is passed the handle of a global memory block. It decrements the block's *HM_otherInfo* field. If the field reaches zero, **MemDecRefCount()** will immediately free the block. The routine does not return anything.

15.4 malloc()

`malloc()`, `calloc()`, `realloc()`, `free()`

GEOS provides support for the Standard C memory allocation routines. However, support is limited by the nature of the 80x86 and the GEOS memory management system.

A geode can request memory with **malloc()** or **calloc()**. When a geode does this for the first time, the memory manager will allocate a fixed block and return a pointer to within the fixed block. (This block is actually a special kind of LMem heap.) Because the memory is in a fixed block, the geode does not need to access it with handles; it can use the pointer directly. If the block fills up, the manager can allocate another fixed block for these requests.

However, there are some problems with this. The main problem is that fixed blocks degrade the memory manager's performance. The more a geode uses **malloc()**, the more memory is tied up in fixed blocks. And, as always, contiguous memory is limited to 64K by the 80x86 segmented addressing scheme.

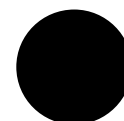
15.4

Most of the time, geodes should use other types of memory allocation. For allocating small chunks of data, applications should use **LMem** routines or techniques built on top of them (database items, chunk arrays, etc.); for larger chunks of memory, applications should use memory manager routines or HugeArrays. However, to help writers port C code to GEOS, **malloc()** and its relatives are available.

To get a stretch of contiguous memory, use the routines **malloc()** or **calloc()**. **malloc()** takes one argument, a size in bytes; it returns a void pointer to that many bytes of fixed memory. **calloc()** takes two arguments: a number of structures, and the size of each such structure. It allocates enough memory for that many structures and returns a void pointer to the memory. Both **malloc()** and **calloc()** zero-initialize the memory when they allocate it.

If a routine wants to change the size of memory allocated with **malloc()** or **calloc()** it can use **realloc()**. **realloc()** takes two arguments: a pointer to a piece of memory allocated with **malloc()** or **calloc()**, and a new size in bytes. It returns a void pointer to the memory, which may have been moved to satisfy the request. If it could not satisfy the request, it returns a null pointer, and the original memory is untouched. Note that the pointer you pass **realloc()** *must* be the same pointer that was returned by **malloc/calloc**; if (for example) you allocate 100 bytes and are returned 008Bh:30h, and try to resize it by passing 008Bh:40h to **realloc()**, inappropriate memory will be affected, and the results are undefined.

If you decrease the size of a memory section with **realloc()**, the routine is guaranteed to succeed. If you increase the size, it may fail; if it does succeed, the new memory will *not* be zero-initialized. Reallocating a block down to



Memory Management

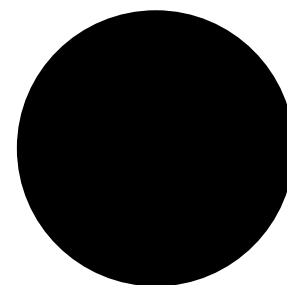
566

zero memory is the same as freeing it. You can pass a null pointer to **realloc()** along with the size; this makes **realloc()** function like **malloc()**.

When you are done with memory allocated by **malloc**-family routines, you should call **free()** to free the memory for other **malloc()** calls. As with **realloc()**, you must pass the same pointer that you were originally given.

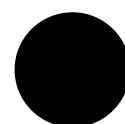
15.4

Local Memory



16

16.1	Design Philosophy	569
16.2	Structure of a Local Memory Heap	570
16.2.1	The Local Heap	571
16.2.2	Chunks and Chunk Handles.....	572
16.2.3	Types of LMem Heaps	573
16.3	Using Local Memory Heaps	577
16.3.1	Creating a Local Heap.....	577
16.3.2	Using Chunks	579
16.3.3	Contracting the LMem Heap	581
16.3.4	Example of LMem Usage	581
16.4	Special LMem Uses	583
16.4.1	Chunk Arrays	584
16.4.1.1	Structure of the Chunk Array	584
16.4.1.2	Creating a Chunk Array.....	586
16.4.1.3	Adding, Removing, and Accessing Elements.....	587
16.4.1.4	Chunk Array Utilities.....	590
16.4.1.5	Example of Chunk Array Usage	592
16.4.2	Element Arrays.....	595
16.4.2.1	Creating an Element Array.....	596
16.4.2.2	Adding an Element	597
16.4.2.3	Accessing Elements in an Element Array	599
16.4.2.4	Removing An Element From An Element Array	600
16.4.2.5	The “Used Index” and Other Index Systems.....	601
16.4.3	Name Arrays.....	602
16.4.3.1	Creating a Name Array	603
16.4.3.2	Accessing Elements in a Name Array	606





The GEOS memory manager is well suited to dealing with blocks of memory in the 2K-6K range. However, for small amounts of memory, the manager's overhead becomes significant. For this reason, GEOS provides local memory (or *LMem*) routines. These routines let an application allocate a block of memory and designate it as a *local memory heap*. The application can then request small amounts of memory from this heap. The LMem library automatically reorganizes the heap when necessary to make space available.

16.1

The local memory routines are also used to manipulate objects. All objects are special LMem chunks which are stored in a special type of LMem heap (called an *object block*). Most of the routines which work on chunks can also be used on objects.

Before you read this chapter, you should be familiar with the use of handles in GEOS and with the 80x86's segment:offset memory referencing. You should also be familiar with the global memory manager (see "Memory Management," Chapter 15).

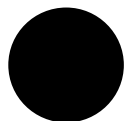


16.1 Design Philosophy

The GEOS memory manager is designed to deal with blocks of memory which are measured in kilobytes. Every memory block needs an entry in the global handle table; each such entry takes up 16 bytes. This memory overhead is insignificant for a 10K code block but very significant for, for example, a 100 byte database entry. Furthermore, GEOS allows a limited number of global handles. If an application were to use global blocks for small amounts of data, it could use up too many handles.

On the other hand, if an application were to store lots of small pieces of data in a single global block, it might have to write elaborate memory-management routines. It would have to make sure that it could resize a piece of data at will and shuffle other data to make room. This could force programmers to spend a lot of their time writing support code.

For these reasons, GEOS provides local memory routines. Applications can designate a block of memory as a local-memory heap and use LMem routines



to request small amounts (*chunks*) of data. The local memory library automatically shuffles chunks in that heap to make room for new requests. Applications can change the size of a chunk at will. The cost of using the LMem routines is one added layer of indirection and a small amount of memory overhead for each chunk.

16.2

The LMem routines have another advantage. They provide a uniform way of managing small pieces of memory. This lets GEOS add functionality which applications can use in a variety of ways. For example, GEOS implements an array-management scheme based on LMem chunks; this scheme comes complete with a modified Quicksort routine. Similarly, all GEOS objects are stored in object blocks, which are a special kind of local-memory heap. This makes it easy to add or delete objects dynamically.

16.2 Structure of a Local Memory Heap

A local memory heap looks and acts much like the global heap. However, it is contained entirely within a single memory block. This block is initialized with a 16-byte **LMemBlockHeader** (described on page 577), a local memory handle table, and a local memory heap. Optionally, a space for data may be allocated between the header and the handle table.

Each allocated section of memory within a local heap is called a *chunk*, and the handles of these chunks are called *chunk handles*. See Figure 16-1 on page ● 571 for an illustration of a local heap.

Chunks comprise a chunk of data preceded by a word that contains the length of the data (in bytes). When the heap is created, a certain number of chunk handles will be allocated. If a chunk is requested after all of these chunks have been given out, the local memory routines will enlarge the LMem handle table, relocating chunks as necessary. Unused chunks are stored in a linked list.

16.2.1 The Local Heap

An application may designate any block as a local memory heap. The block may or may not be fixed, swappable, or discardable, as the application desires.

A block may have other data besides the local heap. When a local heap is created, an offset into the block can be specified. The heap will put its header structure at the beginning of the block and the handle table at the specified offset; everything in between will be left untouched by the LMem routines. The offset must be larger than the standard header or else may be zero, indicating that the default offset should be used.

16.2

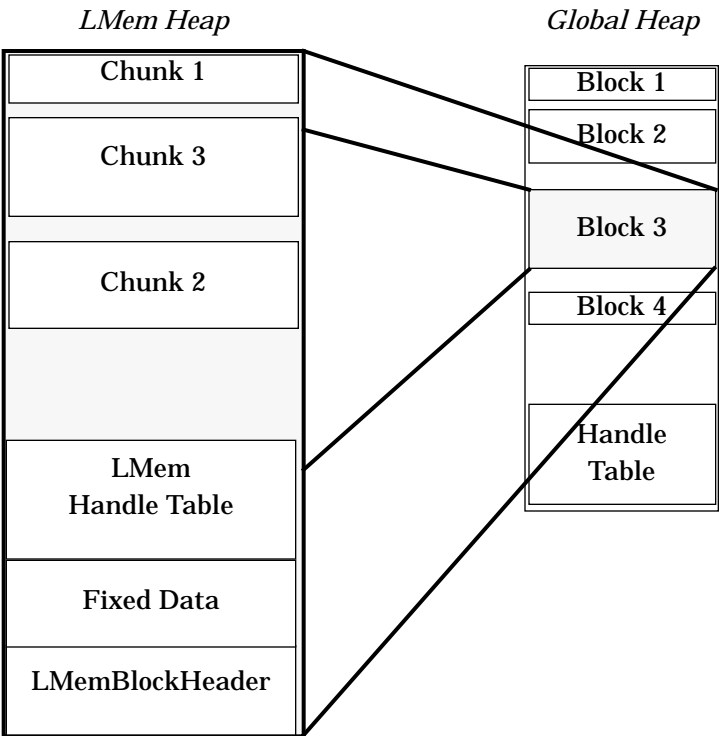


Figure 16-1 *A Local Memory Heap*
A block on the Global Heap that has been allocated as a Local Memory heap. It is partitioned into a block header, a chunk of fixed data, an LMem handle table, and a local memory heap.

Before performing any operations on a local memory heap, an application must lock the heap the way it would any other block. Some local memory routines may need to resize the block; this may cause the block to be moved on the global heap. Therefore, you should assume that these routines may invalidate all pointers to the relevant block. The descriptions of routines which may behave this way will contain a warning to that effect. When you initialize an LMem heap, you can specify that it should never be resized; this is advisable if the heap is in a fixed block.

16.2

A virtual-memory file block may contain an LMem heap. For details on this, see “Virtual Memory,” Chapter 18.

16.2.2 Chunks and Chunk Handles

Just as blocks on the local heap are accessed with handles, chunks are accessed via chunk handles. Each chunk handle is an offset into the block containing the local memory heap; thus, the segment address of the locked heap, combined with the chunk handle, make up a pointer to a location within the local memory heap's chunk handle table. That location contains another offset which, when combined with the segment address of the block, composes a far-pointer to the actual chunk. Figure 16-2 on page ● 573 shows the use of a chunk handle to access a chunk.



Chunk Addresses Are Volatile

Adding or resizing a chunk or sending a message can change all chunk addresses in an LMem heap.

Chunks are movable within the local heap; whenever a chunk is created or resized, the local memory manager may move any chunks in that heap. There is no locking mechanism for chunks; thus, creating or resizing a chunk can potentially invalidate pointers to all the other chunks, forcing the application to again dereference the handles for these chunks. Be warned that many message handlers can cause heap compaction. As a general rule, you should not save chunk addresses around message sends; instead, dereference the chunk handles to get the current address.

Chunks are aligned along dwords. This speeds up chunk moves and similar operations. This means that when you request a chunk, its size may be slightly larger than you request.

Objects are special kinds of LMem chunks. An optr is simply the global memory handle of an LMem heap followed by the chunk handle of an object. For this reason, many LMem routines come in two formats: one which is

passed an optr, and one which is passed the global and chunk handles. There is also a macro, **ConstructOptr()**, which is passed a memory handle and a chunk handle and returns an optr constructed from the two.

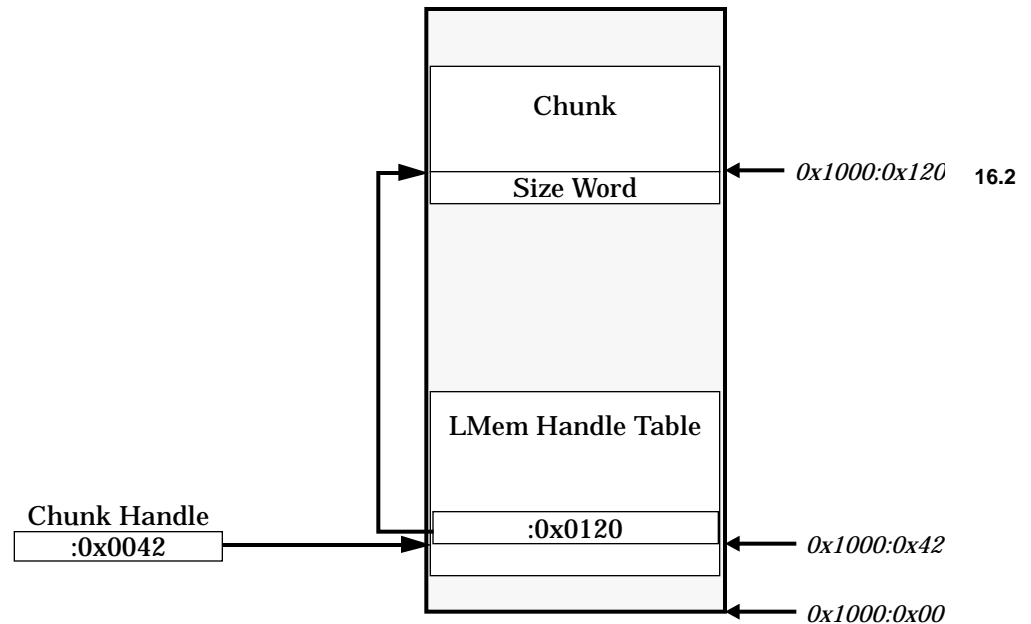


Figure 16-2 Referencing a Chunk

A chunk handle consists of an offset into the block containing the local heap; this offset indicates an entry in the local memory handle table, where an offset to the chunk data is stored. In this case, the block containing the LMem heap begins at address 1000:00. All addresses shown are in hexadecimal.

16.2.3 Types of LMem Heaps

LMemType, LMemFlags

In addition to being used for general memory needs, local memory heaps perform many specific functions in the GEOS system. When an LMem heap is created, a member of the **LMemType** enumerated type is passed, indicating to what use the LMem heap will be put. Several types are available:



LMEM_TYPE_GENERAL

The LMem heap will be used for general data storage, possibly including a chunk, name, or element array. When an application creates an LMem heap, it will almost always be of type “General” or “Object.”

LMEM_TYPE_WINDOW

Windows are stored in memory as LMem heaps. The header contains information about the window; each region in the window is stored as a chunk. Applications will not directly create Window heaps.

LMEM_TYPE_OBJ_BLOCK

Objects are stored in object blocks, which are LMem heaps. An object block has some extra header information and contains one chunk which contains only flags. All the objects in the block are stored as chunks on the heap. Applications can directly create object blocks; for more information, see “GEOS Programming,” Chapter 5.

LMEM_TYPE_GSTATE

A GState is an LMem heap. The GState information is in the header, and the application clip-rectangle is stored in a chunk. Applications do not directly create GState blocks; rather, they call a GState creation routine, which creates the block. (See “Graphics Environment,” Chapter 23.)

LMEM_TYPE_FONT_BLOCK

Font blocks are stored as LMem heaps. Applications do not create font blocks directly.

LMEM_TYPE_GSTRING

Whenever a GString is created or loaded, a GString LMem heap is created, and elements are added as chunks. The heap is created automatically by the GString routines; applications should not create GString blocks. (See section 23.8 of chapter 23.)

LMEM_TYPE_DB_ITEMS

The Virtual Memory mechanism provides routines to create and manage *database items*, short pieces of data which are dynamically allocated and are saved with the VM file. These items are stored in special database LMem heaps, which are created in special database blocks in the VM file. Applications

do not directly allocate DB blocks; rather, they call DB routines, which see to it that the blocks are created. (See “Database Library,” Chapter 19.)

When an LMem heap is allocated, certain flags are passed to indicate properties the heap should have. Some of these flags are passed only for system-created heaps. The flags are stored in a word-length record (**LocalMemoryFlags**); the record also contains flags indicating the current state of the heap. The **LocalMemoryFlags** are listed below:

16.2

LMF_HAS_FLAGS

Set if the block has a chunk containing only flags. This flag is set for object blocks; it is usually cleared for general LMem heaps.

LMF_IN_RESOURCE

Set if the block has just been loaded from a resource and has not been changed since being loaded. This flag is set only for object blocks created by the compiler.

LMF_DETACHABLE

Set if the block is an object block which can be saved to a state file.

LMF_DUPLICATED

Set if block is an object block created by the **ObjDuplicateResource()** routine. This flag should not be set by applications.

LMF_RELOCATED

Set if all the objects in the block have been relocated. The object system sets this when it has relocated all the objects in the block.

LMF_AUTO_FREE

This flag is used by several object routines. It indicates that if the block's in-use count drops to zero, the block may be freed. This flag should not be set by applications.

LMF_IN_MEM_ALLOC

This flag is used in error-checking code to prevent the heap from being validated while a chunk is being allocated. For internal use only—do not modify.

LMF_IS_VM

Set if LMem heap is in a VM block and the block should be marked dirty whenever a chunk is marked dirty. This flag is automatically set by the VM code when an LMem heap is created in or attached to a VM file. This flag should not be set by applications.

LMF_NO_HANDLES

Set if block does not use chunk handles. A block can be set to simulate the C **malloc()** routine; in this case, chunks are not relocated after being created, so chunk handles are not needed. Ordinarily, these blocks are created by the **malloc()** routine, not by applications. (See the discussion of **malloc()** in section 15.4 of chapter 15.)

LMF_NO_ENLARGE

Indicates that the local-memory routines should not enlarge this block to fulfill chunk requests. This guarantees that the block will not be moved by a chunk allocation request; however, it makes these requests more likely to fail.

LMF_RETURN_ERRORS

Set if local memory routines should return errors when allocation requests cannot be fulfilled. If the flag is not set, allocation routines will fatal-error if they cannot comply with requests. This flag is generally clear for expandable LMem blocks, since many system routines (such as **ObjInstantiate()**) are optimized in such a way that they cannot deal with LMem allocation errors.

LMF_DEATH_COUNT

This field occupies the least significant three bits of the flag field. It means nothing if the value is zero. If it is non-zero, it indicates the number of remove-block messages left which must hit **BlockDeathCommon** before it will free the block. This flag is used by the handlers for **MSG_FREE_DUPLICATE** and **MSG_REMOVE_BLOCK**.

STD_LMEM_OBJ_FLAGS

This is a constant which combines the **LMF_HAS_FLAGS** and **LMF_RELOCATED** flags. These flags should be set for all object blocks.

16.3 Using Local Memory Heaps

Local memory heaps are much like the global heap and are accessed in much the same way. Local heaps are simple to create and manage.

Many applications will not need to use local heaps directly; instead, they will use more advanced data-management mechanisms based on local heaps, such as chunk arrays or database blocks. However, even if you use these mechanisms, you should be familiar with this section; this will help you understand how the other mechanisms work.

16.3

Remember that every local memory heap resides in a global memory block. All the rules for using memory blocks apply. (See “Memory Etiquette” on page 552.)

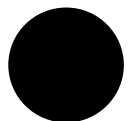
16.3.1 Creating a Local Heap

```
LMemInitHeap(), MemAllocLMem()
```

Before you create a local heap, you must first allocate and lock a block on the global heap using **MemAlloc()** and **MemLock()**. (Alternatively, you can allocate a fixed block.) Then, you must call **LMemInitHeap()**.

LMemInitHeap() creates and initializes a local memory heap. It must be passed several arguments:

- ◆ The handle of a locked or fixed block which will contain the local memory heap. It may be moveable, discardable, both, or neither; it should *not* have the flag **HF_LMEM** set. If the LMem heap will be resizable, the block may be of any size; the LMem routines will enlarge it if necessary. If the heap will not be resizable (i.e. **LMF_NO_ENLARGE** is passed), it must be created large enough to accommodate the heap.
- ◆ The offset within the block where the local heap's handle table will begin. The local heap will fill the space to the end of the block; any data between the **LMemBlockHeader** and the indicated offset will be zero-initialized. If an application will not need a fixed data space, it should specify an offset of zero; the handle table will then be put immediately after the **LMemBlockHeader**. Often, when an application needs a fixed data space, it will define a special structure, the first element of which is an **LMemBlockHeader**, and will pass the size of that structure as the



offset. It can then access the fixed data by using the fields of the structure. If the offset specified is non-zero but is less than the size of an **LMemBlockHeader**, **LMemInitHeap()** will return an error.

- ◆ A member of the **LMemType** enumerated type, specifying the type of block to be created (see page 573).
- ◆ A word of **LocalMemoryFlags** for the heap. (See page 575.)
- ◆ A word specifying the number of chunk handles to leave room for in the starter handle table. When these chunks have all been allocated, the local memory manager will expand the LMem handle table to create more chunk handles. No matter how big the starter handle table is, the heap will initially contain just one chunk (whose size is specified by the next argument). Applications should generally pass the constant **STD_LMEM_INIT_HANDLES**.
- ◆ A word specifying the amount of space to be assigned to the chunk created when the heap is initialized. When more space is needed, the chunks will expand to the end of the block, and (if necessary) the block itself will be expanded. Applications should generally pass the constant **STD_LMEM_INIT_HEAP**.

LMemInitHeap() creates the **LMemBlockHeader** and the chunk handle table. It also creates a single free chunk; more chunks will automatically be created as needed. It may resize the block passed (unless the flag **LMF_NO_ENLARGE** is passed); therefore, any pointers to the block may become invalid. It does not return anything.

If you want to create a memory block and initialize it as an LMem heap in one operation, call **MemAllocLMem()**. This routine takes two arguments: a member of the **LMemType** enumerated type, and the amount of space to leave for the header (again, a zero size indicates that the default header size should be used). **MemAllocLMem()** allocates a movable, swappable block in the global heap, then initializes an LMem heap in that block. If you specify an **LMemType** of **LMEM_TYPE_OBJ_BLOCK**, **MemAllocLMem()** will pass the **STD_LMEM_OBJECT_FLAGS** flags; otherwise, it will pass a clear **LocalMemoryFlags** record.

16.3.2 Using Chunks

```
LMemAlloc(), LMemDeref(), LMemFree(), LMemGetChunkSize(),
LMemReAlloc(), LMemInsertAt(), LMemDeleteAt(),
LMemDerefHandles(), LMemFreeHandles(),
LMemGetChunkSizeHandles(), LMemReAllocHandles(),
LMemInsertAtHandles(), LMemDeleteAtHandles()
```

Once a local heap has been initialized, you can allocate, use, and free chunks at will. Chunks can only be manipulated while the block containing the LMem heap is fixed or locked on the global heap. 16.3

LMemAlloc() allocates a new chunk on the local heap. It is passed the handle of the block containing the heap and the size of the chunk needed.

LMemAlloc() returns the handle of the new chunk (which must then be dereferenced before the chunk is used). The size requested will be rounded up as necessary to ensure that the chunks are dword-aligned. An additional two bytes will be allocated to store the size of the chunk; these bytes will be before the data. This routine may compact the chunks on the local heap, so all pointers to that heap will be invalidated; they will have to be dereferenced by their chunk handles. Furthermore, the block itself may be moved (if LMF_NO_ENLARGE is not set). Even fixed blocks may be moved if they need to expand to accommodate new chunks.

All of the following routines come in two forms. As noted, an optr is simply the handle of an object block, followed by the object's chunk handle. For this reason, most LMem routines come in two slightly different formats: one where the chunk is specified with an optr, and one where it is specified with the two handles. In all other ways, the two versions of each routine are identical. Indeed, in assembly there is only a single version of each routine; the only difference is in how the C routines take their parameters.

Once you have allocated a chunk, you must dereference its chunk handle in order to use it. You can do this with **LMemDeref()**. This routine takes a single parameter, namely the optr. It returns a pointer to the data portion of the chunk (after the size word). This pointer will remain valid until the block is unlocked or until a routine is called which can cause block resizing or heap compaction (e.g. **LMemAlloc()**). Since these routines can invalidate chunk-pointers, it is important that data-synchronization routines be used if more than one thread is accessing the heap; otherwise, one thread may cause



the heap to be shuffled while another thread is trying to read from it. The version which takes handles is named **LMemDerefHandles()**.

When you are done using a chunk of memory, you should free it with **LMemFree()**. This routine is passed an `optr`; it does not return anything. It does not resize the block or shuffle chunks; therefore, pointers to other chunks will not be invalidated by **LMemFree()**. The version which takes handles is named **LMemFreeHandles()**.

16.3

You can find out the size of any chunk by calling the routine **LMemGetChunkSize()**. This routine is passed an `optr`; it returns the size of the chunk in bytes (not counting the chunk's size word). The version which takes handles is named **LMemGetChunkSizeHandles()**.

Chunks can be resized after creation. The Boolean routine **LMemReAlloc()** takes two arguments, namely an `optr` and the new size of the chunk. If the new size is larger than the old one, bytes will be added to the end of the chunk; chunks may be shuffled and the block may be resized, so all pointers to chunks will be invalidated. The new bytes will not be zero-initialized. If the new chunk size is smaller than the old one, the chunk will be truncated; pointers to chunks will not be invalidated. This routine will fail only if the LMem heap ran out of space and could not be resized. In this case, it will return non-zero without changing the chunk. If it succeeds, it returns zero. The version which takes handles is called **LMemReAllocHandles()**.

You can add bytes inside a chunk with the Boolean routine **LMemInsertAt()**. This routine takes three arguments: the `optr`, an offset within the chunk, and the number of bytes to add. The new space is added beginning at the specified offset; it is initialized to zeros. This may cause chunks to be shuffled and/or the block to be expanded; pointers to chunks are therefore invalidated. Note that it is your responsibility to make sure that the offset within the chunk really is in the chunk; otherwise, results are undefined. If **LMemInsertAt()** fails (because the LMem heap ran out of space and could not be expanded), it returns non-zero without changing the chunk; otherwise it returns zero. The version which takes handles is named **LMemInsertAtHandles()**.

You can delete bytes within a chunk with the routine **LMemDeleteAt()**. This routine takes three arguments: the `optr`, the offset within the chunk of the first byte to be deleted, and the number of bytes to delete. This routine does not invalidate pointers to chunks. The routine does not return anything.

Note that it is your responsibility to make sure that all the bytes to be deleted are within the chunk, i.e. that the offset and number of bytes passed do not specify bytes that are beyond the end of the chunk. If you fail to do this, results are undefined. The version which takes handles is named **LMemDeleteAtHandles()**.

16.3.3 Contracting the LMem Heap

16.3

```
LMemContract()
```

The local memory manager routines ordinarily take care of heap compaction. However, you can also order compaction at will.

The routine **LMemContract()** compacts the heap and then frees all the unused heap space (by truncating the block with the LMem heap). The routine takes one argument, namely the handle of the (locked or fixed) block containing the LMem heap. It shuffles all the chunks, thus invalidating pointers to chunks; however, it is guaranteed not to move the block on the global heap.

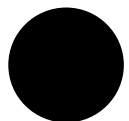
16.3.4 Example of LMem Usage

At first, the local memory techniques can seem tricky. This section contains an example of LMem usage in Goc. The example shows the basic principles of LMem usage.

Code Display 16-1 LMem Usage in GOC

```
/*
 * Declarations
 */

/* We'll want to have a fixed data area at the start of the block. That area will
 * have to start with an LMemBlockHeader, but after that, we can put whatever we
 * want. To make it easy to access the fixed data, we define a structure.
 */
typedef struct {
    LMemBlockHeader MLMBH_standardHeader;
```



Local Memory

582

```
        float      MLMBH_someData;
        float      MLMBH_someMoreData;
        char       MLMBH_someChars[10];
    } MyLMemBlockHeader;

MyLMemBlockHeader    *thisHeapsHeader;
MemHandle            thisHeapsHandle;
ChunkHandle          firstChH, secondChH;
char                 *firstChPtr, *secondChPtr;
int                  i;

16.3
/*
 * Code
 */

/* We have to create the LMem heap. First, we create the block: */
thisHeapsHandle = MemAlloc(          /* MemAlloc returns the block handle */
    2048,                          /* Allocate 2K; can grow as necessary */
    HF_SWAPABLE,                   /* Make block swapable. LMemInitHeap()
                                   * will add the flag HF_LMEM. */
    HAF_ZERO_INIT | HAF_LOCK);      /* Zero & lock the block
                                   * upon allocation */

LMemInitHeap(thisHeapsHandle,        /* Pass handle of locked block */
    LMEM_TYPE_GENERAL,              /* Allocate a general heap */
    0,                              /* Don't pass any flags */
    sizeof(MyLMemBlockHeader),      /* Offset to leave room for header */
    STD_INIT_HANDLES,               /* Standard # of starter handles */
    STD_INIT_HEAP); /* Allocate standard amt. of empty heap */

/* The block is still locked; we can initialize the fixed data section. */
thisHeapsHeader = (MyLMemBlockHeader *) MemDeref(thisHeapsHandle);
thisHeapsHeader->MLMBH_someData = 3.1415926;

/* Now, we allocate some chunks. This invalidates pointers to this heap (such as
 * thisHeapsHeader), since chunk allocation may cause the heap to be resized (and
 * thus moved). The block must be locked when we do this.
 */
firstChH = LMemAlloc(               /* LMemAlloc returns a chunk handle */
    thisHeapsHandle,                /* Pass handle of block . . . */
    100);                          /* . . . and number of bytes in chunk */

secondChH = LMemAlloc(thisHeapsHandle, 50);

/* Now, we start writing data to a chunk: */
firstChPtr = (char *) LMemDerefHandles(thisHeapsHandle, firstChH);
for(i = 0; i <= 30; i++)
    firstChPtr[i] = 'x';
```



```

/* We can insert 10 bytes into the middle of the second chunk. This may cause the
 * chunks or blocks to be shuffled; all pointers are invalidated
 */
LMemInsertAtHandles(thisHeapsHandle, secondChH, /* Block & chunk handles */
                    20, /* Insert after 20th byte */
                    30); /* Insert 30 null bytes */

/* If we want to access the first chunk, we need to dereference its handle again:
 */
firstChPtr = (char *) LMemDeref(thisHeapsHandle, firstChH);
for(i = 1; i <= 15; i++)
    firstChPtr[(i<<1)] = 'o';

/* When we're done with a chunk, we should free it. This does not invalidate
 * any pointers to other chunks.
 */
LMemFreeHandles(thisHeapsHandle, firstChH);

/* If we won't be using an LMem heap for a while, we should unlock it the way we
 * would any block: */
MemUnlock(thisHeapsHandle);

/* When we're done with the LMem heap, we can just free it. */
MemFree(thisHeapsHandle);

```

16.4

16.4 Special LMem Uses

Local memory heaps are used for many purposes in GEOS. Objects are stored in special LMem heaps, called *object blocks*; windows, GStrings, and many other things are implemented as LMem heaps; and LMem heaps can be used to manage arrays of data. Most of these things are done by the system, transparent to the user; however, some of these things can be called on directly by applications.

Most of these techniques are described in their own chapters. One use of LMem heaps will be described in this chapter; namely, the use of LMem chunks to store special arrays of data via the Chunk Array routines. The chapter will also describe special purpose variants of the Chunk Array: the Element Array and the Name Array.



16.4.1 Chunk Arrays

16.4

Very often an application will need to keep track of many different pieces of data and access them by an index. An application can do this in the traditional way, i.e., allocate a block of memory and set it up as an array. However, GEOS provides a mechanism which is often more suitable: the *chunk array*. The chunk array routines let you dynamically insert or delete elements in the middle of an array; you can get a pointer to an arbitrary element (specified by its index number); you can sort the array based on any arbitrary criterion. The array can be specified as “uniform-size” (all elements the same size, specified when the chunk array is created), or “variable-size” (each element can be created at an arbitrary size and can be resized at will). Note that either type of array can grow or shrink dynamically; while the elements may be of a fixed size, the array need not be.

The chunk array is implemented on top of the LMem routines. The entire array is a single chunk in a local memory heap (hence the name). It therefore has a maximum total size of somewhat less than 64K, and memory efficiency drops significantly if it is larger than roughly 6K. If you need a larger array, you should use a Huge Array (see section 18.5 of chapter 18). If you will be using the chunk array routines, you should include **chunkarr.h**.

16.4.1.1 Structure of the Chunk Array

A chunk array is contained in a single chunk in an LMem heap. It begins with a special header structure which specifies certain characteristics of the chunk array: the number of elements in the array, the size of each element (or zero for variable sized arrays), the offset from the start of the chunk to the first element, and the offset from the first element to the end of the array. The header is a structure of type **ChunkArrayHeader**; an application can examine the fields of this structure directly or by using chunk array routines. The creating application can request that the chunk array contain some blank space between the header and the first element; it can use that space however it likes.

Elements can be referenced by index number. The first element has index number zero. You can translate element numbers to pointers, and vice versa, by calling **ChunkArrayElementToPtr()** and **ChunkArrayPtrToElement()** (see section 16.4.1.3 on page 587).

A uniform-size chunk array has a simple structure. After the header (and the extra space, if any) come the elements. They follow one after another, with no free space between them, as shown in Figure 16-3. An application can request a pointer to any element, specified by index; the chunk array routine multiplies the index by the size of each element and adds the product to the address of the first element, producing a pointer to the requested element. An application can also proceed through the array, going from one element to the next, without specifically requesting pointers.

16.4

The structure of a variable-size chunk array is a little bit more complicated. After the header (and extra space) is a lookup-table of two-byte entries, each containing the offset from the start of the chunk to the appropriate element. The chunk array routines maintain this table automatically. When an application wants to reference a given element, the chunk array routine doubles the index number and adds it to the offset to the start of this table; this produces the address of a table entry, which itself contains the offset to the actual element (see Figure 16-4). Effectively, there are two arrays with the same number of elements; the first is a uniform-sized array of offsets to the entries in the second (variable-sized) array. However, this is transparent to the application, which merely requests a pointer to element *n* and is

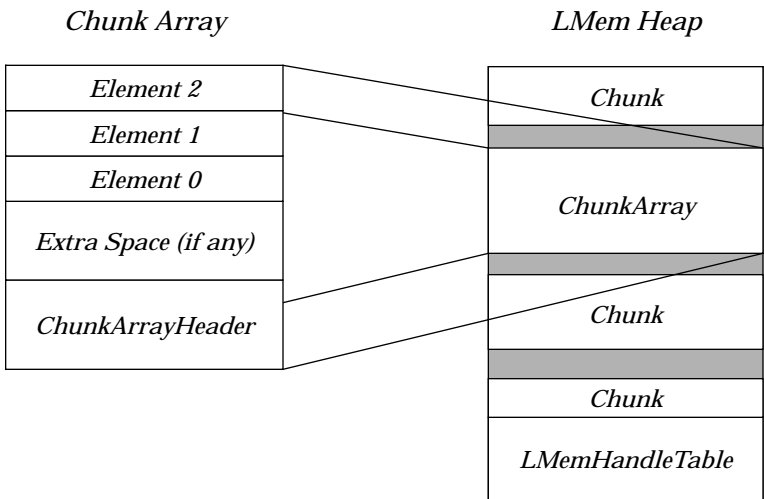


Figure 16-3 *Uniform-Size Chunk Array*
All elements are the same size; thus, to access an individual element, the routines just multiply the element size by the element number and add the product to the address of the first element.

returned that pointer. It does not need to know or care about the extra table constructed by the chunk array routines.

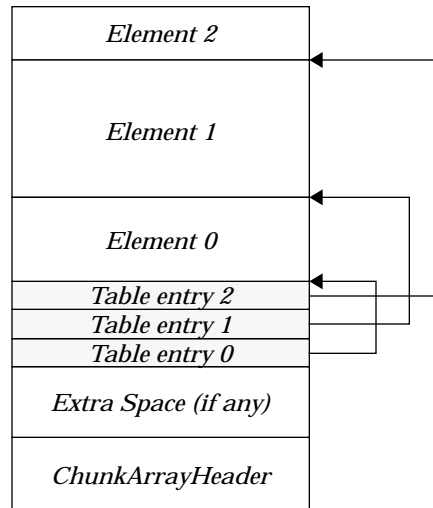


Figure 16-4 Variable-Size Chunk Array

A chunk array with variable-size elements is actually two successive arrays; the first array is an array of chunk offsets to the elements of the second array.

16.4.1.2 Creating a Chunk Array

`ChunkArrayCreate()`, `ChunkArrayCreateAt()`,
`ChunkArrayCreateAtHandles()`, `ChunkArrayHeader`

Chunk arrays are created in local memory heaps. An LMem heap can have several chunk arrays, and it can have other types of chunks besides chunk arrays; however, since an LMem heap is limited to 64K in total size, the more you put in the heap, the less room there is for a chunk array to grow.

The first step in creating a chunk array is to create an LMem heap. Create the heap the same way you would any general LMem heap. The heap should probably be left resizable, since that way it will be able to grow to accommodate the chunk array.

Once you have created the heap, use the routine **ChunkArrayCreate0** to create the chunk array. This routine will allocate a chunk for the chunk

array, initialize the array, and return the chunk handle. Since the routine allocates a chunk, it can cause chunk shuffling or heap resizing; thus, all pointers to the heap are invalidated.

ChunkArrayCreate() takes three arguments:

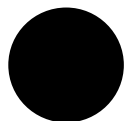
- ◆ The handle of a locked block containing the LMem heap;
- ◆ The size of each element, or zero if the elements are variable-size; and
- ◆ The size of the header for the chunk array. If you pass zero, the routine will automatically leave enough space for a **ChunkArrayHeader** structure; if you pass a non-zero argument, it must be larger than the size of a **ChunkArrayHeader**. If you will need a data space at the start of the array, it is a good idea to define a structure, the first element of which is a **ChunkArrayHeader**, and use that structure to access the fixed data area; you can then pass the size of that structure as this argument. 16.4

There is another version of this routine which creates a chunk array in an existing chunk. This routine, **ChunkArrayCreateAt()**, takes three arguments, namely an optr indicating the chunk, the size of each element, and the size of the header. It allocates a chunk array in that chunk, resizing it if necessary, and returns the chunk's handle. Any data in the chunk may be overwritten (except for whatever data falls in the header area after the **ChunkArrayHeader**). The version which takes handles is called **ChunkArrayCreateAtHandles()**.

When you are done with a chunk array, you can free it with **LMemFree()** the way you would any other chunk.

16.4.1.3 Adding, Removing, and Accessing Elements

```
ChunkArrayAppend(), ChunkArrayAppendHandles(),
ChunkArrayInsertAt(), ChunkArrayInsertAtHandle(),
ChunkArrayDelete(), ChunkArrayDeleteHandle(),
ChunkArrayDeleteRange(), ChunkArrayDeleteRangeHandles(),
ChunkArrayElementResize(),
ChunkArrayElementResizeHandles(),
ChunkArrayElementToPtr(),
ChunkArrayElementToPtrHandles(),
```



```
ChunkArrayPtrToElement(), ChunkArrayPtrToElementHandle(),  
ChunkArrayGetElement(), ChunkArrayGetElementHandles()
```

The chunk array library provides a high-level interface for working with the arrays. The application specifies a request in general terms (e.g., “Insert a new element before element five,” “Give me a pointer to element 20”). The routines take care of the low-level memory management: inserting bytes, swapping chunks to make room, etc. These routines depend on the LMem routines; therefore, the block containing the LMem heap must be locked or fixed when you use these routines (or any other chunk array routines).

When you call a chunk array routine, you must pass the handles specifying the chunk. As with other routines which act on a chunk, these routines come in two formats: one in which the chunk is specified with an optr, and one in which it is specified with the global memory handle and the chunk handle.

Adding elements to a chunk array is easy. To add an element to the end of a chunk array, use the routine **ChunkArrayAppend()**. This routine automatically updates the **ChunkArrayHeader** (and the lookup-table, if elements are variable-sized). The routine takes two arguments, namely the optr and the size of the new element. If the array elements are uniform-sized, the size argument is ignored. The routine will resize the chunk to accommodate the new element, update its header table (and lookup table if necessary), and return a pointer to the element. Since the chunk is resized, all other chunk pointers (and pointers within the chunk array) are invalidated. The version which takes handles is named **ChunkArrayAppendHandles()**.

You can also add an element within the middle of an array. The routine **ChunkArrayInsertAt()** takes three arguments, namely the optr, a pointer to the location at which to insert the element, and the size of the element (ignored for uniform-sized arrays). The routine will insert the appropriate number of bytes at that location in the chunk, update the header and lookup-table, and return a pointer to the new element. Pointers to chunks are invalidated. The version which takes the chunk handle, **ChunkArrayInsertAtHandle()**, is slightly unusual in that it is passed the chunk handle but not the global memory handle; the routine gets the segment address of the chunk from the passed pointers.

When you are done with an element, free it with **ChunkArrayDelete()**. This routine takes two arguments, namely the optr and a pointer to the

element to be deleted. It shrinks the chunk; thus, it is guaranteed not to shuffle chunks, so chunk pointers remain valid (though pointers to elements within the chunk array will be invalidated if the elements come after the deleted element). Again, the handle version, **ChunkArrayDeleteHandle()**, is passed the chunk handle but not the global handle.

If you need to delete several consecutive elements, call **ChunkArrayDeleteRange()**. This routine takes three arguments: the *optr* to the chunk array, the index of the first element to delete, and the number of elements to delete. The specified elements will be deleted. As with **ChunkArrayDelete()**, the global and local heaps will not be shuffled. The handle version, **ChunkArrayDeleteRangeHandles()**, is passed the global handle of the LMem heap and the chunk handle of the chunk array instead of the *optr* to the chunk array.

16.4

Elements in variable-sized arrays can be resized after creation with the routine **ChunkArrayElementResize()**. This routine takes three arguments: the *optr*, the element number, and the new size. The routine resizes the element and updates the lookup table. If the new size is larger than the old, null bytes will be added to the end of the element; chunks may be shuffled, so all chunk pointers are invalidated. If the new size is smaller than the old, the element will be truncated. This is guaranteed not to shuffle chunks, so pointers to chunks remain valid, though pointers within the array may be invalidated. The version which takes handles, **ChunkArrayElementResizeHandles()**, is passed both the global memory handle and the chunk handle.

If you have the index of an element and you want to access that element, use the routine **ChunkArrayElementToPtr()**. It takes three arguments: the *optr*, the element number, and a pointer to a word-length variable. The routine writes the size of the element in the variable and returns a pointer to the element. (If you are not interested in the element's size, pass a null pointer.) It does not change the chunk in any way, so no pointers are invalidated. If you pass an index which is out-of-bounds, **ChunkArrayElementToPtr()** will treat it as the index of the last element. (The constant `CA_LAST_ELEMENT` is often used for this purpose.) However, the error-checking version will always fatal-error if passed the index `CA_NULL_ELEMENT` (i.e. `0xffff`). The version which takes handles is named **ChunkArrayElementToPtrHandles()**.



If you know the address of an element and you need to find out its index, use the routine **ChunkArrayPtrToElement()**. This routine takes two arguments, namely the `optr` and a pointer to the element. It returns the index number of the element. The version which takes the chunk handle, **ChunkArrayPtrToElementHandle()**, is passed the chunk handle and the pointer but not the global memory handle.

16.4

You can copy an element to a specified location with the routine **ChunkArrayGetElement()**. This routine takes three arguments: the `optr`, the element number, and a pointer to a buffer big enough to hold the entire element. The routine will copy the element to the specified buffer. The version which takes handles is called **ChunkArrayGetElementHandles()**.

16.4.1.4 Chunk Array Utilities

```
ChunkArrayGetCount(), ChunkArrayGetCountHandles(),  
ChunkArrayZero(), ChunkArrayZeroHandles(),  
ChunkArrayEnum(), ChunkArrayEnumHandles(),  
ChunkArrayEnumRange(), ChunkArrayEnumRangeHandles(),  
ChunkArraySort(), ArrayQuickSort()
```

To find out how many elements are in a chunk array, use the routine **ChunkArrayGetCount()**. This routine takes one argument, namely the `optr`. It returns the number of elements in the array. It does not change the array; no pointers are invalidated. The version which takes handles is named **ChunkArrayGetCountHandles()**.

If you want to delete all elements in the array but you don't want to free the array itself, use the routine **ChunkArrayZero()**. This routine takes one argument, namely the `optr`. It does not return anything. This routine deletes all the elements in the array, updates the header and lookup tables, and resizes the chunk. Since the chunk is truncated, no chunks are swapped, so no chunk pointers are invalidated (though pointers to elements are, naturally, invalidated). The version which takes handles is named **ChunkArrayZeroHandles()**.

If you want to apply a function to every element in the array, use **ChunkArrayEnum()**. This routine is passed three arguments:

- ◆ The `optr`.

- ◆ A pointer to a Boolean callback routine. This routine will be called for each element in succession. This routine must be declared `_pascal`. If the callback routine ever returns *true*, **ChunkArrayEnum()** will immediately return with value *true* (without checking any more elements). If the callback routine returns *false* for every element, **ChunkArrayEnum()** will return with value *false*.
- ◆ A pointer which is passed to the callback routine.

The callback routine should be written to take two arguments:

16.4

- ◆ A pointer to the start of an element
- ◆ The pointer passed to **ChunkArrayEnum()**

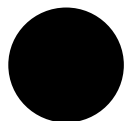
ChunkArrayEnum() can be used for many different purposes, depending on the nature of the callback routine. For example, it can perform some action on every element (in which case it ought always to return *false*); it can analyze the data in the various elements; it can check to see if any element meets some criterion. If it needs to write its results, it might do so at the location indicated by the pointer. **ChunkArrayEnum()** will not cause heap shuffling unless the callback routine causes it; thus, if the callback routine avoids shuffling the heap, it can (for example) be passed a pointer to a chunk in the same LMem heap as the chunk array. The version which is passed handles is named **ChunkArrayEnumHandles()**.

There is another version of **ChunkArrayEnum()** which acts on a range of elements. This routine is called **ChunkArrayEnumRange()**. This routine takes the same arguments as **ChunkArrayEnum()**, plus two more: a start index, and a number of elements to enumerate.

ChunkArrayEnumRange() calls the callback routine for the element with the specified index, and for every element thereafter until it has processed the specified number of elements. You can have it enumerate to the end of the chunk array by passing a count of 0xffff.

The chunk array library also provides a sorting routine,

ChunkArraySort(). This routine performs a modified Quicksort on the array, using insertion sorts for subarrays below a certain size. This gives the sort a performance of $O(n \log n)$. The sort routine takes three arguments: the chunk array's `optr`, a pointer to a comparison function, and a word of data which is passed to the comparison function.



16.4

Whenever the sort routine needs to decide which of two elements comes first, it calls the comparison routine. The comparison routine takes two arguments, namely the `optr` and the word of data passed to **ChunkArraySort()**. It returns a signed word with the following significance: If the first of the elements should come first in the sorted array, it returns a negative number; if the first element ought to come after the second, it should return a positive number; and if it doesn't matter which comes first, it should return zero. You can write a general-purpose comparison routine which can compare based on any of several parts of the element, and you can use the data word to instruct it which part to sort on; or you can use the data word to tell it to sort in ascending or descending order. **ChunkArraySort()** does not cause heap shuffling as long as the comparison routine does not. The routine which takes handles is called **ChunkArraySortHandles()**.

ChunkArraySort() is based on a more general array sorting routine, **ArrayQuickSort()**. **ChunkArraySort()** reads data about the array from the array header and passes the information to **ArrayQuickSort()**. You can call **ArrayQuickSort()** directly for arrays which are not chunk arrays, provided all elements are of uniform size. **ArrayQuickSort()** takes five arguments: a pointer to an array (which should be locked or fixed in memory), the number of elements in the array, the size of each element, a pointer to a callback routine (which has exactly the same format as the **ChunkArraySort()** callback routine), and a data word to pass to that callback routine. It does not return anything.

Note that **ChunkArraySort()** is currently implemented only for chunk arrays with fixed-sized elements.

16.4.1.5 Example of Chunk Array Usage

This section contains an example of how a chunk array might be used. It shows several common chunk array actions, including sorting the array.

Code Display 16-2 Example of Chunk Array Usage

```
/*
 *      Declarations (not in any routine)
 */
```

```

/* We want to store some data right after the chunk array header, so we define our
 * own header structure. Data in this structure (except for the ChunkArrayHeader
 * proper) will not be affected by the chunk array routines. */
typedef struct {
    ChunkArrayHeader    standardChunkArrayHeader;
    int                 someData;
    float               someMoreData;
} MyChunkArrayHeader;

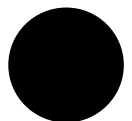
/* For simplicity, we define a structure which will be used for each element in the
 * array. (This is entirely optional.) */
typedef struct {
    char    someText[80];
    int     anInteger;
    float   aFloat;
} MyElementStructure;

/* We define some values to pass to the sort routine. The routine will sort by a
 * different field depending on what value it's passed. */
#define SORT_ARRAY_BY_STRING          0
#define SORT_ARRAY_ASCENDING_BY_INT  1
#define SORT_ARRAY_DESCENDING_BY_FLOAT 2

/* This is the routine we will use to sort the array. We pass the address of this
 * routine to ChunkArraySort(), which will call this routine to compare elements.
 * It returns a negative number if the first element should come before the second
 * in the sorted array, and a positive integer if the second should come before the
 * first. If the elements can be in either order, it returns zero. */
sword _pascal MyElementCompareRoutine(
    MyElementStructure *e1,    /* Address of first element */
    MyElementStructure *e2,    /* Address of second element */
    word               valueForCallback) /* Datum passed in to ChunkArraySort() */
{
    /* We sort differently depending on what the value of valueForCallback is.
     * That way, we can use this one routine for all our sorting.
     */
    switch(valueForCallback) {
        case SORT_ARRAY_ASCENDING_BY_INT:
            /* Compare the elements based on their integer fields. Smaller int
             * comes first.*/
            if (e1->anInteger < e2->anInteger)
                return(-1);
            else if (e1->anInteger > e2->anInteger)
                return(1);
            else return(0);
            break;
    }
}

```

16.4



16.4

```
case SORT_ARRAY_DESCENDING_BY_FLOAT:
/* Compare the elements based on their float fields. Larger float
 * comes first.*/
    if (e1->aFloat > e2->aFloat)
        return(-1);
    else if (e2->aFloat < e2->aFloat)
        return(1);
    else return(0);
    break;

case SORT_ARRAY_BY_STRING:
/* In this case, we call the localization routine to compare the
 * two strings. The localization routine has the same return
 * conventions as this routine, so we return its result directly.
 */
    return(LocalCmpStrings(e1->someText, e2->someText, 40));
    break;

default:
/* If we get here, we were passed a bad callback word. The callback
 * routine therefore does not express a preference in ordering the
 * two elements; it shows this by returning zero.
 */
} /* end of switch */

}

/* All of the above appears in some declaration section. The code below might
 * appear in any routine which creates a chunk array. First, the declarations:
 */
MemHandle          blockWithHeap;
chunkHandle        myChunkArray;
MyChunkArrayHeader *chunkArrayAddress;
MyElementStructure *currentElement;

/* Now the code. Here, blockWithHeap has already been set to hold the block handle
 * of an LMem heap.
 */

MemLock(blockWithHeap); /* Always lock LMem heap before acting on it */
myChunkArray = ChunkArrayCreate(blockWithHeap,
                                sizeof(MyElementStructure), /* Size of each element */
                                sizeof(MyChunkArrayHeader)); /* Size of header */

/* Let's write some data into our part of the header. We need the array's address:
 */
chunkArrayAddress = LMemDerefHandles(blockWithHeap, myChunkArray);
chunkArrayAddress->someData = 42; /* This data won't be affected */
chunkArrayAddress->someMoreData = 2.7182818; /* by chunk array actions */
```

```
/* Now, let's create an element: */
currentElement = ChunkArrayAppendHandles(blockWithHeap, myChunkArray, 0);
    /* That invalidates chunkArrayAddress */
currentElement->anInteger = 1999;
currentElement->aFloat = 1.4142135;
strcpy(currentElement->someText, "Work is the curse of the drinking class.\n" \
    "  --Oscar Wilde")

/* We're done with the array for the moment, so we unlock it: */
MemUnlock(blockWithHeap);

/* Let's assume that several other elements are created now. */

/* . . . */

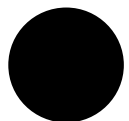
/* Now we need to sort the array: */
MemLock(blockWithHeap);
ChunkArraySortHandles(blockWithHeap, myChunkArray,
    SORT_ARRAY_ASCENDING_BY_INT, /* this is passed to comp. routine */
    MyElementCompareRoutine);

/* Array is now sorted! */
```

16.4

16.4.2 Element Arrays

Sometimes an application will create an array with a high duplication rate; that is, the array may contain many identical elements. This can be inefficient if the duplication rate is very high or elements are very large. For this reason, GEOS provides a special variant of the chunk array known as the *element array*. Every element in an element array has a reference count. When you insert an element, the insertion routine checks whether an identical element already exists in the array. If it does, the routine does not add another copy; instead, it increments the reference count of the element already in the array and returns its element index. If no such element exists, the routine copies the new element into the array, gives it a reference count of 1, and returns its element number. The application may specify a comparison routine which determines whether an element already exists in the array; or it may instruct the insertion routine to do a byte-level comparison.



Note that elements in an element array may be of fixed, uniform size, or they may be of variable size (just as with chunk arrays). When you create an element array, you must specify the size of each element; specifying a size of zero indicates that the elements are of variable size.

Members of an element array keep their index numbers until they are freed. If an element is deleted, the element array routines actually just resize the element to zero and add it to a free list. This means that an element with index 12 might not be the thirteenth element in the array, as it would in a chunk array (remember, indexes start with zero); there might be freed elements before it. For this reason, we speak of an element in an element array having a “token” instead of an index; you should generally consider a token to be an opaque value. Nevertheless, in most situations, element array tokens behave just like chunk array indexes.

When you delete a reference to an element, its reference count is decremented. If the reference count reaches zero, the routine calls an application-specified callback routine to delete the element itself.

Note that adding an element to an element array requires a linear search through the existing elements; thus, element arrays are inefficient for large numbers of elements, if elements will be continually added. Accessing elements, however, takes constant time, since the element array routines can quickly translate an element’s token into the offset to that element. Thus, it takes no longer to access an element in an element array than it does to access one in a chunk array.

16.4.2.1 Creating an Element Array

```
ElementArrayCreate(), ElementArrayCreateAt(),  
ElementArrayCreateAtHandles(), ElementArrayHeader
```

To create an element array, call the routine **ElementArrayCreate()**. Like **ChunkArrayCreate()**, it takes three arguments: the LMem heap’s handle, the size of each element (or 0 for variable-sized elements), and the size to leave for the array header. The routine allocates a chunk in the LMem heap and initializes it as an element array. There is one significant difference: Element arrays begin with an **ElementArrayHeader**, a structure whose first component is a **ChunkArrayHeader**. If you are allocating free space between the header and the array, make sure to leave enough room for an

ElementArrayHeader. If you do not need to allocate free space, you can pass a header size of zero, as with **ChunkArrayCreate()**.

There is another version of this routine, **ElementArrayCreateAt()**, which creates the element array in a pre-existing chunk. This routine takes three arguments: an optr indicating the chunk, the size of each element, and the size of the header. It creates the element array in the specified chunk, resizing it if necessary. Any data in the chunk may be overwritten (except for whatever data falls in the header area after the **ElementArrayHeader**). There is also a version which takes handles instead of an optr; it is called **ElementArrayCreateAtHandles()**.

16.4

The routine returns the handle of the newly-created element array. It can cause heap compaction or resizing; therefore, all pointers to the heap are invalidated.

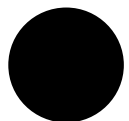
16.4.2.2 Adding an Element

```
ElementArrayAddElement(),  
ElementArrayAddElementHandles(),  
ElementArrayAddReference(),  
ElementArrayAddReferenceHandles()
```

Adding an element to an element array is somewhat different from adding one to a chunk array. To add an element to a chunk array, you merely call the append routine, then write the element into the allocated space. If you want to add an element to an element array, you must first write out the data for the element in a buffer. You then pass the address of this data to **ElementArrayAddElement()**, which compares your new element with the elements already in the array, and copies it into the array if necessary.

ElementArrayAddElement() takes four arguments:

- ◆ An optr indicating the element array;
- ◆ The address of the element to copy into the array;
- ◆ A pointer to a callback comparison routine (see below), or a null pointer to do a byte-wise comparison;
- ◆ A dword of data to pass to the comparison routine.



You may have your own criteria for deciding whether an element should be copied into an array. For example, elements in the array may have three data fields; perhaps you count two elements as matching if the first two data fields match. For this reason, **ElementArrayAddElement()** lets you specify your own comparison routine. The callback routine should be a Boolean routine, declared `_pascal`, which takes three arguments:

- ◆ The address of the element to add;
- ◆ The address of an element in the array to compare the new element to;
- ◆ The callback data dword passed to **ElementArrayAddElement()**.

ElementArrayAddElement() calls the callback routine to compare the new element to each element in the array. If the callback routine ever returns *true*, **ElementArrayAddElement()** has found a matching element in the array; it will increment that element's reference count and return its index. If the callback routine returns *false* for every element, **ElementArrayAddElement()** copies the new element into the array and gives it a reference count of 1. It returns the element's index; the element will keep that index until it is freed. Note that there is no way to specify where in an element array a new element should be added. If there are free spaces in the array, the new element will be created in the first free space; otherwise, it will be appended to the end of the array.

If you want to do a bitwise comparison, pass in a null pointer as the callback routine. **ElementArrayAddElement()** will then do a bitwise comparison of the elements, treating two elements as equal only if every byte matches. The bitwise comparison is implemented as a machine-language string instruction; it is therefore very fast.

If you know that the element you want to add is already in the array, call **ElementArrayAddReference()**. This routine simply increments the reference count of a specified element; it does no comparisons. It is therefore much faster than **ElementArrayAddElement()**.

Both of these routines have counterparts which are passed handles instead of an `optr`; these counterparts are named **ElementArrayAddElementHandles()** and **ElementArrayAddReferenceHandles()**.

16.4.2.3 Accessing Elements in an Element Array

```
ElementArrayElementChanged(),
ElementArrayElementChangedHandles()
```

Elements in element arrays are accessed in almost the same way as elements in chunk arrays. There is one major difference. Each element in an element array begins with a **RefElementHeader** structure, which contains the element's reference count. For this reason, it is a good idea to declare special structures for your elements and have the first component of that structure be the **RefElementHeader** structure (as in the code sample below).

16.4

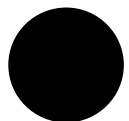
Code Display 16-3 Structure for Element Array Elements

```
/* We need to declare two different structures. One will have all the fields in the
 * element proper. We will use this when we create an element, and we will pass its
 * address to ElementArrayAddElement(). The other will contain two fields: the
 * RefElementHeader structure, and the element body structure. When we dereference
 * an element, we are returned the address of such a structure.
 */

typedef struct {
    word        amount;           /* This has the element's data fields */
    float       interestRate;
    char        description[20];
} MyElementBody;

typedef struct {
    RefElementHeader  header; /* We won't use this—it holds ref count */
    MyElementBody    body;
} MyElement;
```

Note that if you change an element, this may make it identical to another element in the element array; in this case, the two could be combined into one. To check for this situation, call **ElementArrayElementChanged()**. This routine takes four arguments: the optr to the element array, the token for the element changed, a callback comparison routine, and a dword of data which is passed to the callback routine. **ElementArrayElementChanged()** checks to see if the element is identical to any other element in the array. It calls the comparison routine to compare elements. (You can force a bitwise comparison by passing a null function pointer.) If it matches another



element, the two elements will be combined; i.e., the element passed will be deleted, and the matching element will have its reference count increased appropriately. The matching element's token will be returned; you will have to change any references to the old element appropriately. If no match is found, the token which was passed will be returned. The version which takes handles is called **ElementArrayElementChangedHandles()**.

16.4

16.4.2.4 Removing An Element From An Element Array

```
ElementArrayRemoveReference(),  
ElementArrayRemoveReferenceHandles(),  
ElementArrayDelete(), ElementArrayDeleteHandles()
```

When you want to remove an element from an element array, you should ordinarily call **ElementArrayRemoveReference()**. This routine decrements the element's reference count; it does not, however, delete the element unless the reference count reaches zero (i.e. the last reference to the element has been deleted).

This routine takes four arguments: the optr of the array, the index of the element, a pointer to a callback routine, and a dword-sized constant to be passed to it. There may be certain bookkeeping tasks you want to perform when an element is actually being deleted but not when it is just having its reference count decremented. In this case, you can pass the address of a callback routine, which will be called on any element to be deleted just before the deletion occurs. After the callback routine returns, the element will be removed. If you do not need to have a callback routine called, pass a null function pointer. As noted, when an element is removed, it is actually just resized to zero; that way the index numbers of following elements are preserved.

If you want to delete an element regardless of its reference count, call **ElementArrayDelete()**. This routine takes two arguments, namely the optr indicating the array and the index of the element to be deleted. It does not take a callback routine; perform any necessary bookkeeping before you call it.

Both of these routines have counterparts which take handles; these counterparts are named **ElementArrayRemoveReferenceHandles()** and **ElementArrayDeleteHandles()**.

16.4.2.5 The “Used Index” and Other Index Systems

```
ElementArrayGetUsedCount(),
ElementArrayGetUsedCountHandles(),
ElementArrayUsedIndexToToken(),
ElementArrayUsedIndexToTokenHandles(),
ElementArrayTokenToUsedIndex(),
ElementArrayTokenToUsedIndexHandles()
```

16.4



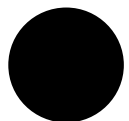
Advanced Topic

Very few applications will ever need to use these routines.

Sometimes its useful to have a special index system for element arrays. Perhaps you would like the used elements to be numbered sequentially, that is, the first “used” element would be element “zero,” even if there were free elements before it. This would require setting up a second index system, besides the one used by the element array routines. GEOS provides routines with this functionality.

To find out the number of elements in an element array, call **ElementArrayGetUsedCount()**. This routine can return either the number of elements in use or the number of “in use” elements which satisfy any arbitrary criteria. The routine takes three arguments: the optr to the element array, a dword of data which is passed to a callback routine, and a pointer to a Boolean callback routine. That callback routine should itself take two arguments: a pointer to an element, and the dword passed to **ElementArrayGetUsedCount()**. The callback routine is called once for each “in use” element. The callback should return *true* if the element should be counted; otherwise, it should return *false*. For example, the callback routine might return *true* if the element is longer than 10 bytes; in this case, **ElementArrayGetUsedCount()** would return the number of elements which are longer than 10 bytes. To have every used element counted, pass a null function pointer. The version of this routine which takes handles is called **ElementArrayGetUsedCountHandles()**.

If you use a different indexing scheme, you will need a way to translate the index into the normal element array token. To do this, call the routine **ElementArrayUsedIndexToToken()**. This routine takes four arguments: the optr of the element array, the index count, a dword (which is passed to the callback routine), and a callback routine. The callback routine is of the same format as the callback routine passed to **ElementArrayGetUsedCount()**; it should return *true* if the element meets some criterion. **ElementArrayUsedIndexToToken()** translates the index



passed into the element array's token for that element. For example, if the callback routine returns *true* for elements which are longer than 10 bytes, and you pass an index of five, **ElementArrayUsedIndexToToken()** will return the token for the sixth element in the element array which is longer than 10 bytes. (Remember, all indexes are zero-based.) Again, passing a null function pointer makes the routine count all “in-use” elements. The version which takes the element array's handles is called **ElementArrayUsedIndexToTokenHandles()**.

16.4

To translate a token back into this kind of index, call **ElementArrayTokenToUsedIndex()**. This routine takes four arguments: the optr to the element array, an element token, a callback routine (as with the other routines in this section), and a dword which is passed along to the callback routine. The routine finds the element whose token was passed and returns the index it would have under the indexing system defined by the callback routine. Again, passing a null function pointer makes the routine count every “in-use” element. The routine which takes handles is called **ElementArrayTokenToUsedIndexHandles()**.

16.4.3 Name Arrays

Applications can build on chunk arrays and element arrays in many ways. The chunk array library includes one example of an elaboration on these structures: the *name array*. The name array is a special kind of element array in which elements can be accessed by a “name” label as well as by a token. Elements in a name array are of variable size. Each element is divided into three sections: The first is the **RefElementHeader**; every element in an element array must begin with one of these (and the name array is a kind of element array). The second is the data section. The data section is the same size for every element in a given name array; this size may be anything from zero bytes up to a maximum of NAME_ARRAY_MAX_DATA_SIZE (64 bytes). The data section is followed by a “name” section. This section contains a sequence of bytes of any length up to a maximum of NAME_ARRAY_MAX_NAME_SIZE (256 bytes). The name may contain nulls and need not be null terminated. You can translate a name into the element's token by calling **NameArrayFind()** (described below).

Note that creating elements in a name array, as in any element array, requires a search through all elements; it thus takes linear time. Furthermore, translating a name into a token also requires a linear search through the elements, and thus also takes linear time. Name arrays thus become slow if they grow too large. Accessing an element by token, however, still takes constant time.

16.4.3.1 Creating a Name Array

16.4

```
NameArrayCreate(), NameArrayCreateAt(),
NameArrayCreateAtHandles(), NameArrayAdd(),
NameArrayAddHandles(), NameArrayHeader, NameArrayAddFlags
```

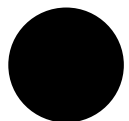
Creating a name array is much like creating an element array. Every name array must begin with a **NameArrayHeader**. This structure has the following definition:

```
typedef struct {
    ElementArrayHeader    NAH_meta;
    word                  NAH_dataSize;
} NameArrayHeader;
```

This structure contains one new field, namely *NAH_dataSize*. This field specifies the size of the data area of each element; the name area is of variable size. You may examine this field at will, but you may not change it. You may set up a fixed data area between the **NameArrayHeader** and the elements. The usual way to do this is to define a structure whose first element is a **NameArrayHeader** structure.

To create a name array, call the routine **NameArrayCreate()**. This routine is passed three arguments:

- ◆ The global handle of an LMem heap. The name array will be created in this block.
- ◆ The size of the data area in each element. The total size of the element will vary, depending on the size of the name. Remember, there is a three byte **RefElementHeader** at the start of every element (before the data section).



- ◆ The size of the header structure for the name array. If you will not need a fixed data area, you can pass a size of zero, and enough space will automatically be left for a **NameArrayHeader**.

The routine allocates a chunk in the specified heap, initializes a name array in that chunk, and returns the chunk's handle. If it fails for any reason, it returns a null chunk handle. Since the routine allocates a chunk, all pointers to the LMem heap are invalidated.

16.4

If you want to create a name array in a specific chunk, call **NameArrayCreateAt()**. This routine is almost the same as **NameArrayCreate()**. However, instead of being passed a memory handle, **NameArrayCreateAt()** is passed an optr to a chunk. The name array will be created in that chunk. Any data in that chunk (outside of the fixed data area) will be destroyed. Note that if the chunk is too small for the name array, **NameArrayCreateAt()** will resize it; thus, pointers to the LMem heap may be invalidated. There is a version of this routine which takes the chunk's global and chunk handles instead of its optr; this routine is called **NameArrayCreateAtHandles()**.

To create an element, call **NameArrayAdd()**. This routine creates an element and copies the data and name into it. The routine takes five arguments:

- ◆ The optr to the name array.
- ◆ A pointer to an array of characters containing the element's name.
- ◆ The length of the name, in bytes. This many characters will be copied into the name. If you pass a length of zero, bytes will be copied until a null byte is reached (the null will not be copied).
- ◆ A word-length set of **NameArrayAddFlags**. Only one flag is currently defined, namely **NAAF_SET_DATA_ON_REPLACE**. This flag is described below.
- ◆ A pointer to the data section. The data will be copied into the new element. (The length of the data portion was specified when the name array was created.)

NameArrayAdd() allocates the element, copies in the data and name, and returns the element's token. If an element with the specified name already exists, **NameArrayAdd()** will not create a duplicate. Instead, if the flag **NAAF_SET_DATA_ON_REPLACE** was passed, **NameArrayAdd()** will copy

the new data section into the existing element; if the flag was not passed, it will leave the existing element unchanged. In either case, it will return the existing element's token and increment its reference count. If an element is added, the name array may have to be resized; therefore, pointers into the chunk array will be invalidated. There is a version in which the name array is specified by its global and chunk handles; this version is called **NameArrayAddHandles()**.

16.4

Code Display 16-4 Allocating a Name Array

```

/* We want a fixed data space, so we define our own header structure. */
typedef struct {
    NameArrayHeader MNAH_meta;      /* Must begin with a NameArrayHeader!!! */
    char *          MNAH_comments[32];
} MyNameArrayHeader;

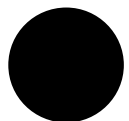
/* The data section of the name array will be this structure: */
typedef struct {
    double          MDSS_myDataFloat;
    int             MDSS_myDataInts[20];
} MyDataSectionStruct;

/* Every element in the name array will have this structure: */
typedef struct {
    RefElementHeader MES_header;     /* For internal use */
    MyDataSectionStruct MES_data;
    char              MES_name[];    /* We don't know how long this
                                     * will actually be */
} MyElementStruct;

MemHandle      myLMemHeap;          /* Assume this is initialized */
ChunkHandle    myNameArray;

/* Sample call to NameArrayCreate() */
myNameArray = NameArrayCreate(myLMemHeap, sizeof(MyDataSectionStruct),
                              sizeof(MyNameArrayHeader));

```



16.4.3.2 Accessing Elements in a Name Array

```
NameArrayFind(), NameArrayFindHandles(),  
NameArrayChangeName(), NameArrayChangeNameHandles()
```

Name array routines can be accessed with all the routines used for accessing element arrays. However, a few special purpose routines are also provided.

If you know the name of an element and want a copy of its data, call **NameArrayFind()**. This routine is passed four arguments:

- ◆ The optr to the name array.
- ◆ A pointer to a character buffer. The buffer should contain the name of the element sought.
- ◆ The length of the name. If a length of zero is passed, the name is considered to be null terminated (the trailing null is *not* part of the name).
- ◆ A pointer to a return buffer. The data portion of the element will be copied to this location.

NameArrayFind() will do a linear search through the elements. If it finds one with the name specified, it will return that element's token and copy the data portion into the return buffer. If there is no element with the specified name, **NameArrayFind()** will return the constant `CA_NULL_ELEMENT`. The routine **NameArrayFindHandles()** is identical, except that the name array is specified by its global and chunk handles.

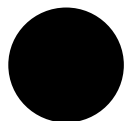
To change an element's name, call **NameArrayChangeName()**. This routine is passed four arguments:

- ◆ The optr to the name array.
- ◆ The token of the element whose name will be changed.
- ◆ A pointer to a character buffer containing the new name for the element.
- ◆ The length of the new name. If a length of zero is passed, the name is considered to be null terminated (the trailing null is *not* part of the name).

NameArrayChangeName() changes the element's name. If the new name is longer than the old, the element will have to be resized; this will invalidate pointers within that block. **NameArrayChangeNameHandles()** is

identical, except that the name array is specified by its global and chunk handles.

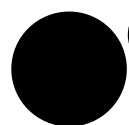
16.4



Local Memory

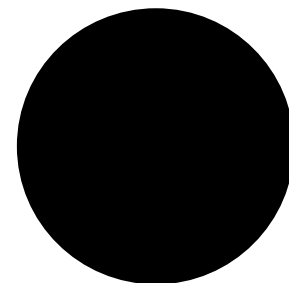
608

16.4



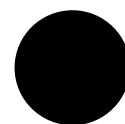
Concepts book

File System



17

17.1	Design Philosophy	611
17.2	File System Overview	613
17.3	Disks and Drives	616
17.3.1	Accessing Drives	616
17.3.2	Accessing Disks	619
17.3.2.1	Registering Disks	620
17.3.2.2	Getting Information about a Disk	621
17.3.2.3	Saving and Restoring a Disk Handle	624
17.3.2.4	Other Disk Utilities	626
17.4	Directories and Paths	631
17.4.1	Standard Paths	632
17.4.2	Current Path and Directory Stack	636
17.4.3	Creating and Deleting Directories	639
17.5	Files	641
17.5.1	DOS Files and GEOS Files	641
17.5.2	Files and File Handles	643
17.5.3	GEOS Extended Attributes	643
17.5.4	File Utilities	653
17.5.5	FileEnum()	655
17.5.6	Bytewise File Operations	661
17.5.6.1	Opening and Closing Files	662
17.5.6.2	Reading From and Writing To Files	666
17.5.6.3	Getting and Setting Information about a Byte File	668
17.5.6.4	Data-Access Synchronization	669





Every operating system needs a way to interact with files. Files are used to hold both data and executable code. They are also the simplest way of transferring data from one computer to another.

GEOS provides powerful file-management functionality. It runs on top of a disk-operating system, and uses that DOS to read files from different media. Applications which run under GEOS need only interact with the GEOS file-management system; they are insulated from the differences between versions of DOS. They are also insulated from the differences between various file-storage media: CD-ROM drives, network file servers, and floppy and hard disks all present the same interface. 17.1

The GEOS file system provides functionality that many versions of DOS do not have. It allows the use of virtual directories, so (for example) the system's FONT directory could actually comprise several physical directories. GEOS files have functionality which DOS files lack. For example, GEOS provides support for file-sharing and data-access synchronization across networks. Nevertheless, GEOS lets applications access standard DOS files and directories when desired.

You may not need to use much of the file system directly. The document control and file-selector objects can let the user select and open files transparently to the application. Many applications will never need to negotiate the directory structure.

Before reading this section, you should have read "System Architecture," Chapter 3, and "First Steps: Hello World," Chapter 4. You should also be familiar with GEOS handles (discussed in "Handles," Chapter 14).



17.1 Design Philosophy

The GEOS file system was designed to meet two goals. First, the file system should insulate geodes from differences in hardware, making them device independent; and second, it should give geodes all the functionality they might plausibly want without being cumbersome.



One of the hallmarks of GEOS is its ability to insulate applications from differences in hardware. The GEOS file system plays a large role in this. Geodes which need access to files make their requests to the GEOS file system, which accesses the actual storage media. The virtues of this approach are several:

- ◆ Geodes can rely on a consistent API for accessing media. When an application needs to open a file, it uses exactly the same techniques whether the file resides on a hard or floppy disk, a network drive, or some other medium.
- ◆ GEOS can easily expand to make use of new technologies. To support a new storage medium, GEOS just needs a new device driver; all existing application binaries will automatically work with the new device.
- ◆ Applications need not worry about what DOS the system might be running under. The GEOS file system does all interaction with the DOS and automatically takes advantages of each DOS' strengths. To support new versions of DOS, GEOS just needs new drivers.

GEOS is also flexible. Flexibility means more than just being able to use new technology in the same old ways; it means providing functionality to serve unforeseen purposes. GEOS provides this:

- ◆ The file system allows such advanced functionality as virtual paths and directories. Files can thus be distributed across various media (for example, templates and fonts on CD-ROM drives, company files on a network drive, other documents on local hard disks) transparently to both user and application.
- ◆ Other functionality can be added. For example, all GEOS files can have names up to 32 characters long, regardless of strictures on individual media. The file system maps the GEOS "virtual names" to device-dependent "native names" transparently to the user and application.

While GEOS provides this advanced functionality, it still lets geodes access the raw files. If a geode wants to access any file as a sequence of bytes (the DOS model), it may do so. Similarly, geodes may work with physical DOS directories if they wish to.

17.2 File System Overview

The GEOS file system manages all access to files on any type of storage. Whenever a geode needs to access a data file, it calls a file system routine. The file system makes any necessary calls to the computer's DOS. Like much of the GEOS system, it is driver-based; this makes it easy to expand GEOS to accommodate new technologies.

When a geode needs to access a file, it makes a call to the file system. The file system sends requests through the appropriate driver. For example, if the geode needs access to a file on a local hard or floppy disk drive, the file system will send commands through a DOS driver, which will in turn issue appropriate commands to the DOS itself. Similarly, if the file resides on a network drive, the file system will send commands through an appropriate network driver, which will translate them into corresponding commands to the network server. (See Figure 17-1 below.)

17.2

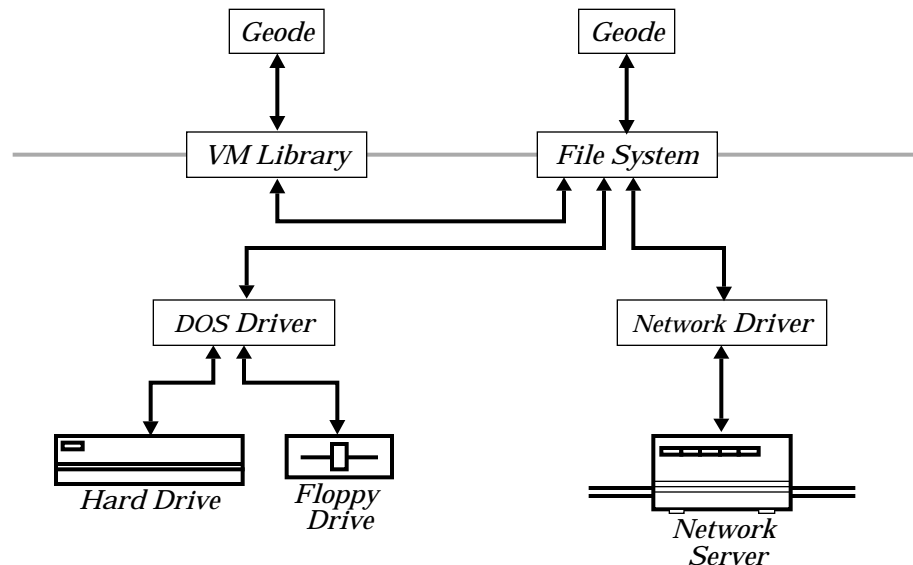


Figure 17-1 Accessing Files Through the File System

The application interacts only with the file system or a library built on top of it (such as the VM library). Everything below that is transparent to both application and user. Arrows depict the flow of commands and data.



Every storage device is known as a *drive*. A drive can be either fixed-media or movable-media. Fixed media drives are always associated with a specific storage medium; in effect, the drive is the same as the storage. The primary example of this is the conventional hard disk. Movable-media drives use a potentially unlimited number of storage objects. Examples include floppy disk drives, CD-ROM drives, and cartridge hard drives.

One way of organizing storage objects is to divide them into *volumes*. Ordinarily, every disk (floppy or hard) is a single volume; however, hard disks are sometimes divided into several volumes. Network storage devices are frequently divided into several volumes. If a single drive is partitioned into several volumes, DOS and GEOS treat each volume as a separate drive. In this chapter, the words *volume* and *disk* will be used interchangeably.

Individual volumes are organized with *directories*. A directory may contain several different files or directories. There is not usually a limit on the number of files a directory may contain; however, no two files or directories belonging to a given directory may have the same name.

Every volume is organized in a *directory tree*. The topmost directory is known as the *root*; it is unnamed, since every volume contains exactly one root directory. The root may contain files or directories. Those directories may themselves contain files or directories, and so on. The total number of files is usually limited only by the size of the storage device.

To specify a file, you need to specify three things: The volume the file resides on, the directory to which the file belongs, and the file's name. In most DOS environments, a volume is generally specified with a letter followed by a colon (for example, the first hard disk is generally specified as "C:"). For non-fixed media (e.g. floppy disks), the letter actually specifies the drive; the volume is presumed to be in that drive. In GEOS, volumes and drives have different identifying systems. Every drive is identified by a *drive number*. This is a positive integer; the first drive has a number of zero, the second is drive one, and so on. Every volume is identified by a token (the *disk handle*).

Specifying the directory is a little trickier. There may be many directories with the same name on a given volume. For this reason, the directory is specified with a *path*. The path begins with the root directory; the root is followed by the name of one of the directories belonging to the root; that directory is followed by the name of one of its subdirectories; and so on, until the desired directory is reached. Since all the directories belonging to a given

directory must have unique names, the path is guaranteed to uniquely specify a directory on a volume. For example, you might specify a directory as “\GEOWORKS\DOCUMENT\MEMOS”; this would indicate a directory named “MEMOS,” which belongs to a directory named “DOCUMENT,” which in turn belonged to a directory named “GEOWORKS,” which was at the root level.

The file itself is identified by its name. Since its directory has been uniquely specified by the volume and path, the name is guaranteed to specify at most one file in that directory. Each disk-operating system has its own conventions about how file names can be formed; for example, MS-DOS requires file names to have a “name” portion of at most eight letters, followed by an “extension” of at most three letters (known as the “FILENAME.EXT” convention). (See Figure 17-2.)

17.2

For convenience, most operating systems let you specify a “working directory.” This is a combination of a volume and a path. If you have a working directory, you can omit either the volume or the path from a file specification, and the volume and/or path of the working directory will be used. This is called a “relative path,” i.e. a path that is relative to the working directory instead of the root directory. (Note that if you specify a disk handle for any operation, you must pass an absolute path which begins at the root of that disk or standard path.) GEOS allows every thread to have its own working directory; each thread can also save paths on a stack, letting the

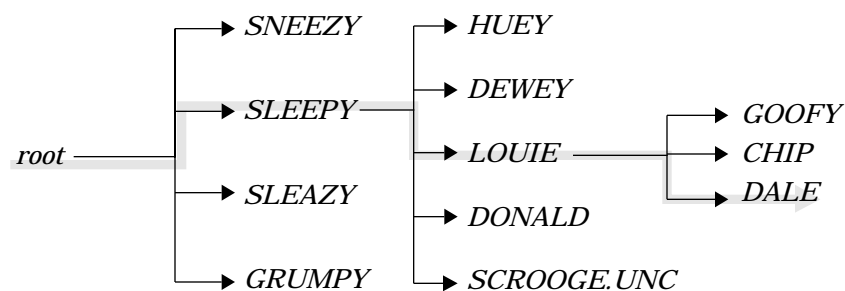


Figure 17-2 A Path Through a Directory Tree

In MS-DOS, the highlighted path would be described as “\SLEEPY\LOUIE\DALE”.



thread switch directories easily. For more details, see section 17.4.2 on page 636.

Some disk-operating systems, such as MS-DOS, allow a working directory for each drive. GEOS does not allow this; you may have only one working directory at a time, regardless of how many drives you have. If you need to switch back and forth between directories on different drives, you can use the directory stack (see section 17.4.2 on page 636).

17.3

17.3 Disks and Drives

GEOS provides an easy interface to storage devices. Every drive (or analogous storage device) is identified by a token. Every volume is also identified by a token. This enables you to easily move from one volume to another.

You generally need only worry about disks and drives when you are opening a file. Once you have a file open, you can access the file by its file handle without paying attention to the disk it resides on. The GEOS file system will automatically prompt the user to change disks whenever necessary.

Note that if you use the document control object to open files, you will probably never have to worry about disks and drives. The document control automatically presents a File Selector dialog box to the user, letting the user navigate among disks and directories; when the user selects a file, the document control will automatically open the file and return its handle to the application.

17.3.1 Accessing Drives

```
DriveGetStatus(), DriveGetExtStatus(),  
DriveGetDefaultMedia(), DriveTestMediaSupport(),  
DriveGetName(), DriveStatus, DriveType, MediaType
```

Most systems running GEOS have access to a number of different drives. With the exception of network drives, the drives available will usually not change during an execution of GEOS, although the volumes mounted on the

drives can change. Every drive is accessed by its *drive number*. This token is a byte-length integer value.

When you wish to open a file, you must specify its volume, not its drive. This is because the volume mounted on a drive can change frequently and without warning.

GEOS provides routines to get information about a drive. To get general information about a drive, call the routine **DriveGetStatus()**. This routine takes the drive number and returns a word-length set of **DriveStatus** flags (defined in **drive.h**). If an error condition exists, such as the drive you request not existing, it returns zero. The following flags may be returned:

17.3

DS_PRESENT

This flag is set if the physical drive exists, regardless of whether the drive contains a disk.

DS_MEDIA_REMOVABLE

This flag is set if the disk can be removed from the drive.

DS_NETWORK

This flag is set if the drive is accessed over a network (or via network protocols), which means the drive cannot be formatted or copied.

DS_TYPE

This is a mask for the lowest four bits of the field. These bits contain a member of the **DriveType** enumerated type.

The lowest four bits of the word contains a member of the **DriveType** enumerated type. The field can be accessed by masking out all the bits except for those set in **DS_TYPE**. **DriveType** comprises the following values:

DRIVE_5_25 Drive uses 5.25-inch floppy disks.

DRIVE_3_5 Drive uses 3.5-inch floppy disks.

DRIVE_FIXED

Drive uses some kind of fixed disk (e.g. a hard drive).

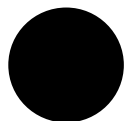
DRIVE_RAM Drive is chip-based (either RAM or ROM).

DRIVE_CD_ROM

Drive uses read-only optical disks.

DRIVE_8

Drive uses 8-inch floppy disks.



DRIVE_UNKNOWN

Drive type is unknown.

DriveGetStatus() returns the information most often needed about a drive. However, you may sometimes need more obscure information. For this reason, GEOS provides the routine **DriveGetExtStatus()**. Like **DriveGetStatus()**, it takes the drive number as an argument and returns a word of flags; however, it returns additional flags. The flags returned by **DriveGetStatus()** are set in the lower byte of the returned word; special additional flags are set in the upper byte. Like **DriveGetStatus()**, **DriveGetExtStatus()** returns zero if the drive specified is invalid. The following flags are defined for the upper byte:

DES_LOCAL_ONLY

This flag is set if the device cannot be viewed over a network.

DES_READ_ONLY

This flag is set if the device is read only, i.e. no data can ever be written to a volume mounted on it (e.g., a CD-ROM drive).

DES_FORMATTABLE

This flag is set if disks can be formatted in the drive.

DES_ALIAS This flag is set if the drive is actually an alias for a path on another drive.

DES_BUSY This flag is set if the drive will be busy for an extended period of time (e.g., if a disk is being formatted).

Many disk drives can take a variety of disks. For example, high-density 3.5-inch drives can read and write to either 720K disks or 1.44M disks. Every drive has a “default” media type. When you format a disk in that drive, it will, by default, be formatted to the default size. To find out the default disk type, call the routine **DriveGetDefaultMedia()**. This routine takes one argument, namely the drive number. It returns a member of the **MediaType** enumerated type. **MediaType** has the following possible values:

MEDIA_160K, MEDIA_180K, MEDIA_320K, MEDIA_360K

These are all sizes used by 5.25-inch disks.

MEDIA_720K

This is the size of a regular 3.5-inch disk.

MEDIA_1M2

This is the size of a high-density 5.25-inch disk.

MEDIA_1M44

This is the size of a high-density 3.5-inch disk.

MEDIA_2M88

This is the size of an ultra-high-density 3.5-inch disk.

MEDIA_FIXED_DISK

This is returned for all fixed disks.

MEDIA_CUSTOM

This is returned if none of the other values is appropriate. For example, it is returned for CD-ROM drives. 17.3

MEDIA_NONEXISTENT

This is returned if the drive specified does not contain a disk. This value is defined to be equal to zero.

If you want to find out if a drive can accommodate a certain kind of disk, call the routine **DriveTestMediaSupport()**. This Boolean routine takes two arguments: a drive number and a member of the **MediaType** enumerated type. If the drive supports that medium, the routine returns *true* (i.e. non-zero); otherwise, it returns *false* (i.e. zero).

To find out the name of a given drive, call **DriveGetName()**. This routine is passed three arguments: a drive number, a pointer to a character buffer, and the size of that buffer. **DriveGetName()** writes the drive's name to the buffer as a null-terminated string; it returns a pointer to that trailing null. If the buffer was not large enough, or the drive does not exist, it returns a null pointer.

17.3.2 Accessing Disks

Applications will work with disks more than they will work with drives. Once a geode knows a disk's handle, it can ignore such questions as whether the disk is in a drive; it need merely provide the disk's handle. If necessary, the system will prompt the user to insert the disk in the appropriate drive.

17.3.2.1 Registering Disks

```
DiskRegisterDisk(), DiskRegisterDiskSilently()
```

GEOS automatically keeps track of all disks used. The first time a disk is accessed in a session, it is *registered*. This means that it is assigned a *disk handle*.

17.3

The disk handle records certain information, such as the disk's volume name and whether the disk is writable. It also notes in which drive the disk was last inserted; if the system prompts the user to reinsert the disk, it will insist on that drive. A disk is automatically reregistered when certain actions are performed which might invalidate a disk's handle-table entry; for example, it is reregistered if it is formatted. It is also reregistered if someone tries to write a file to a disk which is marked read-only; the user may have ejected the disk and removed its write-protect tab. Note that reregistering a disk does not change its handle; it just brings GEOS's information about the disk up-to-date.

Note that the disk handle is not a reference to the global handle table; thus, Swat commands like **phandle** will not work with disk handles. Disk handles should always be treated as opaque 16-bit tokens.

You can specifically instruct the system to register a disk by calling the routine **DiskRegisterDisk()**. The routine is passed a single argument, namely the drive number. If the disk has an entry in the disk table, the routine will not reregister the disk; it will just return the disk's handle. If the disk has no entry in the table, the system will create an entry and register the disk. In this case, also, the routine will return the (new) disk handle. If the routine fails (for example, because there is no valid disk in the specified drive, or the drive itself does not exist), it returns a null handle.

When a disk is registered, the system notes the volume label. This label is used when the system has to prompt the user to insert a disk. If an unlabeled disk is inserted, the system will choose an arbitrary label for the volume (e.g. "UNNAMED1"). The system does not actually write this label to the disk; the label is used by the system and discarded when the session ends. Ordinarily, the system will present an alert box to inform the user about the temporary label. You can suppress this notification by calling the system routine **DiskRegisterDiskSilently()**. This routine has the same arguments and return values as **DiskRegisterDisk()**.

17.3.2.2 Getting Information about a Disk

```
DiskGetVolumeInfo(), DiskGetVolumeFreeSpace(),
DiskGetDrive(), DiskGetVolumeName(), DiskFind(),
DiskCheckWritable(), DiskCheckInUse(), DiskCheckUnnamed(),
DiskForEach(), DiskInfoStruct, DiskFindResult
```

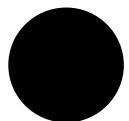
GEOS provides many routines to get information about disks. If geodes call disk routines at all, they are most likely to call these. Most of these routines are passed the handle of the disk. If you know the disk's volume label and need to find out its handle, call the routine **DiskFind()** (described below). If you know the disk is currently in a drive and you need to find out its handle, register the disk (see "Registering Disks" on page 620). Note that any routine which is passed a disk handle can be passed a standard path constant; in this case, the routine will give you information about the disk containing the **geos.ini** file.

17.3

The basic disk-information routine is **DiskGetVolumeInfo()**. This returns information about the size of the disk and the amount of free space available. The routine is passed two arguments: the disk handle and a pointer to a **DiskInfoStruct** structure (shown below). The routine fills in the fields of the **DiskInfoStruct** structure and returns zero if it was successful. If it fails for any reason, it returns an error value and sets the thread's error value (which can be recovered with **ThreadGetError()**). The usual error value returned is the constant **ERROR_INVALID_VOLUME**.

Code Display 17-1 The DiskInfoStruct Structure

```
typedef struct {
    word        DIS_blockSize; /* # of bytes in a block; smallest size
                               * file system can allocate at once */
    sdword      DIS_freeSpace; /* # of bytes free on disk */
    sdword      DIS_totalSpace; /* Total size of the disk in bytes */
    char        DIS_name[VOLUME_BUFFER_SIZE];
                               /* Volume name; if disk is unnamed, this
                               * is the temporary name. String is
                               * null-terminated. */
} DiskInfoStruct;
```



If you just want to know a disk's name, call **DiskGetVolumeName()**. This routine takes two arguments: the disk handle and the address of a character buffer. (The buffer must be at least `VOLUME_NAME_LENGTH_ZT` characters long.) It writes the volume name to the buffer as a null-terminated string, and it returns the buffer's address. If the volume is unnamed, **DiskGetVolumeName()** writes the temporary volume name.

Note that all the routines which return a volume's name will return the temporary name if the volume is unnamed. For this reason, if you want to find out if a volume is unnamed, you must use a special purpose routine, namely **DiskCheckUnnamed()**. This Boolean routine is passed the disk's handle. If the volume does not have a permanent label, the routine returns *true*; otherwise, it returns *false*.

If you want to know how much free space is available on a disk, call the routine **DiskGetVolumeFreeSpace()**. The routine is passed the disk handle; it returns (as a dword) the number of free bytes available. If the volume is currently read-only (e.g. a floppy disk with the write-protect tab set), it returns the amount of space that would be available if the volume were made read/write. If the volume is, by its nature, not writable (e.g. a CD-ROM disk), the routine will return zero. It will also return zero if an error condition occurs; in this case, it will also set the thread's error value.

If you want to know what drive a volume is associated with, call **DiskGetDrive()**. This routine takes one argument, namely the volume's disk handle. It returns the number of the drive which had that disk. Note that it will return this value even if that drive is no longer usable.

If you know the label of a volume which has been registered and you need to find out its handle, call the routine **DiskFind()**. The routine takes two arguments: the address of a null-terminated string containing the volume name and a pointer to a variable of the **DiskFindResult** enumerated type. It will return the disk's handle; if no disk with the specified name has been registered, it will return a null handle. **DiskFindResult** has the following possible values:

DFR_UNIQUE

Exactly one volume with the specified name has been registered. The handle of that volume was returned.

DFR_NOT_UNIQUE

Two or more volumes with the specified label have been

registered. The handle of an arbitrary one of these volumes was returned. If you want to find the handles of all of these disks, call **DiskForEach()**, described below.

DFR_NOT_FOUND

No disk with the specified label has been registered. A null handle was returned.

To check if a volume is writable, call the Boolean routine

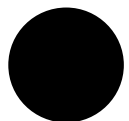
DiskCheckWritable(). The routine takes one argument, the disk's handle. If the disk is writable, the routine returns *true* (i.e. non-zero). If the disk is non-writable, the routine returns *false* (i.e. zero).

17.3

To see if a disk is being used by any threads, call **DiskCheckInUse()**. The routine takes one argument: the disk's handle. It returns *true* (i.e. non-zero) if a file on the disk is open or if any thread has a directory on that disk in its directory stack (see section 17.4.2 on page 636). If neither condition applies, the routine returns *false* (i.e. zero).

If you want to perform an action on every disk, call **DiskForEach()**. This routine takes one argument, a pointer to a Boolean callback routine. The callback routine should take a single argument, the handle of a disk. **DiskForEach()** calls the callback routine once for every registered disk. It passes the disk handle to the callback routine, which can take any action it wants; for example, it could call one of the other disk-information routines. The callback routine can make **DiskForEach()** halt prematurely by returning a non-zero value. If the callback routine forced an early halt, **DiskForEach()** returns the last disk handle which had been passed to the callback routine; otherwise it returns a null handle. This routine is commonly called to look for a specific disk. To do this, simply have the callback routine check each disk to see if it is the one sought; if it is, simply return *true*, and **DiskForEach()** will return that disk's handle.

DiskForEach() does not need to examine the actual disks; it works from the information the file-system stores about all registered disks. This means that **DiskForEach()** will not have to prompt the user to insert any disks. Of course, the callback routine may need to examine the disks, in which case the user will be prompted when necessary.



17.3.2.3 Saving and Restoring a Disk Handle

`DiskSave()`, `DiskRestore()`, `DiskRestoreError`

17.3

A disk does not necessarily have the same handle from one execution of GEOS to another. This can pose a problem for an application which is restarting from a state file. In order to reopen a file, it has to know the file's location. If it knows the file's location relative to a standard path, there is no problem, since the application can use the standard path constant in the place of a disk handle. If the file is not in a standard path, the application will need some way of figuring out the disk's handle on restart.

For this reason, GEOS provides **DiskSave()** and **DiskRestore()**.

DiskSave() saves information about a disk in an opaque data structure.

DiskRestore() reads such a data buffer and returns the handle of the disk described; it even arranges to prompt the user if the disk has not been registered yet.

To save a disk handle, call **DiskSave()**. This routine takes three arguments:

- ◆ The disk handle.
This may be a standard path.
- ◆ A pointer to a buffer.
DiskSave() will write opaque data to that buffer; you will need to pass that data to **DiskRestore()** to restore the handle.
- ◆ A pointer to an integer.
When you call **DiskSave()**, that integer should contain the size of the buffer (in bytes). When **DiskSave()** exits, the integer will contain the size of the buffer needed or used (as described below).

If **DiskSave()** was successful, it will return *true*. The integer parameter will contain the size of the buffer actually needed; for example, if the buffer had been 100 bytes long and **DiskSave()** returns 60, you can safely free the last 40 bytes in the buffer. If **DiskSave()** failed, it will return *false*. If it failed because the buffer was too small, it will write the size needed into the integer passed; simply call **DiskSave()** again with a large enough buffer. If **DiskSave()** failed for some other reason (e.g. the disk belongs to a drive which no longer exists), it will write a zero value to the integer.

To restore a disk, call **DiskRestore()**. This routine takes two arguments:

- ◆ A pointer to the opaque data structure written by **DiskSave()**.
- ◆ A pointer to a callback routine. The callback routine is called if the user must be prompted to insert the disk. If you pass a null function pointer, **DiskRestore()** will fail in this situation.

If the disk in question has already been registered or is currently in its drive, **DiskRestore()** will return its handle. If the disk is not registered and is not in any drive, **DiskRestore()** will call the callback routine. The callback routine should accept the following four arguments:

17.3

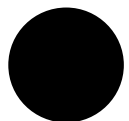
- ◆ A pointer to a null-terminated string containing the drive name, specified according to the DOS convention (e.g. "C:").
- ◆ A pointer to a null-terminated string containing the disk volume label.
- ◆ The **DiskRestoreError** (see below) which would be returned if the callback routine hadn't been called.
- ◆ A pointer to a pointer to the opaque data structure. If the callback routine causes the data structure to move, it should fix up the pointer to point to the new location. Note that the two strings (the drive name and the disk label) are located within this data structure; thus, if you cause this structure to move, both of those pointers will be invalidated.

If the callback routine believes the user inserted the correct disk, it should return `DRE_DISK_IN_DRIVE`. Otherwise, it should return a **DiskRestoreError** constant. In this case, **DiskRestore()** will fail and set the thread's error value to the constant specified. If the callback routine returns an error, that error will generally be `DRE_USER_CANCELLED_RESTORE`.

If **DiskRestore()** is successful, it will return the disk handle; this may be different from the disk's handle in the previous execution. You may now free the data buffer, if you like. If **DiskRestore()** fails, it will return a null handle and set the thread's error value. There are several different **DiskRestoreError** values; they are listed below.

`DRE_DRIVE_NO_LONGER_EXISTS`

The disk had last been used in a drive which is no longer attached to the system (or the appropriate file-system driver is no longer present).



DRE_REMOVABLE_DRIVE_DOESNT_HOLD_DISK

The disk was in a removable-media drive, and that drive doesn't contain the disk.

DRE_USER_CANCELLED_RESTORE

A callback routine was called and returned this value to **DiskRestore()**.

DRE_COULDNT_CREATE_NEW_DISK_HANDLE

DiskRestore() found the disk in the drive but was for some reason unable to create the disk handle.

DRE_REMOVABLE_DRIVE_IS_BUSY

The appropriate drive exists but is unavailable due to some time-consuming operation (e.g. a disk format).

17.3.2.4 Other Disk Utilities

`DiskSetVolumeName()`, `DiskFormat()`, `DiskCopy()`, `FormatError`, `DiskCopyCallback`, `DiskCopyError`

GEOS provides several utilities for working with disks. These utilities allow geodes to copy disks, format them, and change their volume names. Most applications will never need to use these utilities; they can rely on the users to take care of disk formatting with an application like GeoManager.

However, some applications will want to make use of them. For example, an archiving program might automatically format storage disks and give them appropriate labels.

Changing a Volume Name

If you want to set or change a volume's name, you should call **DiskSetVolumeName()**. This routine takes two arguments: the volume's handle and the address of a null-terminated string (containing the new volume name). If it is able to change the volume's name, it returns zero; otherwise, it returns an error code. It sets or clears the thread's error value appropriately. The following error codes may be returned:

ERROR_INVALID_VOLUME

An invalid disk handle was passed to the routine.

ERROR_ACCESS_DENIED

For some reason, the volume's name could not be changed. For example, the volume might not be writable.

ERROR_DISK_STALE

The drive containing that disk has been deleted. This usually only happens with network drives.

Formatting a Disk

17.3

If a geode needs to format a disk, it can call the routine **DiskFormat()**. This routine can do either low-level or high-level ("soft") formats. The routine does not interact with the user interface; instead, it calls a callback routine, which can arrange any such interaction. **DiskFormat()** takes seven arguments:

- ◆ The number of the drive containing the disk to be formatted.
- ◆ The address of a null-terminated string containing the disk's new volume name.
- ◆ A member of the **MediaType** enumerated type (see page 618).
- ◆ Flags indicating how the format should be done. The following flags are available:

DFF_CALLBACK_PERCENT_DONE

A callback routine will be called periodically. The callback routine will be passed a single argument, namely the percentage of the format which has been done, expressed as an integer.

DFF_CALLBACK_CYL_HEAD

A callback routine will be called periodically. The callback routine will be passed a single argument, namely the cylinder head being formatted. If both **DFF_CALLBACK_PERCENT_DONE** and **DFF_CALLBACK_CYL_HEAD** are passed, results are undefined.

DFF_FORCE_ERASE

A "hard format" should be done; that is, the sectors should be rewritten and initialized to zeros. If this flag is not set, **DiskFormat()** will do a "soft format" if possible; it will check the sectors and write a blank file allocation table, but it will not necessarily erase the data from the disk.

- ◆ A pointer to an unsigned integer variable; the number of good sectors on the disk will be written here.
- ◆ A pointer to an unsigned integer variable; the number of bad sectors on the disk will be written here.
- ◆ The address of a Boolean callback routine, as described above. The routine should be passed either the current cylinder and head or the percentage formatted, depending on the flag passed to **DiskFormat()**. It should return *true* to abort the format, or *false* (i.e. zero) to continue with the format. If neither `DFF_CALLBACK_PERCENT_DONE` nor `DFF_CALLBACK_CYL_HEAD` is passed, the callback routine will never be called, so this argument may be a null pointer.

DiskFormat() returns a member of the **FormatError** enumerated type. If the format was successful, it will return the constant `FMT_DONE` (which is guaranteed to equal zero). Otherwise, it will return one of the following constants:

```
FMT_DRIVE_NOT_READY
FMT_ERROR_WRITING_BOOT
FMT_ERROR_WRITING_ROOT_DIR
FMT_ERROR_WRITING_FAT
FMT_ABORTED
FMT_SET_VOLUME_NAME_ERROR
FMT_CANNOT_FORMAT_FIXED_DISKS_IN_CUR_RELEASE
FMT_BAD_PARTITION_TABLE,
FMT_ERR_READING_PARTITION_TABLE,
FMT_ERR_NO_PARTITION_FOUND,
FMT_ERR_MULTIPLE_PRIMARY_PARTITIONS,
FMT_ERR_NO_EXTENDED_PARTITION_FOUND
FMT_ERR_CANNOT_ALLOC_SECTOR_BUFFER
FMT_ERR_DISK_IS_IN_USE
FMT_ERR_WRITE_PROTECTED
FMT_ERR_DRIVE_CANNOT_SUPPORT_GIVEN_FORMAT
FMT_ERR_INVALID_DRIVE_SPECIFIED
FMT_ERR_DRIVE_CANNOT_BE_FORMATTED
FMT_ERR_DISK_UNAVAILABLE
```

Copying Disks

GEOS provides a routine for copying disks. This routine, **DiskCopy()**, maintains a balance between the two goals of limiting memory usage and minimizing disk swapping. It will reformat the destination disk if necessary. The routine does a sector-for-sector copy; therefore, the destination disk must either be of exactly the same type as the source disk (i.e., same medium and size), or it must be reformatable to be the same size. For this reason, neither the source nor the destination may be a fixed disk.

17.3

DiskCopy() does not interact with the user directly, even though the user may have to swap disks. Instead, it calls a callback routine whenever interaction with the user may be necessary. The routine takes the following arguments:

- ◆ The drive number of the source drive.
- ◆ The drive number of the destination drive. This may be the same as the source drive.
- ◆ The address of a Boolean callback routine. The routine must take three arguments: a member of the **DiskCopyCallback** enumerated type (described below), a disk handle, and a word-sized parameter holding any other appropriate information. The routine should return non-zero to abort the copy; otherwise, it should return zero.

The callback routine is called under a variety of circumstances. When it is called, the first argument passed is a member of the **DiskCopyCallback** enumerated type, which specifies both why the callback routine was called and what the other arguments signify. **DiskCopyCallback** contains the following types:

CALLBACK_GET_SOURCE_DISK

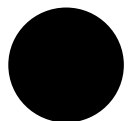
The callback routine should prompt the user to insert the source disk. The second argument is meaningless for this call. The third argument is the number identifying the drive.

CALLBACK_GET_DEST_DISK

The callback routine should prompt the user to insert the destination disk. The second argument is meaningless for this call. The third argument is the number identifying the drive.

CALLBACK_REPORT_NUM_SWAPS

The second argument is meaningless for this call. The third



argument is the number of disk swaps that will be necessary to copy the disk. The callback routine may wish to report this number to the user and ask for confirmation.

`CALLBACK_VERIFY_DEST_DESTRUCTION`

If the destination disk has already been formatted, the callback routine will be called with this parameter. The callback routine may wish to remind the user that the destination disk will be erased. The second argument is the handle of the destination disk; this is useful if, for example, you want to report the disk's name. The third argument is the destination drive's number. As in the other cases, the callback routine can abort the format by returning non-zero.

17.3

`CALLBACK_REPORT_FORMAT_PERCT`

If the destination disk needs to be formatted, **DiskCopy0** will periodically call the callback routine with this parameter. In this case, the second argument will be meaningless; the third parameter will be the percentage of the destination disk which has been formatted.

`CALLBACK_REPORT_COPY_PERCT`

While the copy is taking place, **DiskCopy0** will periodically call the callback routine with this parameter. In this case, the second parameter will be meaningless; the third parameter will be the percentage of the copy which has been completed.

If the copy was successful, **DiskCopy0** returns zero. Otherwise, it returns a member of the **DiskCopyError** enumerated type, which has the following members:

`ERR_DISKCOPY_INSUFFICIENT_MEM`

This is returned if the routine was unable to get adequate memory.

`ERR_CANT_COPY_FIXED_DISKS`

`ERR_CANT_READ_FROM_SOURCE`

`ERR_CANT_WRITE_TO_DEST`

`ERR_INCOMPATIBLE_FORMATS`

`ERR_OPERATION_CANCELLED`

This is returned if the callback routine ever returned a non-zero value, thus aborting the copy.

ERR_CANT_FORMAT_DEST

17.4 Directories and Paths

Information, whether code or data, is always stored in files. However, storage volumes are not simply collections of files. Rather, they are organized into directories. There are two main reasons for this organization.

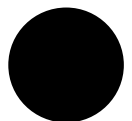
17.4

First, a large storage device can easily contain hundreds or thousands of files. Computer users need to be able to find a specific file quickly. If they were confronted with a list of all the files on a storage device, they would have a hard time finding the one they want. The directory structure solves this problem; users can navigate through the directory tree until they find the file they want.

The other reason for the directory structure is to expand the name-space for files. If every file on a volume were forced to have a unique name, users would find themselves severely restricted. Directories allow more flexibility; since a file is identified by both its name and its location, its name need only be unique within its directory.

A file can be uniquely specified by a *full path*. A full path specifies the volume containing the file as well as all the directories containing the file, starting with the root directory. The volume specification can usually be omitted, indicating that the file is on the same volume as the current working directory. Alternatively, the file can be specified with a *relative path*. A relative path specifies the file's position relative to the current working directory, instead of starting with the root.

In most implementations of DOS, there is a standard way of describing a path. These conventions are used in GEOS as well. A full path begins with a backslash, which represents the root directory. This may be followed by one or more directory specifications, separated by backslashes. The first directory listed would be a member of the root directory; each following directory would be a member of the preceding directory. A relative path is the same, except that it does not begin with a backslash. In either a full or a relative path, there are two special directory names. A single period (".") specifies the current directory; that is, the path "." indicates the current working directory,



and the path “\GEOWORKS\DOCUMENT” is the same as the paths “\GEOWORKS\DOCUMENT\.” and “\GEOWORKS\.\DOCUMENT”. A double period (“..”) indicates the parent of the current directory; thus, “\GEOWORKS\DOCUMENT\..” is equivalent to “\GEOWORKS”.

17.4

The GEOS file management system allows each thread to have a working directory. Whenever a thread needs to open a file, it can rely on its working directory instead of passing a full path. GEOS provides an added mechanism: it defines certain system *standard paths* which can be reached with simple system commands. These paths provide a way to standardize directory usage; an application might, for example, keep a certain data file in the standard PRIVDATA directory, leaving the user to decide where that PRIVDATA directory may be. This is covered in detail in the following section.

17.4.1 Standard Paths

The GEOS system is designed to run on a wide variety of architectures. For example, it can be stored on a hard disk, recorded in ROM chips, or resident on a network server and run by members of the network. This presents a difficulty. The system uses many files, storing both code (such as libraries) and data (such as fonts). It can't assume that they are in any specific place. For example, a user might be running GEOS from a network server, but she might have several fonts for personal use on her own hard disk. The system has to be able to look in all the right places. Applications have a similar dilemma. An application needs to be able to choose an appropriate place to create data files, and needs to be able to find them later.

The solution to this is to use *standard paths*. There are a number of standard directories GEOS uses. Each one of these has a constant associated with it. The constants are treated as special disk handles. For example, the routine **FileSetCurrentPath()** (described in section 17.4.2 on page 636) takes two arguments, a disk handle and a path string. If the disk handle passed is the constant SP_DOCUMENT and the string passed is “Memos”, the routine will look in the standard document path for a directory called “Memos” and will open it.

There are two advantages to standard paths. The first is that they give the system and the applications points of reference. An application does not need

to worry where to write its internal data files; it can simply install them in the standard PRIVDATA directory.

The second advantage is that the standard paths do not need to correspond to any one specific directory. A standard path can actually be several different DOS directories. For example, the GEOS kernel looks for font files in the standard path SP_FONT. The user may define this path to be several directories; it might comprise a local directory for the most commonly used fonts, a network directory for some decorative fonts, etc. The file system would automatically locate all of these fonts. Similarly, the SP_DOCUMENT directory might comprise a directory on a ROM disk for read-only files, as well as a directory on a hard disk where files are written. A standard path is considered “read-only” if and only if all of the directories which make up the standard path are read-only. When you create a file directly in a standard path (as opposed to in a subdirectory of a standard path), the file system will write it to the directory on the disk containing the **geos.ini** file.

17.4

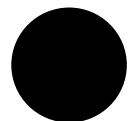
The standard paths have a default hierarchy. For example, the standard path SP_VIDEO_DRIVERS usually refers to a directory named VIDEO which belongs to the standard path SP_SYSTEM. However, the user is allowed to define each standard path however he or she wishes. For this reason, the programmer may not make any assumptions about where each standard path is located.

Below is a list of standard paths currently implemented. The paths are described as if they were single directories; remember, however, that each one of these may comprise several directories. Each path is accompanied by notes on how it is used and where it might commonly be found. The user can decide where each of these directories is, so make no assumptions about their locations. These paths will always be supported; future versions of GEOS may add new paths.

SP_TOP The top level directory. This is the directory which contains GEOS.INI. This is often C:\GEOWORKS.

SP_APPLICATION
All non-system applications are in this directory or a directory belonging to it. This is often a WORLD directory under SP_TOP.

SP_DOCUMENT
All document files should go here. This is commonly a DOCUMENT directory under SP_TOP.



SP_SYSTEM All libraries and drivers belong to this directory. All libraries must go in this directory, not in a subdirectory. Drivers are further grouped into subdirectories, one for each type of driver. This is commonly a SYSTEM directory under SP_TOP. Geodes should never need to directly access this directory, other than for installing drivers to it.

SP_PRIVATE_DATA
This contains data files which should not be accessed by users. By convention, each geode creates a subdirectory with its own data files. For example, the application HELLO.GEO would use a directory called HELLO in SP_PRIVATE_DATA. This is commonly a PRIVDATA directory under SP_TOP.

SP_DOS_ROOM
This directory contains the DOS-room launchers. It is commonly a DOSROOM directory under SP_TOP.

SP_STATE This directory contains all state files. It is commonly a STATE directory under PRIVDATA.

SP_USER_DATA
This directory holds data files that the user may add to, delete, upgrade, or otherwise change. Each data file type should have its own subdirectory. This is commonly a USERDATA directory under SP_TOP.

SP_FONT This directory contains all font data files, no matter what format they are. It is often USERDATA\FONT.

SP_SPOOL This directory contains all application spooler files. It is commonly USERDATA\SPOOL.

SP_SYS_APPLICATION
This directory contains all system applications. For example, it contains the Geoworks Pro "Welcome" application. These applications should not be launched by the user from the desktop. This directory is commonly SYSTEM\SYSAPPL.

SP_MOUSE_DRIVERS
This directory contains all mouse drivers. It is commonly SYSTEM\MOUSE.

SP_PRINTER_DRIVERS

This contains all printer drivers. It is commonly SYSTEM\PRINTER.

SP_FILE_SYSTEM_DRIVERS

This directory contains drivers for file systems. It has both DOS and network drivers. It is commonly SYSTEM\FS.

SP_VIDEO_DRIVERS

This directory contains video drivers. It is commonly SYSTEM\VIDEO.

17.4

SP_SWAP_DRIVERS

This directory has all of the swap drivers. It is commonly SYSTEM\SWAP.

SP_KEYBOARD_DRIVERS

This directory has all of the keyboard drivers. It is commonly SYSTEM\KEYBOARD.

SP_FONT_DRIVERS

This directory has all of the font drivers. It is commonly SYSTEM\FONT.

SP_IMPORT_EXPORT_DRIVERS

This directory has all of the import/export libraries. It is commonly SYSTEM\IMPEX.

SP_TASK_SWITCH_DRIVERS

This directory contains all task-switching drivers. It is commonly SYSTEM\TASK.

SP_HELP_FILES

This directory contains all help files. It is commonly USERDATA\HELP.

SP_TEMPLATE

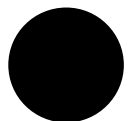
This directory contains document templates. It is commonly DOCUMENT\TEMPLATE.

SP_POWER_DRIVERS

This directory contains power-management drivers. It is commonly SYSTEM\POWER.

SP_HWR

This directory contains the handwriting recognition libraries. It is commonly SYSTEM\HWR.



SP_WASTE_BASKET

This directory contains files that have been dragged to the waste basket but have not yet been thrown out. All non-open files and directories in this path are periodically deleted. Temporary files should usually be created here; they will not be deleted until after they have been closed. This directory is commonly PRIVDATA\WASTE.

17.4

SP_BACKUP This directory contains backup files made by the document control. It is commonly \PRIVDATA\BACKUP.

17.4.2 Current Path and Directory Stack

`FileSetCurrentPath()`, `FileGetCurrentPath()`,
`FileConstructFullPath()`, `FileParseStandardPath()`,
`FileResolveStandardPath()`, `FilePushDir()`, `FilePopDir()`,
`FileResolveStandardPathFlags`

Every thread has a *current path*. When the thread opens a file, it can pass just the name of the file; the file system combines this name with the current path to find the file. The path is a combination of a disk handle and a directory sequence. To set the thread's current path, call the routine **FileSetCurrentPath()**, which takes two arguments: a disk handle and a pointer to a null-terminated string. The string should contain a sequence of directories specified in the normal DOS convention. To change to a standard path, pass the path constant as the disk handle and a null string (i.e. a pointer to a null byte). To change to a subdirectory of a standard path, pass the path constant as the disk handle and a pointer to a relative or absolute path specification; for example, to change to the HELLO directory in PRIVDATA, pass the disk handle constant SP_PRIVATE_DATA and a pointer to the string "HELLO". **FileSetCurrentPath()** returns the handle of the disk. If you change to a standard path, it returns the path constant; if you change to a directory within a standard path, it returns the constant of the closest standard path. In the event of error, it returns a null handle and sets the thread's error value. The error most commonly returned is ERROR_PATH_NOT_FOUND, indicating that the specified directory could not be found or does not exist.

To find out the current path, call the routine **FileGetCurrentPath()**. This routine takes two arguments: the address of a character buffer and the size

of the buffer. It returns the handle of the current path's disk and writes the path (without drive specifier) to the buffer, truncating the path if the buffer is too small. If the directory is a standard path or a subdirectory of one, **FileGetCurrentPath()** will return the disk-handle constant for that path and will write an absolute path to the buffer. If you want a full path, use **FileConstructFullPath()** below.

To translate a standard path into a full path, call **FileConstructFullPath()**, which takes five arguments:

17.4

- ◆ A Boolean value indicating whether the drive name should be prepended to the path string returned.
- ◆ The handle of a disk. A null argument indicates that the path string passed is relative to the current working directory. The handle may be a standard path constant, indicating that the path string passed is relative to that standard path.
- ◆ A pointer to a string containing a path relative to the location indicated by the previous argument. This may be an empty string.
- ◆ A pointer to a pointer to a character buffer. The path will be written to this buffer. The routine will update this to be a pointer to a pointer to the null terminator.
- ◆ The length of that buffer.

The routine writes the full path to the buffer and returns the disk handle. If it is unable to construct a full path, it returns a null handle.

To find the standard path to a given location, call the routine **FileParseStandardPath()**. This routine is passed two arguments:

- ◆ The handle of the disk. Passing a null disk handle indicates that the path string contains a drive specifier.
- ◆ A pointer to a pointer to a path string. This path should begin at the root of the disk specified.

FileParseStandardPath() returns the standard path constant. It also updates the pointer to point to the remaining portion of the path. For example, if you pass a pointer to a pointer to the string “\GEOWORKS\DOCUMENT\MEMOS\APRIL”, the pointer would be updated to point to the “\MEMOS\APRIL” portion, and the handle SP_DOCUMENT would be returned. If the path passed does not belong to a standard path, the

constant `SP_NOT_STANDARD_PATH` will be returned, and the pointers will not be changed.

Because each standard path is made up of one or more directories (possibly on different devices), it can be hard to know just where a file is. For that reason, GEOS provides **FileResolveStandardPath()**. This routine is passed a relative path to a file; it then constructs the full path of the file, starting from the root of the disk (*not* from a standard path); it also returns the handle of the actual disk containing the file.

FileResolveStandardPath() is passed four arguments:

- ◆ A pointer to a pointer to a character buffer. The full path will be written to that buffer, and the pointer will be updated to point to the null terminator.
- ◆ The length of the buffer, in bytes.
- ◆ A pointer to a relative path string. This path is relative to the current working directory, which is usually a standard path.
- ◆ A set of **FileResolveStandardPathFlags**.

FileResolveStandardPath() writes the full, absolute path to the buffer specified. It also returns the handle of the disk containing that file. If it cannot find the file specified, it returns a null handle. There are two **FileResolveStandardPathFlags** available:

FRSPF_ADD_DRIVE_NAME

The path string written to the buffer should begin with the drive name (e.g., "C:\GEOWORKS\DOCUMENT\MEMOS").

FRSPF_RETURN_FIRST_DIR

FileResolveStandardPath() should not check whether the passed path actually exists; instead, it should assume that the path exists in the first directory comprising the standard path, and return accordingly.

In addition to having a current path, every thread has a *directory stack*. The stack is used for switching paths quickly. You can at any time push the current path onto the stack by calling **FilePushDir()**. This routine pushes the directory on the directory stack and returns nothing. You can change the current path to the one on top of the directory stack by calling **FilePopDir()**. This pops the top directory off the stack and makes it the current path. (If the directory stack is empty, the result is undefined.) These routines are very

useful when you write subroutines which may need to change the current working directory; they can push the old directory at the start of the routine and pop it at the end.

Files are often specified “by their paths.” This simply means specifying them with a string containing the directory path and ending with the file name. This path may be either relative or absolute.

17.4.3 Creating and Deleting Directories

17.4

`FileCreateDir()`, `FileDeleteDir()`

You can create directories with **FileCreateDir()**. The routine takes a single argument, namely the address of a character string. If the string is simply a directory name, it will attempt to create that directory at the current location. If the string is a relative path, it will create the directory at the end of the path, if possible. For example, passing the string “Memos\September” will cause it to check if there is a directory called “Memos” at the current location. If there is, it will create the directory “September” inside the directory “Memos”. If the string is an absolute path (i.e. there is a backslash before the first directory name), it will behave the same way, but it will start with the root directory. The routine returns zero if it was successful. If it was unsuccessful, it will return one of the following members of the **FileError** type:

ERROR_INVALID_NAME

The name passed was inappropriate for directories on that device.

ERROR_PATH_TOO_LONG

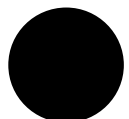
The path string was longer than is permitted by the file system for that device.

ERROR_FILE_EXISTS

A file or directory with the specified name already exists at the specified location.

ERROR_PATH_NOT_FOUND

The path string was in some way invalid; for example, it might have instructed **FileCreateDir()** to create the directory inside of a directory which does not exist.



`ERROR_ACCESS_DENIED`

The thread is not able to create directories in the specified location.

`ERROR_DISK_STALE`

The drive that disk was on has been removed.

`ERROR_DISK_UNAVAILABLE`

The validation of the disk in that drive was aborted by the user.

17.4

Note that the directory name can be any acceptable GEOS file name; that is, it may be up to 32 characters long and can contain any characters other than backslashes, colons, asterisks, or question marks. For further information about GEOS file names, see section 17.5.1 on page 641.

You can delete directories with **FileDeleteDir()**. This routine takes a single argument, namely the address of a character string. This string can specify a relative or absolute path, as with **FileCreateDir()**. It attempts to delete the directory specified. (Note that you are not allowed to delete your current directory or a non-empty directory.) If it successfully removes the directory, it returns zero. Otherwise, it returns one of the following members of the **FileError** type:

`ERROR_PATH_NOT_FOUND`

The directory specified could not be found or does not exist.

`ERROR_IS_CURRENT_DIRECTORY`

This directory is some thread's current directory, or else it is on some thread's directory stack. Note that you cannot delete your own working directory.

`ERROR_ACCESS_DENIED`

The calling thread does not have permission to delete the directory. This is also returned if the directory was on a read-only device.

`ERROR_DIRECTORY_NOT_EMPTY`

The directory specified is not empty. Delete all the files before you attempt to delete the directory.

`ERROR_DISK_STALE`

The drive that disk was on has been removed.

`ERROR_DISK_UNAVAILABLE`

The validation of the disk in that drive was aborted by the user.

17.5 Files

When data is not actually in a computer's memory, it needs to be grouped together in a manageable form. Most storage devices group data together into *files*. A file is a collection of information. It may be a program, either in machine language or in a higher-level language (as with batch files); or it may be data.

When an application is accessing a data file, it is said to have that file *open*. It may be open for read and write access, just for reading, or just for writing. If a file is open, it can not be moved or deleted (although it can be copied).

17.5

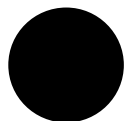
You will probably not have to read all of this section. "Bytewise File Operations" (section 17.5.6) is important only if you will be using DOS files or GEOS byte files. Most applications will work with GEOS *Virtual Memory* (VM) files.

17.5.1 DOS Files and GEOS Files

Most file systems have a simple convention of what a file is. Applications treat files as if they were a sequence of bytes. They can read the bytes in order from the file, copying a sequence of bytes into memory. They have a "position" in the file, indicating the next byte to be read. An application can copy data from memory to a file, replacing the data there.

GEOS allows geodes to access any type of DOS file using the normal DOS conventions. However, it also provides its own file format, the GEOS file. GEOS files are stored on the disk as normal DOS files. This means that they can be copied from one disk to another by any normal DOS procedure; they can be uploaded or downloaded, compressed, archived, or transferred to different devices exactly as if they were ordinary DOS files. Nevertheless, when a system is running GEOS, the files have added functionality. GEOS data files have special *extended attributes*, which keep track of such things as the file's tokens, protocol numbers, etc. For more information about extended attributes, see section 17.5.3 on page 643.

Most GEOS data files are *Virtual Memory* files. VM files can automatically back up data; they allow their users to insert data at any point in the file, and when the file's user needs access to data the VM manager automatically



allocates memory and swaps the data into it. These files are created and accessed using special VM routines; for more information, see “Virtual Memory,” Chapter 18. Note that you can have a GEOS file hold raw data instead of having VM capability. Such a file is known as a “byte” file since it is treated as a sequence of bytes with no structure except what is specifically added by the file’s creator. All of the routines for working with DOS files can be used with GEOS byte files.

17.5

One basic difference between GEOS files and DOS files is in naming conventions. Each file system and disk-operating system has its own convention for how files may be named. By far, the most common convention is the one used by MS-DOS: each file is identified by a name of up to eight characters, followed by an optional extension of up to three characters. There are further restrictions on which characters may be part of a name; for example, none of the characters may be a period.

GEOS provides more versatility. Each file has two different names. It has a *native* name; this is the name used by the device holding the file and must meet all of its restrictions. For example, if the file is kept on a usual DOS-compatible disk, the native name would be in the FILENAME.EXT format. The other name is the *virtual* file name. This may contain a number of characters equal to FILE_LONGNAME_LENGTH and may contain any characters other than backslashes (“\”), colons (“:”), asterisks (“*”), and question marks (“?”). Any time a file’s name is needed, either the virtual or the native file name may be used. When a file is created, GEOS automatically maps its virtual name into an appropriate native name.

Special GEOS information, such as the file’s virtual name and extended attributes, is stored in the body of the file itself. Thus, while applications should not access this information directly, they can still be assured that it will be preserved when the file is copied by any normal DOS techniques. This information is stored in a special header which is transparent to geodes. If you use one of the bitwise file operations, you will not be able to affect the header.

17.5.2 Files and File Handles

`FileDuplicateHandle()`

In order to read or change a file, you must *open* it with an appropriate routine. When you open a file, the file system sees to it that the file will not be erased or moved until you close it. When a geode exits, all files it has open are automatically closed. When GEOS exits, all files are closed. If an application restarts from a saved state, it will have to reopen all files.

17.5

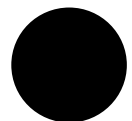
When you open a file, the GEOS file system returns a *file handle*. This is a global handle associated with that file. Whenever you need to perform an action on a file you have opened, you specify the file by passing its handle. Note that there may be several handles which all refer to the same file. If more than one thread will need to use the same handle, they may synchronize their access by using **HandleP0** and **HandleV0**; for details, see section 15.3.6 of chapter 15. The Virtual Memory routines include data-access synchronization routines for use with VM files; for details, see “Virtual Memory,” Chapter 18.

It is sometimes useful to have two different handles which indicate the same file. GEOS provides the routine **FileDuplicateHandle()**. This routine takes one argument, namely the handle of an open file. It creates and returns another handle which indicates the same file. You will have to close both handles to close the file. **FileDuplicateHandle()** works on any type of file handle; that is, it can be used on the handles of DOS files, GEOS byte files, or VM files. Note that the duplicate handle will have the same read/write position as the original.

17.5.3 GEOS Extended Attributes

`FileGetHandleExtAttributes()`, `FileGetPathExtAttributes()`,
`FileSetHandleExtAttributes()`, `FileSetPathExtAttributes()`,
`FileExtAttrDesc`, `FileDateAndTime`, `FileAttrs`

All GEOS files, whether they contain code or data, have special *extended attributes*. Geodes cannot look at these directly; instead, they make calls to the file system when they want to examine or change the attributes. There are many different extended attributes; however, they are all accessed and



changed in a uniform way. Some of the extended attributes are also supported for non-GEOS files. The following extended attributes are currently available:

```
FEA_MODIFICATION
FEA_FILE_ATTR
FEA_SIZE
FEA_FILE_TYPE
FEA_FLAGS
FEA_RELEASE
FEA_PROTOCOL
FEA_TOKEN
FEA_CREATOR
FEA_USER_NOTES
FEA_NOTICE
FEA_CREATION
FEA_PASSWORD
FEA_CUSTOM
FEA_NAME
FEA_GEODE_ATTR
FEA_PATH_INFO
FEA_FILE_ID
FEA_DESKTOP_INFO
FEA_DRIVE_STATUS
FEA_DISK
FEA_DOS_NAME
FEA_OWNER
FEA_RIGHTS
```

There are also two special constants, `FEA_MULTIPLE` and `FEA_END_OF_LIST`. These are also described below.

There are two different routines to read a file's extended attributes:

FileGetHandleExtAttributes() and **FileGetPathExtAttributes()**.

These routines are the same except in the way the file is specified: in one, the handle of an open file is passed, whereas in the other, the address of a path string is passed.

FileGetHandleExtAttributes() takes four arguments. The first is the handle of the file whose attributes are desired; this may be a VM file handle or a byte-file handle. The second is a constant specifying the attribute desired. All extended attributes which are currently supported are listed above; more may be added later. The third is a pointer to a buffer; the attribute's value will be written into that buffer. The fourth argument is the size of the buffer (in bytes). Before it returns,

FileGetHandleExtAttributes() will write the value of the attribute into the buffer. If successful, it will return zero; otherwise, it will return one of the following error codes: 17.5

ERROR_ATTR_NOT_SUPPORTED

The file system does not recognize the attribute constant passed.

ERROR_ATTR_SIZE_MISMATCH

The buffer passed was too small for the attribute requested.

ERROR_ATTR_NOT_FOUND

The file does not have a value set for that attribute. This is returned if you try to get certain extended attributes of a non-GEOS file.

ERROR_ACCESS_DENIED

You do not have read-access to the file.

FileGetHandleExtAttrs() can also fetch several attributes at once. For details on this, see the section on **FEA_MULTIPLE** (page 646).

You can get a file's extended attributes without having it open by calling **FileGetPathExtAttributes()**. This routine takes a pointer to a null-terminated path string instead of a file handle. This makes it suitable for examining the attributes of an executable file or directory. Note that the file system will still have to open the file in order to get the attributes. If any geode (including the caller) has the file open with "deny-read" exclusive, the call will fail with error condition **ERROR_ACCESS_DENIED**. If it could not find the file specified, it will return **ERROR_FILE_NOT_FOUND**.

To change one of a file's extended attributes, make a call either to **FileSetHandleExtAttributes()** or to **FileSetPathExtAttributes()**. These routines take the same arguments as the **FileGet...()** routines above; however, they copy the data from the buffer into the attribute, instead of vice

versa. These routines return zero if the operation was successful. Otherwise, they return one of the following error codes:

ERROR_ATTR_NOT_SUPPORTED

The file system does not recognize the attribute constant passed. This is returned if you try to set an extended attribute for a non-GEOS file.

ERROR_ATTR_SIZE_MISMATCH

The buffer passed was the wrong size for the attribute specified.

ERROR_ACCESS_DENIED

FileSetHandleExtAttributes() returns this if the caller does not have write-access to the file. **FileSetPathExtAttributes()** returns this if any geode (including the caller) has the file open with “deny-write” exclusive access, or if the file is not writable.

ERROR_CANNOT_BE_SET

The extended attribute cannot be changed. Such attributes as **FEA_SIZE** and **FEA_NAME** cannot be changed with the **FileSet...()** routines.

FEA_MULTIPLE

By passing this extended attribute, you can get or set several extended attributes at once. This is also the only way to get, set, or create a custom attribute. If you pass this, the other arguments are slightly different. The first argument is still the file specifier (handle or path), and the second argument is **FEA_MULTIPLE**. However, the third argument is the base address of an array of **FileExtAttrDesc** structures, and the fourth argument is the number of these structures in the array. The array has one element for each attribute you wish to get or set. Each **FileExtAttrDesc** structure has the following fields:

FEAD_attr This is the numerical constant for the attribute to be read or set. If a custom attribute is being set, this should be **FEA_CUSTOM**.

FEAD_value If the attribute is being set, this is the address of the new value. If the attribute is being read, this is the address of the buffer into which to copy the value.

FEAD_size This is the size of the buffer or value pointed to by *FEAD_value*.

FEAD_name If *FEAD_attr* is set to `FEA_CUSTOM`, this is the address of a null-terminated string containing the custom attribute's name. If *FEAD_attr* is set to anything else, this field is ignored.

FEA_CUSTOM

In addition to the system-defined extended attributes, any GEOS file may have any number of custom attributes. Each custom attribute is named by a null-terminated ASCII string. To create a custom attribute, call one of the **FileSet...()** routines, specifying the new attribute with a **FileExtAttrDesc** structure (as described immediately above). If you try to read a custom attribute which has not been defined for that file, the routine will fail with error condition `ERROR_ATTR_NOT_FOUND`.

17.5

Note that not all file systems support the use of custom extended attributes; therefore, you should write your applications so they can perform correctly without using them.

FEA_MODIFICATION

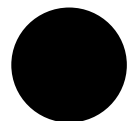
Every file has a “last modified” time. This is automatically updated whenever the file is written to. To find the modification time of a file, get the extended attribute `FEA_MODIFICATION`. The modification time is returned as a 32-bit **FileDateAndTime** value. The value has the following fields, each of which is small enough to fit in a signed-byte variable:

FDAT_YEAR This field records the year, counting from a base year of 1980. (The constant `FDAT_BASE_YEAR` is defined as 1980.) This field is at an offset of `FDAT_YEAR_OFFSET` bits from the low end of the value.

FDAT_MONTH This field records the month as an integer, with January being one. It is located at an offset of `FDAT_MONTH_OFFSET`.

FDAT_DAY This field records the day of the month. It is located at an offset of `FDAT_DAY_OFFSET`.

FDAT_HOUR This field records the hour on a 24-hour clock, with zero being the hour after midnight. It is located at an offset of `FDAT_HOUR_OFFSET`.



FDAT_MINUTE

This field records the minute. It is located at an offset of `FDAT_MINUTE_OFFSET`.

FDAT_2SECOND

This field records the second, divided by two; that is, a field value of 15 indicates the 30th second. (It is represented this way to let the second fit into 5 bits, thus letting the entire value fit into 32 bits.) It is located at an offset of `FDAT_2SECOND_OFFSET`.

17.5

The macros **FDATExtractYear()**, **...Month()**, **...Day()**, **...Hour()**, **...Minute()**, and **...2Second()** all extract the specified field from a **FileDateAndTime** value. The macro **FDATExtractSecond()** extracts the `FDAT_2SECOND` field and doubles it before returning it. The **FDATExtractYearAD()** extracts the year field and adds the base year, thus producing a word-sized year value.

FEA_FILE_ATTR

There are certain attributes which all files have. These attributes specify such things as whether the file is hidden, whether it is read-only, and several other things. To get these attributes, call an extended attribute routine with argument `FEA_FILE_ATTRIBUTES`. The attributes are passed or returned in a **FileAttrs** record. This record has the following fields:

FA_ARCHIVE

This flag is set if the file requires backup. Backup programs typically clear this flag.

FA_SUBDIR This flag is set if the “file” is actually a directory. Geodes may not change this flag.

FA_VOLUME

This flag is set if the “file” is actually the volume label. This flag will be *off* for all files a geode will ever see. Geodes may not change this flag.

FA_SYSTEM This flag is set if the file is a system file. Geodes should not change this flag.

FA_HIDDEN This flag is set if the file is hidden from normal directory searches. For example, a `GenFileSelector`, by default, does not list files that have this flag set.

FA_RDONLY This flag is set if the file is read-only.

Many file systems (including DOS) require that files be closed when you set these attributes. For that reason, you cannot change these attributes with **FileSetHandleExtAttributes()**. You must use either **FileSetPathExtAttributes()** or **FileSetAttributes()** (described below in section 17.5.6.3). If you try to set this field with **FileSetHandleExtAttributes()**, you will be returned **ERROR_ATTR_CANNOT_BE_SET**.

17.5

FEA_SIZE

This attribute is simply the size of the file in bytes. It is dword-sized (allowing for files as large as 2^{32} bytes, or 4096 megabytes). The attribute can be read, but not directly changed.

FEA_FILE_TYPE

This attribute is a member of the **GeosFileType** enumerated type and should not be altered. The type has the following values:

GFT_NOT_GEOS_FILE

The file is not a GEOS file. This constant is guaranteed to be equal to zero.

GFT_EXECUTABLE

The file is executable; in other words, it is some kind of geode.

GFT_VM The file is a VM file.

GFT_DATA The file is a GEOS byte file (see below).

GFT_DIRECTORY

The file is a GEOS directory.

GFT_LINK The file is a symbolic link (not yet implemented).

FEA_FLAGS

This attribute is a word-sized flag field, named **GeosFileHeaderFlags**. The following flags are implemented:

GFHF_TEMPLATE

The file is a document template.

GFHF_SHARED_SINGLE

The file can be opened for shared-single access.

GFHF_SHARED_MULTIPLE

The file can be opened for shared-multiple access.

Shared-single and shared-multiple access are described in the VM chapter. For more details, see “Virtual Memory,” Chapter 18.

17.5

FEA_RELEASE

This attribute is a **ReleaseNumber** structure. Generally, only geodes have release numbers. The structure has the following fields:

RN_major The file's major release number. An increase in the major release number generally indicates a change which is not downward-compatible.

RN_minor The file's minor release number. An increase in the minor release number generally indicates that the new version is compatible with previous versions.

RN_change A field for use by individual manufacturers.

RN_engineering
A field for use by individual manufacturers.

FEA_PROTOCOL

This attribute contains the file's protocol numbers. A **ProtocolNumber** structure is returned. For a discussion of file protocols, see section 13.4.1 of “GenDocument,” Chapter 13 of the Object Reference Book.

FEA_TOKEN

This attribute is the file's token. It consists of a **GeodeToken** structure. For more information about tokens, see “Applications and Geodes,” Chapter 6.

FEA_CREATOR

This attribute is the token of the document's creator. It consists of a **GeodeToken** structure. For more information about tokens, see “Applications and Geodes,” Chapter 6.

FEA_USER_NOTES

This attribute is a null-terminated string. It is displayed in the file's "Info" box. Users can edit this string with GeoManager.

FEA_NOTICE

This attribute contains the file's copyright notice.

FEA_CREATION

17.5

This attribute is a **FileDateAndTime** structure. It contains the time when the file was created.

FEA_PASSWORD

This attribute contains the file's encrypted password, if any.

FEA_NAME

This attribute contains the file's virtual name. It is a null-terminated character string.

FEA_GEODE_ATTR

This attribute contains information about the geode. If the file is not a geode, this field's value will be zero. If it is a geode, it will contain a record of **GeodeAttrs**. This record has the following fields:

GA_PROCESS This geode has a process thread.

GA_LIBRARY This geode is a library.

GA_DRIVER This geode is a driver.

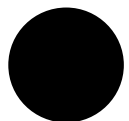
GA_KEEP_FILE_OPEN

The GEOS kernel keeps this file open while GEOS is running.

GA_SYSTEM This file is part of the kernel.

GA_MULTI_LAUNCHABLE

This geode can be loaded more than once simultaneously.



GA_DRIVER_INITIALIZED

The geode is a driver which has been opened, loaded and initialized.

GA_LIBRARY_INITIALIZED

The geode is a library which has been opened and loaded, and whose entry point has been called.

GA_GEODE_INITIALIZED

The geode has been opened and completely initialized.

GA_USES_COPROC

The geode uses a math coprocessor, if one is available.

GA_REQUIRES_COPROC

The geode can only run if a math coprocessor or emulator is present.

GA_HAS_GENERAL_CONSUMER_MODE

The geode is an application which can be run in GCM mode.

GA_HAS_ENTRY_POINTS_IN_C

This geode is a library or driver which can be called from C code.

FEA_PATH_INFO

This field contains information about the file's path. It is for internal use by the kernel.

FEA_FILE_ID

This field is for internal use by the kernel.

FEA_DESKTOP_INFO

This field is for use by the desktop manager.

FEA_DRIVE_STATUS

This attribute contains the **DriveExtendedStatus** word for the drive containing the file. The **DriveExtendedStatus** value is described in section 17.3.1 on page 616.

17.5.4 File Utilities

```
FileDelete(), FileRename(), FileCopy(),  
FileMove(), FileGetDiskHandle()
```

Most of the time, such actions as copying, deleting, and renaming files are handled by desktop management programs like GeoManager. However, other geodes may need to perform these actions themselves. For example, if you use a temporary file, you may wish to delete it when you're done. The GEOS file system provides routines for these situations. One file utility, **FileEnum()**, is elaborate enough to be treated in its own section; for details, see section 17.5.5 on page 655.

17.5

To delete a file, call **FileDelete()**. This routine takes one argument, namely the address of a path string. If it can delete the file, it returns zero; otherwise, it returns an error code. Common error conditions include the following:

ERROR_FILE_NOT_FOUND

No such file exists in the specified directory.

ERROR_PATH_NOT_FOUND

An invalid path string was passed.

ERROR_ACCESS_DENIED

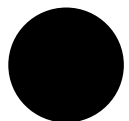
You do not have permission to delete that file, or it exists on a read-only volume.

ERROR_FILE_IN_USE

Some geode has that file open.

To change a file's name, call **FileRename()**. This routine takes two arguments: a pointer to the path string specifying the file, and a pointer to a string specifying the new name for the file. If successful, **FileRename()** returns zero; otherwise, it returns one of the above error codes.

To make a copy of a file, call **FileCopy()**. This routine takes four arguments: the handles of the source and destination disks (which may be the same), and the addresses of source and destination path strings. Passing a disk handle of zero indicates the current path's disk. Each string specifies a path relative to the location specified by the corresponding disk handle. If the handle is a disk handle, the path is relative to that disk's root. If the disk handle is a standard path constant, the path string is relative to that standard path. If the disk handle is null, the path is relative to the current working directory.



FileCopy() will make a copy of the file in the specified location with the specified name. If a file with that name and location already exists, it will be overwritten. **FileCopy()** returns zero if successful. Otherwise it returns one of the following error codes:

ERROR_FILE_NOT_FOUND

No such source file exists in the specified directory.

ERROR_PATH_NOT_FOUND

An invalid source or destination path string was passed.

ERROR_ACCESS_DENIED

You do not have permission to delete the existing copy of the destination file, or the destination disk or directory is not writable.

ERROR_FILE_IN_USE

Some geode has the existing destination file open.

ERROR_SHORT_READ_WRITE

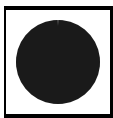
There was not enough room on the destination disk. The destination disk will be left unchanged (i.e. a partial copy of the file will not be made).

To move a file from one directory to another, either on the same disk or on different disks, call **FileMove()**. This routine takes four arguments: the handles of the source and destination disks (which may be the same), and pointers to source and destination path strings. Passing a null disk handle indicates the current working directory. Each string specifies a path relative to the location specified by the corresponding disk handle. If the handle is a disk handle, the path is relative to that disk's root. If the disk handle is a standard path constant, the path string is relative to that standard path. If the disk handle is null, the path is relative to the current working directory. If the copy is successful, **FileMove()** will return zero; otherwise, it will return one of the above error codes.

If you want to find out the handle of the disk containing an open file, call **FileGetDiskHandle()**. This routine is passed the file handle; it returns the disk handle. This is useful if the geode has to prepare for a shutdown; it can get the disk handle with **FileGetDiskHandle()**, then save that handle with **DiskSave()** (see "Saving and Restoring a Disk Handle" on page 624). With this information (and the file name), the geode will be able to reopen the file when it restarts.

17.5.5 FileEnum()

FileEnum(), FileEnumLocateAttr(), FileEnumWildcard(),
FileEnumAttrs, FileEnumSearchFlags,
FileEnumStandardReturnType, FEDosInfo, FENAMEAndAttr,
FileEnumCallbackData



Advanced Topic

Very few applications
will need to use
FileEnum() directly.

You may sometimes need to perform an action on every file that matches certain criteria. For these situations, the file system provides the routine **FileEnum()**. **FileEnum()** can be called in two ways. The usual way is to have **FileEnum()** provide certain information about every file in a directory (or every file of a given type); when called, **FileEnum()** will allocate a buffer and fill it with data structures, one for each matching file. Less commonly, you can have **FileEnum()** call a callback routine for every file which matches certain criteria; this callback routine can take a broader range of actions.

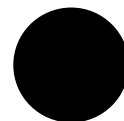
17.5

You should not often need to use the **FileEnum()** routine. The most common time that applications would need this functionality is when they present a list of the files in a directory to the user. The GenFileSelector object provides this functionality, making all necessary calls to **FileEnum()** transparently to both the user and the geode. Note that the document control objects automatically create and manipulate a file selector when appropriate. For further information, see “GenFile Selector,” Chapter 14 of the Object Reference Book and “GenDocument,” Chapter 13 of the Object Reference Book.

FileEnum() takes the following arguments:

- ◆ A pointer to a **FileEnumParams** structure (see below). The data in this structure will tell **FileEnum()** what to do.
- ◆ A pointer to a MemHandle variable. **FileEnum()** will allocate a memory block to hold information about the files, and will write the block’s handle to this location.
- ◆ A pointer to a word variable. If **FileEnum()** was unable to fit information about all the files into the block passed, it will write the number of files not handled into this variable.

FileEnum() returns the number of files which were returned in the buffer.



The **FileEnumParams** structure specifies how **FileEnum()** should perform. The structure is defined as follows. A summary of each field's role follows; for full details, see the reference entry for **FileEnum()**.

```
typedef struct _FileEnumParams {
    FileEnumSearchFlags      FEP_searchFlags;
    FileExtAttrDesc *        FEP_returnAttrs;
    word                     FEP_returnSize;
    FileExtAttrDesc *        FEP_matchAttrs;
    word                     FEP_bufSize;
    word                     FEP_skipCount;
    word _pascal (*FEP_callback)
        (struct _FileEnumParams *    params,
         FileEnumCallbackData *      fecd,
         word                        frame);
    FileExtAttrDesc *        FEP_callbackAttrs;
    dword                   FEP_cbData1;
    dword                   FEP_cbData2;
    word                    FEP_headerSize;
} FileEnumParams;
```

FEP_searchFlags

This is a byte-length flag field. The flags are of type **FileEnumSearchFlags** (described below). These flags specify which files at the current location will be examined by **FileEnum()**. They also specify such things as whether a callback routine should be used.

FEP_returnAttrs

This field specifies what information is wanted about the files. It is a pointer to an array of **FileExtAttrDesc** structures (see page 646). The attributes will be written to the return buffer; each file will have an entry, containing all the attributes requested for that file. You can also request certain return values by setting *FEP_returnAttrs* to equal a member of the **FileEnumStandardReturnType** (again, by casting the **FileEnumStandardReturnType** value to type **void ***). The **FileEnumStandardReturnType** enumerated type is described later in this section.

FEP_returnSize

This is the size of each entry in the returned buffer. If a standard return type or an array of **FileExtAttrDesc** structures was passed, each entry in the returned buffer will contain all the extended attribute information requested for that file.

FEP_matchAttrs

This field specifies which files should be enumerated by **FileEnum()**. It is a pointer to an array of **FileExtAttrDesc** structures (see page 646). These structures specify values for certain extended attributes. Only those files whose extended attributes match these will be enumerated. If you do not want to filter out any files in the working directory, or if you will use the callback routine to filter the files, pass a null pointer in this field.

17.5

FEP_bufsize

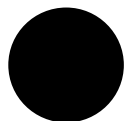
This field specifies the maximum number of entries to be returned in the buffer. If you do not want to set a limit, pass the constant **FEP_BUFSIZE_UNLIMITED**. The buffer will be grown as necessary.

FEP_skipCount

This field contains the number of matching files to be ignored before the first one is processed. It is often used in conjunction with *FEP_bufSize* to examine many files a few at a time. Each time **FileEnum()** is called, you can enumerate a certain number of files; by adjusting the skip count each time, you can start each enumeration where the previous one ended. In this way you could walk through all the matching files in the directory. Note that if the **FileEnumSearchFlags** bit **FESF_REAL_SKIP** is set (in *FEP_searchFlags*), the first files in the directory will be skipped *before* they are tested to see if they match. This is faster, since the match condition won't have to be checked for the first files in the directory.

FEP_callback

This field holds a pointer to a Boolean callback routine. The callback routine can check to see if the file matches some other arbitrary criteria. The callback routine is called for any files which match all the above criteria. It can then decide whether to enumerate the file however it wishes. The callback routine should be declared **_pascal**. If the file should be accepted by



FileEnum(), the callback should return *true*; otherwise it should return *false*. You can also instruct **FileEnum()** to use one of the standard callback routines by passing a member of the **FileEnumStandardCallback** enumerated type. In this case, *FEP_callbackAttrs* is ignored; **FileEnum()** will automatically pass the appropriate information to the callback routine. (Note that if the *FESF_CALLBACK* bit of the *FEP_searchFlags* field is not set, the *FEP_callback* field is ignored.) The callback routine may not call any routines which are in movable memory at the time **FileEnum()** is called, except for routines which are in the same resource as the callback routine.

FEP_callbackAttrs

This field specifies what additional attributes should be read for the callback routine. This field is a pointer to an array of **FileExtAttrDesc** structures (see page 646). The array will be filled in with the appropriate information for each file before the callback routine is called. Note that if the *FESF_CALLBACK* bit of the *FEP_searchFlags* is not set, the *FEP_callbackAttrs* is ignored. If you do not need any attributes passed to the callback routine, set this field to be a null pointer.

FEP_cbData1, *FEP_cbData2*

These are dword-length fields. Their contents are ignored by **FileEnum()**; they are used to pass information to the callback routine. If you do not call a standard callback routine, you may use these fields any way you wish.

FEP_headerSize

If the flag *FESF_LEAVE_HEADER* is set, **FileEnum()** will leave an empty header space at the beginning of the return buffer. The size of the header is specified by this field. If the flag *FESF_LEAVE_HEADER* is clear, this field is ignored.

The first field of the **FileEnumParams** structure, *FEP_searchFlags*, is a word-length record containing **FileEnumSearchFlags**. The following flags are available:

FESF_DIRS Directories should be examined by **FileEnum()**.

FESF_NON_GEOS

Non-GEOS files should be examined by **FileEnum()**.

FESF_GEOS_EXECS

GEOS executable files should be examined by **FileEnum()**.

FESF_GEOS_NON_EXECS

GEOS non-executable files (e.g., VM files) should be examined by **FileEnum()**.

FESF_REAL_SKIP

If a skip count of n is specified, the first n files will be skipped regardless of whether they matched the attributes passed. In this case, **FileEnum()** will return the number of files passed through in order to get enough files to fill the buffer; the return value can thus be the real-skip count for the next pass.

17.5

FESF_CALLBACK

FileEnum() should call a callback routine to determine whether a file should be accepted.

FESF_LOCK_CB_DATA

This flag indicates that the **FileEnumParams** fields *FEP_callback1* and *FEP_callback2* are far pointers to movable memory that must be locked before **FileEnum()** is called.

FESF_LEAVE_HEADER

If set, **FileEnum()** should leave an empty header space at the start of the return buffer. The size of this buffer is specified by the *FEP_headerSize* field.

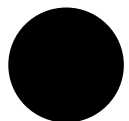
You can specify precisely which data you want about each file and in which arrangement it should be returned. However, for most purposes, you can get enough information by requesting one of the standard return types. If you pass a member of **FileEnumStandardReturnType** in *FEP_returnAttrs*, **FileEnum()** will write an array of structures to the return buffer; each file will have one such structure. The structures are shown below in Code Display 17-2 on page ● 660. **FileEnumStandardReturnType** has the following values:

FESRT_COUNT_ONLY

FileEnum() will not allocate any memory or return data about files; instead, it will simply return the number of files which match the specified criteria.

FESRT_DOS_INFO

FileEnum() will return an array of **FEDosInfo** structures.



These structures contain basic information about the file: its virtual name, size, modification date, DOS attributes, and path information (as a **DirPathInfo** record).

FESRT_NAME

FileEnum() will return an array of **FileLongName** strings, each one of which is FILE_LONGNAME_BUFFER_SIZE characters long; every one of these will contain a file's virtual name followed by a null terminator.

17.5

FESRT_NAME_AND_ATTR

FileEnum() will return an array of **FENameAndAttr** structures, each one of which contains a file's DOS attributes and virtual name.

The **FEDosInfo** structure includes a word-sized record which describes the file's position relative to the standard paths. It contains the following fields:

DPI_EXISTS_LOCALLY

This bit is set if the file exists in a directory under the primary tree.

DPI_ENTRY_NUMBER_IN_PATH

This is the mask for a seven-bit field whose offset is DPI_ENTRY_NUMBER_IN_PATH_OFFSET.

DPI_STD_PATH

This is the mask for an eight-bit field whose offset is DPI_STD_PATH_OFFSET. If the file is in a standard path, this field will contain a **StandardPath** constant for a standard path containing the file. This need not be the "closest" standard path; for example, if the file is in the "World" directory, this constant might nevertheless be SP_TOP.

Code Display 17-2 Standard FileEnum() Return Structures

```
/* These structures are the standard FileEnum() return types. You can also
 * instruct FileEnum() to return any arbitrary extended attribute information.
 */

typedef struct {          /* These are returned if you specify FESRT_DOS_INFO */
    FileAttrs    FEDI_attributes;          /* File's DOS attributes;
                                           * see section 17.5.6.3 on page 668 */
    FileDateAndTime FEDI_modTimeDate      /* Last modification time; see page 647 */
}
```

```

    dword      FEDI_fileSize;          /* Size of file (in bytes) */
    FileLongName FEDI_name;            /* File's virtual name; null-terminated */
    DirPathInfo FEDI_pathInfo;        /* PathInfo structure, described above */
} FEDosInfo;

typedef struct{                       /* These are returned if you specify FESRT_NAME_AND_ATTR */
    /*
    FileAttrs   FENAA_attributes;      /* File's DOS attributes;
                                        * see section 17.5.6.3 on page 668 */
    FileLongName FENAA_name;          /* File's virtual name; null-terminated */
    */
} FENameAndAttr;

```

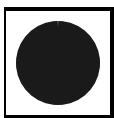
17.5

There is currently one standard callback routine provided. This routine, **FileEnumWildcard()**, rejects all files whose names don't match a wildcard string. To call this routine, set *FEP_callback* as follows:

```
enumParams.FEP_callback = (void *) FESC_WILDCARD;
```

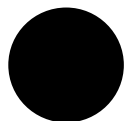
FEP_cbData1 should be a pointer to a null-terminated wildcard string. In this string, “*” denotes a sequence of zero or more of any character, and “?” denotes any one character; all other characters represent themselves. If a file's virtual name matches the wildcard string, information about it will be returned by **FileEnum()**. If the lowest byte of *FEP_cbData2* is non-zero, the comparison will be case-insensitive; otherwise, it will be case-sensitive. (The other bytes of *FEP_cbData2* are ignored.) This is different from standard DOS wildcarding; for example, the name “doc*.” matches any file that begins with the letters “doc” and ends with a period, whereas a DOS wildcard string “doc*.” would match only those files whose name start with “doc” and which have no extension.

17.5.6 Bytewise File Operations



Advanced Topic

There are several routines designed for working with files as a string of bytes. These routines may be used to work with DOS files or with GEOS byte files. You can open any file (including an executable file or a VM file) for byte-level access. This may be useful for such things as file-compression routines; however, be aware that if you make any changes to such files, you could invalidate them. For this reason, if you open a VM or executable file for byte-level access, you should open it for read-only use.



17.5.6.1 Opening and Closing Files

```
FileOpen(), FileCreate(), FileCreateTempFile(),  
FileClose(), FileAccessFlags
```

17.5

The GEOS file system provides several routines for opening files for byte-level access. If you are working with GEOS Virtual Memory files, you should use the appropriate VM routines to open and close the files (see “Virtual Memory,” Chapter 18). You should use the byte-level routines only if you are working with DOS files or with GEOS byte files. You may occasionally want to read a VM file or an executable file as a string of bytes. In this rare case, you must use the routines in this section. Note, however, that you should not change the VM file with these routines; it is safe only to open it for read-only access.

To open a file, call **FileOpen()**. This routine takes two arguments: a set of **FileAccessFlags** and a pointer to a null-terminated string. The string should specify the name of the file (either the virtual name or the native name may be used). It may simply be a file name, or it may be a relative or absolute path. The **FileAccessFlags** record specifies two things: what kind of access the caller wants, and what type of access is permitted to other geodes. A set of **FileAccessFlags** is thus a bit-wise OR of two different values. The first specifies what kind of access the calling geode wants and has the following values:

FILE_ACCESS_R

The geode will only be reading from the file.

FILE_ACCESS_W

The geode will write to the file but will not read from it.

FILE_ACCESS_RW

The geode will read from and write to the file.

The second part specifies what kind of access other geodes may have. Note that if you try to deny a permission which has already been given to another geode (e.g. you open a file with **FILE_DENY_W** when another geode has the file open for write-access), the call will fail. The following permissions can be used:

FILE_DENY_RW

No geode may open the file for any kind of access, whether read, write, or read/write.

FILE_DENY_R

No geode may open the file for read or read/write access.

FILE_DENY_W

No geode may open the file for write or read/write access.

FILE_DENY_NONE

Other geodes may open the file for any kind of access.

Two flags, one from each of these sets of values, are combined to make up a proper **FileAccessFlags** value. For example, to open the file for read-only access while prohibiting other geodes from writing to the file, you would pass the flags FILE_ACCESS_R and FILE_DENY_W as follows:

17.5

```
myHandle = FileOpen("MyFile",  
                    (FILE_ACCESS_R | FILE_DENY_W));
```

If successful, **FileOpen()** returns the file's handle. If it is unsuccessful, it returns a null handle and sets the thread's error value. The following error values are commonly returned:

ERROR_FILE_NOT_FOUND

No file with the specified name could be found in the appropriate directory.

ERROR_PATH_NOT_FOUND

A relative or absolute path had been passed, and the path included a directory which did not exist.

ERROR_TOO_MANY_OPEN_FILES

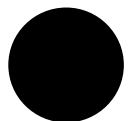
There is a limit to how many files may be open at once. If this limit is reached, **FileOpen()** will fail until a file is closed.

ERROR_ACCESS_DENIED

Either the caller requested access which could not be granted (e.g. it requested write access when another geode had already opened the file with FILE_DENY_W), or the caller tried to deny access when that access had already been granted to another geode (e.g. it tried to open the file with FILE_DENY_W when another geode already had it open for write-access).

ERROR_WRITE_PROTECTED

The caller requested write or read-write access to a file in a write-protected volume.



Note that if you use the document control objects, they automatically make all appropriate calls to **FileOpen()** when the user requests it; you will automatically be passed the file's handle.

FileOpen() can only be called if the file already exists. In order to create a byte file, you must call **FileCreate()**. **FileCreate()** takes four arguments: a set of **FileCreateFlags**, a set of **FileAccessFlags**, a set of **FileAttrs**, and a pointer to a string containing a name for the file. As with **FileOpen()**, the name may be a name alone or a relative or absolute path. The **FileCreateFlags** specifies whether the file should be created if it already exists. The following flags are available:

FILE_CREATE_TRUNCATE

If a file with the given name exists, it should be opened and truncated; that is, all data should be deleted.

FILE_CREATE_NO_TRUNCATE

If the file exists, it should be opened without being truncated.

FILE_CREATE_ONLY

If the file exists, the routine should fail and set the thread's error value to **ERROR_FILE_EXISTS**.

FCF_NATIVE

This flag is combined with one of the above flags if the file should be created in the device's native format; e.g. if it should be a DOS file instead of a GEOS file. The name passed must be an acceptable native file name. If a GEOS file with the specified name already exists, **FileCreate()** will fail with error condition **ERROR_FILE_FORMAT_MISMATCH**.

The first three flags (**FILE_CREATE_...**) are mutually exclusive; exactly one of them must be passed to **FileCreate()**. That flag may or may not be combined with **FCF_NATIVE**.

The **FileAccessFlags** are the same as described in **FileOpen()**. Note, however, that you must request either write access or read/write access when you use **FileCreate()**.

Every file has a set of attributes. These record certain information about the file. If you create a file, you will need to specify values for these attributes. The attributes are described above in the section on "**FEA_FILE_ATTR**" on page 648.

If **FileCreate()** is successful, it will open the file and return its handle. If it fails, it will return a null handle and set the thread's error value. It may return any of the **FileOpen()** errors. It may also return the following errors:

ERROR_FILE_EXISTS

Returned if **FileCreate()** was called with **FILE_CREATE_ONLY** and a file with the specified name already exists.

ERROR_FORMAT_MISMATCH

Returned if **FileCreate()** was called with **FILE_CREATE_TRUNCATE** or **FILE_CREATE_NO_TRUNCATE** and a file exists in a different format than desired; i.e. you passed **FCF_NATIVE** and the file already exists in the **GEOS** format, or vice versa.

17.5

It is often useful to create temporary files which are not seen by the user. In these cases, you generally don't care about the file's name since you will most likely be deleting the file on exit. For these situations **GEOS** provides the routine **FileCreateTempFile()**. **FileCreateTempFile()** is passed a directory; it chooses a unique name for the file. This routine takes two arguments:

- ◆ A set of **FileAttrs**, as described above.
- ◆ A pointer to a null-terminated string specifying the path for the temporary file. This path may be relative or absolute. To create the temporary file in the current directory, pass the string "." This string should contain fourteen extra null bytes at the end, as **FileCreateTempFile()** will write the name of the temporary file at the end of the string. Temporary files are typically created in **SP_WASTE_BASKET**.

If successful, **FileCreateTempFile()** will open the temporary file and return its handle. It will also write the file's name to the end of the string passed. You will need to know the name to delete the file. The name is also useful if **GEOS** shuts down while a temporary file is open; the geode will need to know the temporary file's name in order to reopen it.

When you are done with a file, you should close it by calling **FileClose()**. This releases any restrictions you may have placed on the file and allows the file to be moved or deleted. It is passed two arguments: the file handle and a Boolean value which should be set to *true* (i.e. non-zero) if the geode cannot handle error messages; it will cause **FileClose()** to fatal-error if it cannot



successfully close the file. (This should only be used during development; the flag should never be passed in a finished program.) The routine returns zero if successful; otherwise, it returns a **FileError** value.

17.5.6.2 Reading From and Writing To Files

`FileRead()`, `FileWrite()`, `FilePos()`, `FileCommit()`

17.5

There are a few specific operations you are allowed to perform on data in a byte-file. You can copy data from the file into memory; you can copy data from memory into the file, overwriting the file's contents; you can write data to the end of a file; and you can cut data from the end of the file. If you want to perform more elaborate manipulations on a byte-file, you may wish to create a temporary VM file and copy the data there (see "Virtual Memory," Chapter 18).

Every file handle has a *file position* associated with it. All read and write operations begin at that position; they may also change the position. The first byte in a file is considered to be at position zero. If the file is a GEOS byte file, position zero is immediately after the GEOS header; thus, the header cannot be accessed or altered via the read and write operations.

To read data from a file, call **FileRead()**. This routine takes four arguments. The first is the file's handle. The second is a Boolean indicating whether the caller can handle errors. (This is *true* if the geode cannot handle error messages; it will cause **FileRead()** to fatal-error if it cannot successfully read the data. This should only be used during development; the flag should never be passed in a finished program.) The third is the number of bytes to read. The fourth is the address of a buffer. **FileRead()** will copy the requested number of bytes from the file to the buffer. It will return the number of bytes actually read. This may be less than the number requested, if the end of file is reached; in this case, the thread's error value will be set to `ERROR_SHORT_READ_WRITE`. If **FileRead()** was unable to gain access to the file, it will return -1 and set the thread's error value to `ERROR_ACCESS_DENIED`. In any event, the file position will be incremented by the number of bytes read; thus, it will point to the first byte after the data read.

To write data to a file, call **FileWrite()**. This routine takes four arguments. The first is the file's handle. The second is a Boolean indicating whether the

caller can handle errors. The third is the number of bytes to write. The fourth is the address of a buffer in memory. **FileWrite()** will copy the specified number of bytes from the buffer to the file, starting at the current position and expanding the file as necessary. It will also increment the current position by the number of bytes written. If the current position is not at the end of the file, **FileWrite()** will overwrite the file's existing data. **FileWrite()** returns the number of bytes written. This may be less than the number requested, if the disk ran out of space; in this case, the thread's error value will be set to `ERROR_SHORT_READ_WRITE`. If **FileWrite()** could not get access to the file (as, for example, if the geode had read-only access to the file), it will return -1 and set the thread's error value to `ERROR_ACCESS_DENIED`.

17.5

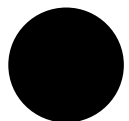
If a file is on a removable disk, the kernel will make sure that the disk is in the appropriate drive before reading from or writing to it. If the disk is not in the drive, the kernel will prompt the user to insert it. The user will have the option of aborting the operation; this will result in the file-access routine failing with error condition `ERROR_DISK_UNAVAILABLE`.

When you write changes to a file, either the GEOS file system or the underlying DOS may choose to cache those changes to save time. All cached changes will be written to the disk when the file is closed. However, you can force the cached changes to be written immediately by calling **FileCommit()**. This routine takes two arguments. The first is the file's handle. The second is a Boolean indicating whether the caller can handle errors. The routine returns zero if the operation was successful; otherwise it returns an error code.

To change the current file position, call **FilePos()**. This routine takes three arguments. The first is the file handle. The second is a member of the **FilePosMode** enumerated type; this value indicates how the new position is specified. The third argument is a number of bytes; it specifies how far the file position will be moved. **FilePosMode** has the following possible values:

`FILE_POS_START`

The file position is set to a specified number of bytes after the start of the file. Passing this mode with an offset of zero will set the file position to the start of the file (i.e. immediately after the header information).



FILE_POS_RELATIVE

The file position is incremented or decremented by a specified number of bytes.

FILE_POS_END

The file position is set to a specified number of bytes before the end of the file. Passing this mode with an offset of zero will set the file position to the end of the file.

17.5

FilePos() returns a 32-bit integer. This integer specifies the file position after the move (relative to the start of the file). To find out the current file position without changing it, call **FilePos()** with mode **FILE_POS_RELATIVE** and offset zero.

17.5.6.3 Getting and Setting Information about a Byte File

`FileGetDateAndTime()`, `FileSetDateAndTime()`,
`FileGetAttributes()`, `FileSetAttributes()`

GEOS provides several routines to get information about files. To get information about a GEOS file, you would ordinarily use one of the extended attributes routines (see section 17.5.3 on page 643). These routines are ordinarily used for non-GEOS files. Nevertheless, all of the following routines can be used on GEOS files.

FileGetDateAndTime() and **FileSetDateAndTime()** are used to get and set the file's modification time. To access a GEOS file's modification time, you would ordinarily call an extended attribute routine, passing **FEA_MODIFICATION**. However, special-purpose routines are provided specifically for changing a file's modification time. Note that these routines may be used for GEOS or non-GEOS files. Similarly, you can change the **FEA_MODIFICATION** attribute even for non-GEOS files. To find out the modification time, call **FileGetDateAndTime()**. This routine is passed the file's handle and returns a **FileDateAndTime** value (as described above on page 647). To change the modification time, call **FileSetDateAndTime()**. This routine is passed the file's handle and a **FileDateAndTime** value. If successful, it returns zero; otherwise, it returns an error code. You must have write permission to change the modification time; otherwise, **FileSetDateAndTime()** will fail with condition **ERROR_ACCESS_DENIED**.

To find out a DOS file's attributes, call **FileGetAttributes()**. This routine is passed a file's path. It returns the file's **FileAttrs** record (as described on page 648). To change the file's attributes, call **FileSetAttributes()**. This routine takes two arguments: the address of a null-terminated path string and a **FileAttrs** record. It returns zero if it was successful; otherwise, it returns an error condition. Note that a file's attributes cannot be changed if the file is open.

17.5.6.4 Data-Access Synchronization

17.5

`FileLockRecord()`, `FileUnlockRecord()`

GEOS provides routines to help threads synchronize file access. This functionality is very elaborate for VM files. For byte files it is less so. Several threads can synchronize their access to a single handle by using **HandleP()** and **HandleV()**, described in section 15.3.6 of chapter 15. If they want to use the file at the same time, they should use **FileLockRecord()** and **FileUnlockRecord()**.

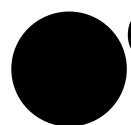
FileLockRecord() takes three arguments: the file handle, a dword specifying the start of the region to be locked, and a dword specifying the length (in bytes) of the region to be locked. If there are no locks on any part of that region, **FileLockRecord()** returns zero; otherwise, it returns the error code `ERROR_ALREADY_LOCKED`. Note that there is nothing to stop another thread or geode from reading or writing to that region. The lock simply prevents anyone from *locking* that region with **FileLockRecord()**. The file's users have to remember to lock any part of the file before accessing it.

To release a lock on a part of a file, call **FileUnlockRecord()**. This routine takes the same arguments as **FileLockRecord()**.

File System

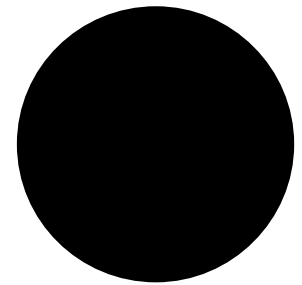
670

17.5



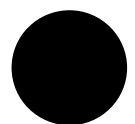
Concepts book

Virtual Memory



18

18.1	Design Philosophy	673
18.2	VM Structure	674
18.2.1	The VM Manager	675
18.2.2	VM Handles.....	676
18.2.3	Virtual Memory Blocks	677
18.2.3.1	The Nature of VM Blocks	677
18.2.3.2	Creating and Using VM Blocks.....	678
18.2.3.3	File Compaction	679
18.2.3.4	File Updating and Backup	680
18.2.4	VM File Attributes.....	681
18.3	Using Virtual Memory	683
18.3.1	How to Use VM	683
18.3.2	Opening or Creating a VM File.....	684
18.3.3	Changing VM File Attributes	687
18.3.4	Creating and Freeing Blocks	687
18.3.5	Attaching Memory Blocks.....	689
18.3.6	Accessing and Altering VM Blocks	690
18.3.7	VM Block Information.....	692
18.3.8	Updating and Saving Files	693
18.3.9	Closing Files.....	695
18.3.10	The VM File's Map Block	696
18.3.11	File-Access Synchronization.....	697
18.3.12	Other VM Utilities.....	699
18.4	VM Chains	700
18.4.1	Structure of a VM Chain	701
18.4.2	VM Chain Utilities	703
18.5	Huge Arrays	705
18.5.1	Structure of a Huge Array	706



18.5.2	Basic Huge Array Routines	708
18.5.3	Huge Array Utilities	713



Most disk-operating systems provide minimal file functionality. A file is simply treated as a sequence of bytes; applications can copy data from files to memory or vice versa, but nothing more elaborate. GEOS provides much more elaborate functionality with its *Virtual Memory* files.

Virtual Memory (VM) is very useful for two reasons. First, it is often impractical to read an entire disk file into contiguous memory; indeed, it is impossible to do so if the file is larger than 64K. The use of virtual memory allows each file to grow arbitrarily large. Second, each disk file is one long, cumbersome stream of data. By using virtual memory files, applications can break files down into smaller, more manageable blocks which the memory manager can handle more efficiently.

18.1

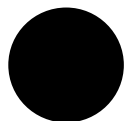


Before you read this chapter, you should have read “Handles,” Chapter 14, and “Memory Management,” Chapter 15. You should also be familiar with the GEOS file system; you should have at least skimmed “File System,” Chapter 17, although you need not have read it in depth.

18.1 Design Philosophy

The GEOS Virtual Memory file management system is designed to provide features not easily available with standard file systems. Designed primarily for use by applications for long-term data storage, it is also used by the system for many other purposes, such as to provide state saving (in the UI). The main benefits it provides to applications include the following:

- ◆ Convenient file structure
GEOS divides VM files into VM blocks, each of which is accessed via a VM block handle. This structure is much like a “disk-based heap,” and is analogous to (and compatible with) the memory heap. VM blocks are accessed the same way whether they are on disk or resident in memory. They can be independently resized and locked.
- ◆ Ease of file manipulation
Many file manipulation techniques are much simpler with VM files. For example, geodes do not have to keep track of which blocks they have changed; instead, when they change a block, they mark it as dirty, and



when they update the file, the virtual memory routines will automatically copy just the changed blocks to the disk.

◆ **Sharing of Data Files**

GEOS maintains information about each VM file and knows when more than one thread is using it. This allows the system to notify all users when one thread has modified a file. The system provides data synchronization for individual blocks.

18.2

◆ **Integrity of Data Files**

GEOS provides several features to protect the integrity of VM files. For example, you can request that GEOS maintain a backup that can be used to restore the file to its last-saved state. In addition, you can have GEOS do all writing to the file synchronously to ensure that all the data in the file stays consistent even if GEOS is somehow interrupted.

◆ **Uniform File Structure**

Because all GEOS data files are based on VM files, utilities can be written which work for all VM files. For example, the Document Control objects can take care of opening, closing, and saving all data files which are based on VM files. Furthermore, data structures can be designed which can be added at will into any VM file.

18.2 VM Structure

Virtual memory can be thought of as a heap stored in a disk file. Like the global heap, it is divided into blocks which are 64K or smaller; each block is accessed via a handle. Blocks can be locked, which brings them into main memory. If blocks are modified while in memory, they are copied back to the file when the file is updated.

The primary component of virtual memory is the VM file. The VM file consists of a VM File Header, a collection of VM blocks, and a special structure called a VM Header. The VM File Header is a standard GEOS file header, containing the file's extended attributes and system bookkeeping information. Geodes may not access it directly; instead, they can make calls to the *extended attributes* routines to access data in the header. (See section 17.5.3 of chapter 17.) The VM blocks and the VM Header do not occupy fixed places in the file. In particular, the VM Header does not necessarily come before all the blocks. Instead, the VM File Header stores the offset within the file to the VM

Header, and the VM Header stores information about the blocks (such as their locations in the file). Furthermore, the blocks in a VM file are arranged in no particular order; they are not necessarily arranged in the order they were created, or in any other sequence. See Figure 18-1 below for a diagram of a VM file.

The VM Header maintains all the bookkeeping information about the VM file. For example, it contains the VM Block Table. The block table is much like the global handle table. Block handles are offsets into the block handle table; a block's entry in the table contains information about the block, such as the block's location in the file. Usually, the Block Table contains entries for blocks that haven't been created yet; when all of these handles have been used, the VM Manager expands the block table. For details, see section 18.2.3 on page 677.

18.2

18.2.1 The VM Manager

The VM manager can be thought of as a memory manager for a disk-based heap, providing all the services a memory manager would and more. The VM manager provides for allocation, use, and deallocation of virtual memory blocks. It manages the Block Table, enlarging it as necessary when more VM block handles are needed. It keeps track of which VM blocks have been loaded into memory, which are currently in use, and which have been "dirtied" (modified) since they were read from the disk; it also keeps track of how many

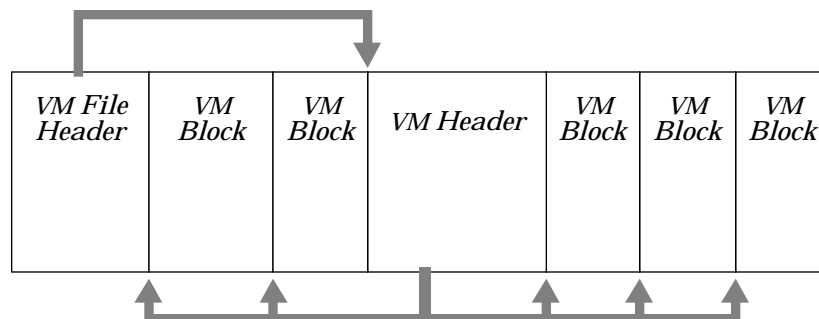
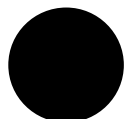


Figure 18-1 *Structure of a VM File*

The VM File Header contains information about the VM file and the offset to the VM Header; the VM Header contains information about the VM Blocks and their offsets.



threads are accessing a given VM file at any time. The VM manager also accomplishes all swapping and read/write operations involving VM files.

18.2.2 VM Handles

There are several different types of handles associated with VM files. It is important not to get them confused.

18.2

- ◆ **File Handles**

When you open or create a VM file, it is assigned a file handle, as discussed in “File System,” Chapter 17. Whenever you call a VM routine, you must specify the VM file by passing this handle. The file handle can change each time the file is opened. Furthermore, if two different network users have the same VM file open at the same time, each user might have a different file handle for the same file.

- ◆ **VM Block Handles**

The VM file has a block table in the VM Header. The block table contains the locations of blocks in the file. A given block has the same block handle every time the file is opened. If a file is duplicated, blocks in the new file will have the same VM handles as the corresponding blocks in the old file. In this chapter, references to “block handles” or “VM handles” are referring to VM block handles unless otherwise noted.

- ◆ **Memory Block Handles**

When a VM block is locked, it is copied into a memory block. This memory block has an entry in the global handle table. The memory block may persist even after the VM block has been unlocked. Ordinarily, you can refer to a VM block by its VM block handle whether or not it is resident in memory; however, in some cases, you may want to use the memory block handle instead of the VM block handle. This saves time when you know the block is resident in memory, because the VM manager doesn’t have to translate the VM block handle into the corresponding global handle. It is also necessary if you need to resize or dirty a VM block. You can instruct the VM manager to use the same global handle each time a given VM block is locked until the file is closed; otherwise, the memory handle for a VM block may change each time the block is locked.

- ◆ **Chunk Handles and Database/Cell Handles**

A VM block can contain a local memory heap. If so, that block will have its own chunk handle table. Also, the Database and Cell libraries have been designed to let the VM file efficiently manipulate small pieces of

data. These libraries are based on the LMem library. Each item of data has its own DB handle as well as a VM Block handle. Database items are discussed in depth in “Database Library,” Chapter 19.

18.2.3 Virtual Memory Blocks

Most file systems treat files as a string of bytes. A user can read a file sequentially or can start reading at a specified distance into the file. This makes it difficult to work with large files. Reading an entire large file into memory may be impractical, and it may be difficult to access different parts of the file at once.

18.2

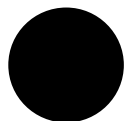
For this reason, GEOS divides its Virtual Memory files into VM blocks. This division is entirely internal to GEOS. When the file is written to disk, it is still a sequence of bytes, and other operating systems can copy the file normally. However, GEOS geodes can access the file much more conveniently by specifying the blocks they wish to access.

18.2.3.1 The Nature of VM Blocks

A VM block is a sequence of bytes in a VM file. It must be small enough to fit in a global memory block (i.e. 64K or less, preferably 2K-6K). It may move about in the file; for this reason, it is accessed by a VM block handle. Blocks are either *free* or *used*. A used block has an entry in the block table and also a space allocated in the file. (This could be a block of free space, which will be freed the next time the file is compacted.) A free block has a slot in the file's handle table but no space in the file; it is available if a thread needs to allocate a block.

Blocks persist after a file has been closed and GEOS has been shut down. A given block is always accessed by the same block handle. There are utilities to copy blocks within a VM file or between files. Blocks in a VM file are in no particular order. If an application wants to set up a sequence of blocks, it can create a VM Chain, in which the first word of each block is the handle of the next block in the chain. However, even chained blocks will probably not be in order in the actual file.

Each VM block can be assigned a “user ID” number. You can request the handles of the VM blocks with any given ID number. You do not have to assign



ID numbers to blocks, but it is sometimes convenient. The ID numbers are stored in the handles' entries in the block table, not in the blocks themselves; this makes it easy to find a block with a specified user ID. User IDs can be changed at will with the routine **VMMModifyUserID()**. Note that all user IDs from 0xff00 to 0xffff are reserved for system use. You can find a block with a specific user ID by calling **VMFind()**; see page 691.

18.2

18.2.3.2 Creating and Using VM Blocks

There are two ways you can create a VM block. The first is to request a VM block: You specify the size of the block, the block is created, and you are returned the handle. This is the method ordinarily used. The second method is to attach memory to a block: You create a global memory block, and instruct the VM manager to add it to the VM file. There are sometimes advantages to this technique; for example, you can create an LMem heap,

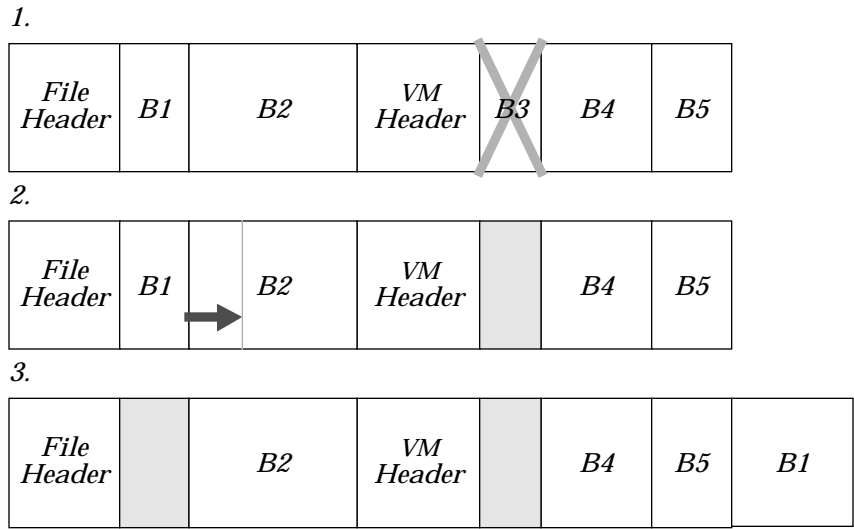


Figure 18-2 A Fragmenting VM File
As blocks are freed and resized, the percentage of the file being used steadily decreases:
1. A VM file with several blocks. The application is now freeing block 3.
2. Block 3 has been freed. The application now needs to expand block 1.
3. Block 1 has been moved to the end of the file, where it has room to grow. The file now contains a lot of unused space.

and then attach it to the VM file; it will then be saved with the file. You can also attach a memory block to an existing VM block; this will destroy whatever used to be in the VM block and replace it with the contents of the new block.

You can dynamically resize a VM block by locking it into memory, resizing the memory block, and saving it back to the disk.

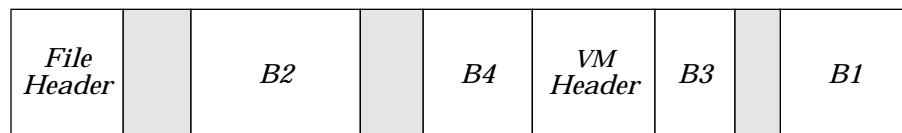
These techniques are described in detail in “Creating and Freeing Blocks” on page 687 and “Attaching Memory Blocks” on page 689.

18.2

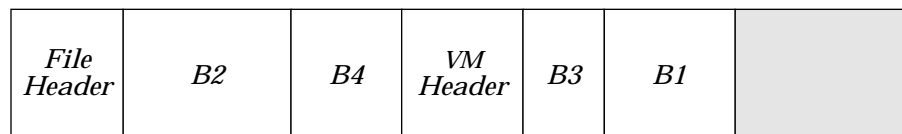
18.2.3.3 File Compaction

When a VM block is freed, the VM manager will note that there is empty space in the file. It will use that space when new blocks are allocated.

1.



2.



3.

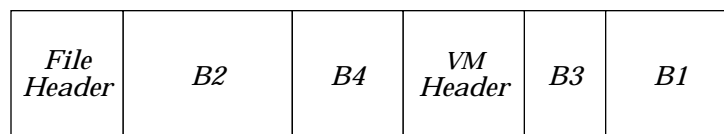


Figure 18-3 VM File Compaction

When the percentage of a file which contains data falls below the “compression threshold,” the VM manager compacts the file.

1. A fragmented file.
2. The VM Manager first copies all the data to the beginning of the file, leaving the free space at the end. It updates the File Header and VM Header appropriately.
3. The VM Manager then deletes the free bytes from the end of the file.



However, since new blocks will probably not fit exactly in the spaces left by freed blocks, some space may be wasted. (See Figure 18-2 below.)

In time, the percentage of wasted file space can grow unacceptably large. To prevent this, the Virtual Memory manager periodically compacts the files. When the ratio of data to free space drops below a certain threshold, the Virtual Memory manager copies the data in the file over the free space (see Figure 18-3 on page ● 679). While a file is being compacted, any requests for access to the file will block until compaction is finished. Note that the format of the data is not changed; the free space between data blocks is simply removed.

When a geode creates a VM file, it can specify a “compression threshold.” When the percentage of used space drops below this threshold, the VM manager will automatically compact the file without any attention from the application. The geode should take care in setting the threshold. If the threshold is too low, the file may grow unacceptably large before it is compacted; on the other hand, if the threshold is too high, the VM manager might spend too much time compacting the file for relatively low gains. The application can specify a threshold of zero; this will cause the system default threshold to be used.

Note that if a file is in “backup mode,” the file will be compacted only on calls to **VMSave()**, **VMSaveAs()**, or **VMRevert()**. If the file is not in backup mode, it can be compacted on any call to **VMUpdate()**.

18.2.3.4 File Updating and Backup

When a block is locked into memory, the VM manager copies the data from the disk block to the global memory block. When the block is unlocked, the VM manager assumes that it can discard the memory block, since the data is already on the disk in the VM block.

If you alter the data in a block, you must notify the VM manager of this fact. You do this by marking the block as *dirty*. When a block has been marked dirty, the VM manager knows that the version in memory is more up-to-date than the version in the disk file. If the flag **VMA_SYNC_UPDATE** is *off* (the default), the block will be written back to the file as soon as possible. If the attribute is *on*, the block will not be copied back to the disk file until **VMUpdate()** is called; until then, the block will be copied to the disk swap

space if memory is needed. The next time you lock the block, you will be given the new, changed version.

When you want to write the dirty blocks to the disk, you can instruct the VM manager to *update* the file. This copies all the dirtied blocks back to the disk and marks all blocks as *clean*. The VM manager also automatically updates the file when it is closed. The updating routines check if the file is currently clean; thus, very little time is lost if you try to update a clean file.

The VM manager can be instructed to notify all users of a file when the file changes from clean to dirty. This has two main uses: it helps maintain data synchronization if many geodes are using the same file, and it lets the document control objects know when to enable the “Save” and “Revert” triggers. (See “GenDocument,” Chapter 13 of the Object Reference Book.)

18.2

The VM manager can be instructed to maintain a backup of a file. If it is so instructed, it will not overwrite the original block when it updates it; instead, it will keep a copy of the original block as well as a copy of the new version. This is transparent to the application. When the VM Manager is instructed to *save* the file, it deletes all the backup blocks. If it is instructed to *revert* the file to its last-saved version, it replaces the new version of the changed blocks with the backup versions, thus restoring the file to its condition as of the last time it was saved. If the VM manager is instructed to “*save-as*” the file, it will create a new file, which does not contain the backup blocks; that is, it contains the file as it currently exists. It will then revert the original file to its last-saved version and close it.

18.2.4 VM File Attributes

VMAttributes

Each VM file has a set of attributes which determine how the VM Manager treats the file. These attributes are specified by a set of **VMAttributes** flags. When a VM file is created, all of these attributes are off; after a file has been created, you can change the attributes with **VMSetAttributes()** (see section 18.3.3 on page 687). The following flags are available:

VMA_SYNC_UPDATE

Allow synchronous updates only. Instructs VM Manager to update the file only when you call an updating routine



(**VMUpdate()**, **VMSave()**, etc.). This attribute is *off* by default (indicating that the VM manager should feel free to update blocks whenever they are unlocked). You should set this attribute if the file might not be in a consistent state every time a block is unlocked.

VMA_BACKUP

Maintain a backup copy of all data. The file can then be restored to its last stored state. This is described above.

18.2

VMA_OBJECT_RELOC

Use the built-in object relocation routines. This attribute must be set if the VM file contains object blocks.

VMA_NOTIFY_DIRTY

If this attribute is set, the VM Manager will notify all threads which have the VM file open when the file changes from clean to dirty. It notifies threads by sending a **MSG_VM_FILE_DIRTY** to each process that has the file open. (This message is defined for **MetaClass**, so any object can handle it.)

VMA_NO_DISCARD_IF_IN_USE

If this attribute is set, the VM manager will not discard LMem blocks of type **LMEM_TYPE_OBJ_BLOCK** if *OLMBH_inUseCount* is non-zero. This attribute must be set if the file contains object blocks. If this attribute is set, each object block will be kept in memory as long as any thread is using an object in the block.

VMA_COMPACT_OBJ_BLOCK

If set, the VM manager will unrelate generic-object blocks before writing them. It does this by calling **CompactObjBlock()**. This allows a VM file to contain generic object blocks.

VMA_SINGLE_THREAD_ACCESS

Set this if only a single thread will be accessing the file. This allows optimizations in **VMLock()**.

VMA_OBJECT_ATTRS

This is not, strictly speaking, a VM attribute. Rather, it is a mask which combines the flags **VMA_OBJECT_RELOC**, **VMA_NO_DISCARD_IF_IN_USE**, and **VMA_SINGLE_THREAD_ACCESS**. All of these attributes must be set if the file contains object blocks.

18.3 Using Virtual Memory

The most common way applications will use Virtual Memory is for their data files. However, there are other uses; for example, VM provides a convenient way to maintain working memory. Some data structures (such as Huge Arrays) can only exist in VM files; an application may create a temporary VM file just for these structures.

18.3

18.3.1 How to Use VM

There are five basic steps to using VM files. The steps are outlined below, and described in greater detail in the following sections.

- ◆ **Open (or create) the VM file**
Before you perform any actions on a VM file, you must open it with **VMOpen()**. This routine can be used to open an existing VM file or create a new one. If you use the document control objects, they will open and create files automatically. Once you have created a VM file, you may want to change its attributes with **VMSetAttributes()**.
- ◆ **Bring a VM block into the global memory heap**
After you open a VM file, you can bring blocks from the file into memory with **VMLock()**. You can also create new blocks with **VMAlloc()** and **VMAttach()**.
- ◆ **Access the data**
Once a VM block has been brought into memory, you can access it the way you would any other memory block. When you are done with the data, you should unlock it with **VMUnlock()**. If you change the memory, you should mark it dirty with **VMDirty()** before unlocking it. This ensures that the new version of the block will be written to the disk.
- ◆ **Update the VM file**
Use one of the several VM updating routines to copy the dirty blocks back to the disk. (If asynchronous update is allowed, the VM file manager will try to update blocks whenever they are unlocked.)
- ◆ **Close the VM file**
Use **VMClose()** when you are done with the file. It will update the file and close it.

18.3.2 Opening or Creating a VM File

`VMOpen()`, `VMOpenTypes`, `VMAccessFlags`

18.3

To create or open a VM file, call the routine **VMOpen()**. You may not need to open and create files directly; if you use the document control objects, they automatically create and open files as the user requests. (See “GenDocument,” Chapter 13 of the Object Reference Book.) **VMOpen()** looks for the file in the thread’s working directory (unless a temporary file is being created, as described below). **VMOpen()** takes four arguments and returns the file handle. The arguments are:

- ◆ **File name**
This argument is a pointer to a string of characters. This string is a relative or absolute path specifying the file to open; if a temporary file is being created, the string is the path of the directory in which to place that file, followed by fourteen null bytes (counting the string-ending null). The name of the temporary file is appended to the path.
- ◆ **VMAccessFlags**
This argument is a set of flags which specifies how the file is accessed. The flags are described below.
- ◆ **VMOpenTypes** enumerated type
This argument specifies how the file should be opened. The **VMOpenTypes** are described below.
- ◆ **Compression threshold**
This is the minimum percentage of a file which must be used for data at any given time. If the percentage drops below this threshold, the file is compacted. If you pass a threshold of zero, the system default threshold is used. The compression threshold is set only when the file is created; this argument is ignored if an existing file is opened.

When you use **VMOpen()**, you must specify how the file should be opened. You do this by passing a member of the **VMOpenTypes** enumerated type. The types are as follows:

`VMO_TEMP_FILE`

If this is passed, the file will be a temporary data file. When you create a temporary file, you pass a directory path, not a file name. The path should be followed by fourteen null bytes,

including the string's terminating null. The system will choose an appropriate file name and add it to the path string.

VMO_CREATE_ONLY

If this is passed, the document will be created. If a document with the specified name already exists in the working directory, **VMOpen()** will return an error condition.

VMO_CREATE

If this is passed, the file will be created if it does not already exist; otherwise it will be opened.

18.3

VMO_CREATE_TRUNCATE

If this is passed, the file will be created if it does not already exist; otherwise, it will be opened and truncated (all data blocks will be freed).

VMO_OPEN

Open an existing file. If the file does not exist, return an error condition.

VMO_NATIVE_WITH_EXT_ATTRS

The file will have a name compatible with the native filesystem, but it will have GEOS extended attributes. This flag can be combined with any of the other **VMOpenType** values with a bit-wise *or*:

You also have to specify what type of access to the file you would like. You do this by passing a record of **VMAccessFlags**. This is a byte-length bitfield. The following flags are available:

VMAF_FORCE_READ_ONLY

If set, the file will be opened read-only, even if the default would be to open the file read/write. Blocks in read-only files cannot be dirtied, and changes in memory blocks will not be updated to the disk VM blocks.

VMAF_FORCE_READ_WRITE

If set, the file will be opened for read/write access, even if the default would be to open the file for read-only access.

VMAF_SHARED_MEMORY

If set, the VM manager should try to use shared memory when locking VM blocks; that is, the same memory block will be used for a given VM block no matter which thread locks the block.



VMAF_FORCE_DENY_WRITE

If set, the file will be opened deny-write; that is, no other threads will be allowed to open the file for read/write access.

VMAF_DISALLOW_SHARED_MULTIPLE

If this flag is set, files with the file attribute GFHF_SHARED_MULTIPLE cannot be opened.

VMAF_USE_BLOCK_LEVEL_SYNCHRONIZATION

If set, the block-level synchronization mechanism of the VM manager is assumed to be sufficient; the more restrictive file-level synchronization is not used. This is primarily intended for system software. (See “File-Access Synchronization” on page 697.)

18.3



If you open a file with VMAF_FORCE_READ_ONLY, it's generally a good idea to also open it with VMAF_FORCE_DENY_WRITE. When you open a file VMAF_FORCE_READ_ONLY, if the file is writable, and is located on a writable device which can be used by other machines (e.g. a network drive), the kernel will load the entire file into memory and make the blocks non-discardable (even when they are clean); this keeps the file you see consistent, even if another user changes the version of the file on the disk. However, this can cause problems if the machine has limited swap space. If the file is opened with VMAF_FORCE_DENY_WRITE, no other device will be allowed to change the file while you have it open, which means the kernel can just load and discard blocks as necessary.

The routine **VMOpen()** returns the file handle. If it cannot satisfy the request, it returns a null handle and sets the thread error word. The error word can be recovered with the **ThreadGetError()** routine. The possible error conditions are:

VM_FILE_EXISTS

VMOpen() was passed VMO_CREATE_ONLY, but the file already exists.

VM_FILE_NOT_FOUND

VMOpen() was passed VMO_OPEN, but the file does not exist.

VM_SHARING_DENIED

The file was opened by another geode, and access was denied.

VM_OPEN_INVALID_VM_FILE

VMOpen() was instructed to open an invalid VM file (or a non-VM file).

VM_CANNOT_CREATE

VMOpen() cannot create the file (but it does not already exist).

VM_TRUNCATE_FAILED

VMOpen() was passed VMO_CREATE_TRUNCATE; the file exists but could not be truncated.

18.3

VM_WRITE_PROTECTED

VMOpen() was passed VMAF_FORCE_READ_WRITE, but the file or disk was write-protected.

18.3.3 Changing VM File Attributes

VMGetAttributes(), VMSetAttributes()

When a VM file is created, it is given a set of **VMAttributes** (see page 681). These attributes can be examined with the routine **VMGetAttributes()**. The routine takes one argument, namely the handle of the VM file (which is overridden if a default VM file is set). It returns the **VMAttributes** flags.

You can change the attributes by calling **VMSetAttributes()**. This routine takes three arguments: the file handle (which may be overridden), a set of bits which should be turned on, and a set of bits which should be turned off. It returns the new **VMAttributes** flags.

18.3.4 Creating and Freeing Blocks

VMAalloc(), VMAallocLMem(), VMFree()

Once you have created a VM file, you have to allocate blocks in order to write data to the file. The usual way to do this is with **VMAalloc()**. This routine takes three word-sized arguments:

- ◆ The file handle
This argument is overridden if a default VM file is set.

- ◆ A user-ID number
This can be any word of data the application wants to associate with the VM block. The application can locate blocks with a given user ID by using **VMFind()** (see page 691).
- ◆ The number of bytes in the block
This may be zero, in which case no memory is allocated; a memory block must be specifically attached with **VMAttach()** (see “Attaching Memory Blocks” on page 689).

18.3

The routine returns the handle of the VM block. Before you can use the block, you have to lock it with **VMLock()**. The block is marked dirty when it is allocated.

There is a routine to allocate a block and initialize it as an LMem heap. This is useful if you are storing object blocks in a VM file. The routine, **VMAllocLMem()**, takes three arguments:

- ◆ The VM file handle
This is overridden if a default VM file is set.
- ◆ A member of the **LMemTypes** enumerated type
This specifies what kind of heap the LMem heap will be. (See section 16.2.3 of chapter 16.)
- ◆ The size of the block header
Use this if you want to store extra data in the LMem block header. To use the standard LMem header, pass an argument of zero. (See section 16.3.1 of chapter 16.)

The routine creates a VM block and allocates a global memory block to go with it. It initializes the heap in the global block and marks the block as dirty. The LMem heap will begin with two LMem handles and a 64-byte heap; this will grow as necessary. The VM block will have a user ID of zero; you can change this if you wish. The routine returns the handle of the new VM block.

There are two other ways to create LMem blocks in a VM file; these ways give you more control of the block's initialization. You can allocate a VM block normally, lock that block, then get the handle of the associated memory block and initialize an LMem heap in it; or you can allocate an LMem heap normally, and attach that memory block to the VM file using **VMAttach()**. For more details on LMem heaps, see “Local Memory,” Chapter 16.

To free a VM block, call **VMFree()**. This routine is passed two arguments: the VM file handle, and the VM block handle. The handle will immediately be freed, even if it is locked. Any associated memory will also be freed. If you want to keep the memory, detach the global memory block from the file (with **VMDetach()**) before you free the block.

18.3.5 Attaching Memory Blocks

18.3

`VMAttach()`, `VMDetach()`

When you use **VMAlloc()**, the VM manager allocates a global memory block and attaches it to a VM block. However, sometimes you want to allocate the block yourself, or you may have an existing memory block which you want to copy into the VM file. To do this, you call the routine **VMAttach()**.

VMAttach() takes three arguments:

- ◆ The VM file handle
The handle of the file to attach.
- ◆ The VM block handle
If you pass a null handle, a free VM block will be allocated and attached to the global memory block. If you pass the handle of an existing block, the data in the VM block will be replaced with the contents of the global memory block.
- ◆ The global memory handle
The memory block must be swappable. After the block is attached, the VM manager may discard or free it, as with any other global blocks used by the VM file.

VMAttach() attaches the global memory block to the VM block. The VM Manager becomes the owner of the memory block. The next time the file is updated, the memory block will be copied to the file. **VMAttach()** returns the handle of the VM block. If it could not perform the attach, it returns a null handle and leaves the global memory block unchanged.

You can also detach the global memory block from a VM block. The routine **VMDetach()** disassociates a global memory block from its VM block. The routine takes three arguments: the VM file handle; the VM block handle; and the **GeodeHandle** of the geode which will be made the owner of the memory block. (Passing a null **GeodeHandle** will make the calling geode the block's

owner.) The VM manager disassociates the memory block from the VM block, changes the memory block's owner, marks it "non-discardable," and returns its handle. If the VM block is not currently in memory, **VMDetach()** will automatically allocate a memory block, copy the VM block's data into it, and return the memory block's handle. If the VM block was dirty, the block will be updated to the file before it is detached. The next time the VM block is locked, a new global memory block will be allocated for it.

18.3

18.3.6 Accessing and Altering VM Blocks

```
VMLock(), VMUnlock(), VMDirty(), VMFind(),  
VMModifyUserID(), VMPreserveBlocksHandle()
```

Once you have opened a VM file and allocated blocks, you will need to access blocks. The VM library provides many routines for doing this.

If you need to access the data in a VM file, you can use the routine **VMLock()**. This routine moves a VM block onto the global heap. It does this by allocating a global memory block (if the VM block is not already associated with a memory block), reallocating the global block if it had been discarded, locking the memory block on the global heap, and copying the VM block into the global block, if necessary. (It will copy the VM block to memory only if necessary, i.e. if the memory block is newly-allocated, or had been discarded and reallocated.) **VMLock()** takes three arguments: the handle of the VM file, the **VMBlockHandle** of the block to lock, and a pointer to a **memHandle** variable. It returns a pointer to the start of the block, and writes the global block's handle into the **memHandle** variable. You can now access the block the same way you would any other block, with one exception: When you are done with the block, you do not call **MemUnlock()**; instead, call the routine **VMUnlock()**, passing it the handle of the *global memory* block (not the handle of the VM block). This will unlock the global block on the heap.



Dirtying Blocks

It is important to call **VMDirty()** while the memory block is still locked; otherwise, the memory block may be discarded before you dirty it.

If you alter the global memory block, you will need to notify the VM manager of this so it will know to copy the changes back to the VM file. You do this by calling the routine **VMDirty()**. **VMDirty()** takes one argument, the handle of the global memory block (*not* the VM block). It is important to dirty the block while it is still locked on the heap; as soon as you unlock a clean block, the VM manager may choose to discard it. Dirty blocks are copied back to the VM file when it is updated. Note that if an object in a VM block is marked

dirty (via **ObjMarkDirty()**), the block is automatically dirtied. Similarly, if you attach a global memory block to a VM block (via **VMAttach()**), the VM block is automatically dirtied.

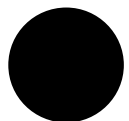
You can dynamically resize VM blocks. To do this, lock the VM block with **VMLock()**; then resize the global memory block with **MemReAlloc()**. Be sure to mark the block dirty so the changes will be copied to the disk file. Note that although the global memory block will remain locked, it may move on the global heap when it is resized. You will therefore need to dereference the global memory handle (with **MemDeref()**) before accessing the memory.

18.3

You can locate VM blocks by their user ID numbers. The routine **VMFind()** takes three arguments: the VM file handle, a VM block handle, and the user ID for which to look. The routine looks through the block table, starting with the handle *after* the one passed, until it finds a block with the specified user ID. If it does not find such a block, it returns a null handle; otherwise, it returns the block's **VMBlockHandle**. Thus, by passing in a block handle of zero, you will get the handle of the first block with the specified ID; by passing back in that block's handle, you will get the next block with that ID; and so on, until you get all the blocks (after which you will be returned a null handle).

You can change a block's user ID number by calling the routine **VMModifyUserID()**. This routine takes three arguments: the VM file handle, the VM block handle, and the new user ID number. Since user IDs are maintained in the block table, not in the blocks themselves, it doesn't matter whether the block is locked, or even whether it is associated with data in the file. (For example, a block allocated with a size of zero can have its user-ID changed.)

Ordinarily, the VM manager can free any unlocked, clean global block if the space is needed. However, you can instruct the VM manager not to free the global block associated with a specific block by calling the routine **VMPreserveBlocksHandle()**. The routine takes two arguments, namely the VM file handle and the VM block handle. It sees to it that the specified VM block will remain attached to the same global block until the VM block is specifically detached (or reattached).



18.3.7 VM Block Information

`VMVMBlockToMemBlock()`, `VMMemBlockToVMBlock()`, `VMInfo()`

Several utilities are provided to give you information about VM blocks.

18.3

If you know the handle of a VM block, you can find out the handle of the associated global block by calling the routine **VMVMBlockToMemBlock()**. This routine takes two arguments, namely the VM file handle and the VM block handle. It returns the global memory handle of the associated block; however, note the caveats regarding global handles in the above section. If the VM block is not currently associated with a global memory block, the routine will allocate a memory block, copy the VM block into it, and return its handle. If the VM handle is not associated with any data in the file and is not attached to a global memory block, **VMVMBlockToMemBlock()** returns a null handle.

Conversely, if you know the handle of a global memory block and want to find out the VM file and block to which it is attached, call the routine **VMMemBlockToVMBlock()**. This routine takes two arguments: the global memory handle, and a pointer to a **VMFileHandle** variable. It returns the VM block handle of the associated VM block, and writes the handle of the VM file to the address passed. If the global memory block is not attached to a VM file, it returns null handles.

The Boolean routine **VMInfo()** is an omnibus information routine. It takes three arguments: the handle of a VM file, the handle of a VM block, and a pointer to a **VMInfoStruct** structure (described below in Code Display 18-1). If the VM block is free, out of range, or otherwise invalid, it returns *false*; otherwise, it returns *true* (i.e. non-zero) and fills in the fields of the **VMInfoStruct**.

Code Display 18-1 VMInfoStruct

```
/* This is the definition of the VMInfoStruct. A pointer to a VMInfoStruct is
 * passed to the routine VMInfo(). The routine fills in the structure's fields.
 */
typedef struct {
    MemHandle      mh;          /* Null handle returned if no block is attached */
```

```

word        size;    /* Size of VM block in bytes */
word        userID; /* User ID (or zero if no user ID was specified)
*/
} VMInfoStruct;

```

18.3.8 Updating and Saving Files

18.3

```

VMUpdate(), VMSave(), VMSaveAs(), VMRevert(),
VMGetDirtyState() VMSave()

```

When you dirty a memory block, that action notifies the VM manager that the block will need to be written back to the file. If the attribute `VMA_SYNC_UPDATE` is *off*, the VM manager will try to update the block to the disk file as soon as the block is unlocked, and will then mark the block as *clean*. However, if the flag is *on*, the manager does not write the block until it is specifically told to *update* the file. At this point, it copies any dirty blocks back over their attached VM blocks, then marks all blocks as *clean*. If you use the document control objects, they will take care of updating and saving the file. However, you may need to call the updating routines specifically.

The routine **VMUpdate()** instructs the VM manager to write all dirty blocks to the disk. It takes one argument, the VM file handle (which is overridden if a thread file has been set). It returns zero if the update proceeded normally; otherwise, it returns either one of the **FileErrors** or one of the three **VMUpdate()** status codes:

VM_UPDATE_NOTHING_DIRTY

All blocks were clean, so the VM disk file was not changed.

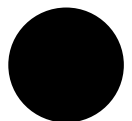
VM_UPDATE_INSUFFICIENT_DISK_SPACE

The file has grown since the last update, and there is not enough room on the disk to accommodate it.

VM_UPDATE_BLOCK_WAS_LOCKED

Some of the VM blocks were locked by another thread, so they could not be updated to the disk.

VMUpdate() is optimized for updating clean files; thus, it costs very little time to call **VMUpdate()** when you are not sure if the file is dirty. If a file is auto-saved, **VMUpdate()** is used.



A VM file can maintain backup copies of updated blocks. If so, updating the file will write changes to the disk, but will not alter those backup blocks. To finalize the changes, call the routine **VMSave()**. This routine updates the file, then deletes all the backup blocks and compacts the file. (See Figure 18-4.) If the file does not have backup capability, **VMSave()** acts the same as **VMUpdate()**.

If a file has the backup capability, you cannot directly access the backup blocks. However, you can instruct the VM manager to restore the file to its last-saved state. The command **VMRevert()** causes the VM manager to check the VM file for blocks which have backups. It then deletes the *non-backup* block, and changes the backup block into a regular block. It also

1.

File Header	4	2'	VM Header	3	1	2
-------------	---	----	-----------	---	---	---

2.

File Header	4'	2'	VM Header	3	1'	2	1	4
-------------	----	----	-----------	---	----	---	---	---

3.

File Header			VM Header	3		2	1	4
-------------	--	--	-----------	---	--	---	---	---

4.

File Header	VM Header	3	2	1	4
-------------	-----------	---	---	---	---

Figure 18-4 Saving a backup-enabled VM file

- 1) This is the file when **VMSave()** is called. Backup blocks are noted with an apostrophe.
- 2) The file is updated (all dirty blocks are written to the disk). This may cause more backup blocks to be created.
- 3) All backup blocks are freed.
- 4) The file is always compacted at the end of a save, whether or not it has fallen below the compression threshold.

discards all blocks in memory that were attached to the blocks which just reverted. The file will then be in its condition as of the last time it was saved. The routine may not be used on files which do not have the flag `VMA_BACKUP` set.

You can save a file under a new name with the routine **VMSaveAs()**. If the file has backup capability, the old file will be restored to its last-saved condition (as if **VMRevert()** had been called); otherwise, the old file will be left in the file's current state. The routine is passed the name of the new file. **VMSaveAs()** copies all the blocks from the old file to the new one. If a block has a backup copy, the more recent version is copied. The new file will thus have the file in its current state; block handles will be preserved. After the new file has been created, if the file has backup-capability, **VMSaveAs()** reverts the original file to its last-saved state. It then closes the old file and returns the handle of the new file.

18.3

If you manage VM files with the document control objects, you generally don't have to call the update or save routines. The document control objects will set up a file menu with appropriate commands ("Save," "Save As," etc.), and will call the appropriate routines whenever the user chooses a command.

If you need to find out whether a file is dirty, call the routine **VMGetDirtyState()**. This routine returns a two-byte value. The more significant byte is non-zero if any blocks have been dirtied since the last update or auto-save. The less significant byte is non-zero if any blocks have been dirtied since the last save, save-as, or revert action. If the file does not have backup-capability, both bytes will always be equal. Note that **VMUpdate()** is optimized for clean files, so it is generally faster to call **VMUpdate()** even if the file might be clean, rather than checking the dirty-state with **VMGetDirtyState()**.

18.3.9 Closing Files

`VMClose()`

When you are done with a VM file for the time being, you should close it with **VMClose()**. This routine updates all the dirty blocks, frees all the global memory blocks attached to the file, and closes the file (thus freeing its handle). The routine is passed two arguments. The first is the handle of the

file to close. The second is a Boolean value, *noErrorFlag*. If this flag is *true*, **VMClose()** will not return error conditions; if it could not successfully close the file, it will fatal-error.

If *noErrorFlag* is *false*, **VMClose()** will update the file and close it. If the file could not be updated, it will return an error condition. Be warned, however, that if for some reason **VMClose()** could not finish updating a file (for example, because the disk ran out of space), **VMClose()** will return an error message, but will close the file and free the memory anyway. Thus, the most recent changes will be lost. For this reason, it is usually safer to first update the file (and handle any error messages returned) and then close it.

When GEOS shuts down, all files are closed. When it restarts, you can open the files manually.

If the file is backup-enabled, the backup blocks will be preserved until the file is next opened. That means, for example, that the next time you open the file, you can continue working on it normally, or you can immediately revert it to its condition as of the last time it was saved (in the earlier GEOS session).

18.3.10 The VM File's Map Block

`VMGetMapBlock()`, `VMSetMapBlock()`

When they're created, the blocks of a VM file are in no particular order. You will need some way to keep track of VM block handles so you can find each block when you need it. The usual way to do this is with a *map block*.

A map block is just like any other VM block. Like other blocks, it can be a standard block, an LMem heap, etc. It is different in only one way: the VM manager keeps track of its handle. By calling the routine **VMGetMapBlock()**, you can get the VM handle of the map block. You can then look inside the map block to get information about the other blocks.

Note that the structure of the VM map block is entirely the concern of the creating geode. The VM manager neither requires nor specifies any internal structure or information content.

To create a map block, allocate a VM block through any of the normal techniques, then pass its VM handle as an argument to **VMSetMapBlock()**.

That block will be the new map block. If there already was a map block, the old block will become an ordinary VM block.

In addition to setting a map block, you can set a *map database item* with the command **DBSetMap()**. For details, see section 19.2.5 of chapter 19.

18.3.11 File-Access Synchronization

18.3

`VMGrabExclusive()`, `VMReleaseExclusive()`

Sometimes several different geodes will need access to the same VM file. Generally, these will be several different copies of the same application, perhaps running on different machines on a network. GEOS provides three different ways shared-access can be handled.

A VM file can be one of three different types: standard, “public,” and “shared-multiple.” By default, all new VM files are standard. The file’s type is one of its extended attributes, and can be changed with the routine **FileSetHandleExtendedAttributes()** (see section 17.5.3 of chapter 17). The document control automatically lets the user select what kind of file to create, and changes its type accordingly. (See “GenDocument,” Chapter 13 of the Object Reference Book.)

Only one geode may write to a standard GEOS VM file at a time. If a geode has the file open for read/write access, no other geode will be allowed to open that file. If a geode has the file open for read-only access, other geodes are allowed to open it for read-only access, but not for read-write access. If a file is opened for read-only access, blocks cannot be dirtied or updated. If a geode tries to open a file for writing when the file is already open, or if the geode tries to open it for reading when the file has already been opened for writing, **VMOpen()** will return an error.

In general, when a file is opened, it is by default opened for read-write access. For example, the document control objects present a dialog box which lets the file be opened for read-only access, but has this option initially turned off. However, some files are used mainly for reference and are infrequently changed. For example, a company might keep a client address book on a network drive. Most of the time, people would just read this file; the file would only occasionally be changed. For this reason, GEOS lets you declare VM file as “public.” Public files are, by default, opened for read-only access. In

all other respects the file is the same as a standard GEOS VM file; it can be opened by several readers at a time, but by only one geode at a time if the geode will be writing.

Sometimes several geodes will need to be able to write to a file at once. For example, a company might have a large customer database, and several users might be writing records to the database at the same time. For this reason, GEOS lets you create “shared-multiple” files. Several geodes can have a “shared-multiple” file open at once. However, a geode cannot access the file whenever it wants. Instead, it must get the file’s semaphore to access the file’s data. When it needs to access the file, it calls **VMGrabExclusive()**. This routine takes four arguments:

- ◆ The handle of the VM file
- ◆ A timeout value
If a timeout value is passed, **VMGrabExclusive()** will give up trying to get the semaphore after a specified number of seconds has passed. If a timeout value of zero is passed, **VMGrabExclusive()** will block until it can get the file’s semaphore.
- ◆ A member of the **VMOperations** enumerated type
This specifies the kind of operation to be performed on the locked file. The *VMOperations* values are described below.
- ◆ A pointer to a word-length variable.
If this call to **VMGrabExclusive()** fails and times out, the operation currently being performed will be written here.

The routine returns a member of the **VMStartExclusiveReturnValue** enumerated type. The following return values are possible:

VMSERV_NO_CHANGES

No other thread has changed this file since the last time this thread had access to the file.

VMSERV_CHANGES

The file may have been altered since the last time this thread had access to it; the thread should take appropriate actions (such as re-reading any cached data).

VMSERV_TIMEOUT

This call to **VMGrabExclusive()** failed and timed out without getting access to the file.

When a geode calls **VMGrabExclusive()**, it must pass a member of the **VMOperations** enumerated type. Most of the values are used internally by the system; while a geode should never pass these values, they may be returned by **VMGrabExclusive()** if the call times out. The following values are defined in **vm.h**:

VMO_READ

This indicates that the geode will not change the file during this access. This lets the kernel perform some optimizations.

18.3

VMO_INTERNAL

VMO_SAVE

VMO_SAVE_AS

VMO_REVERT

VMO_UPDATE

These values are set only by the kernel. Applications should never pass them.

VMO_WRITE

This indicates that the geode may write to the file during this access.

The application may also pass any value between **VMO_FIRST_APP_CODE** and **0xffff**. The kernel treats all these values as synonymous with **VMO_WRITE**; however, the application may choose to associate meanings with numbers in this range (perhaps by defining an enumerated type whose starting value is **VMO_FIRST_APP_CODE**).

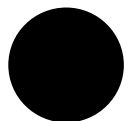
When a thread is done accessing a file, it should release its exclusive access by calling **VMReleaseExclusive()**. The routine takes one argument, namely the file handle. It does not return anything.

18.3.12 Other VM Utilities

VMCopyVMBlock(), **VMSetReloc()**

If you would like to duplicate a VM block, or copy it to another file, call **VMCopyVMBlock()**. This routine is passed three arguments:

- ◆ The **VMFileHandle** of the file containing the source block.



- ◆ The **VMBlockHandle** of the source block.
- ◆ The **VMFileHandle** of the destination file (which may be the same as the source file).

The routine makes a complete duplicate of the source block, copying it to the source file. It returns the **VMBlockHandle** of the duplicate block. Note that the duplicate block will almost certainly have a different block handle than the source block. If the block contains a copy of its own handle, you should update it accordingly.

18.4

Sometimes you will need to perform special actions whenever loading a block into memory or writing it to the disk. For example, you may store data in a compressed format on the disk, and need to expand it when it's loaded into memory. For this reason, you can set a *relocation* routine. This routine will be called whenever the following situations occur:

- ◆ A VM block has just been copied from the disk into memory (routine will be passed the flag `VMRT_RELOCATE_AFTER_READ`).
- ◆ A block is about to be written from memory to the disk (routine is passed `VMRT_UNRELOCATE_BEFORE_WRITE`).
- ◆ A block in memory has just been written to the disk, but is not being discarded (routine is passed `VMRT_RELOCATE_AFTER_WRITE`).
- ◆ A VM block has just been loaded from a resource (routine is passed `VMRT_RELOCATE_FROM_RESOURCE`). This is called by the relocating object, not by the VM manager.
- ◆ A VM block is about to be written to a resource (routine is passed `VMRT_UNRELOCATE_FROM_RESOURCE`). This is called by the unrelocating object, not by the VM manager.

Using the routine **VMSetReloc()**, you can instruct the VM manager to call your relocation routine whenever appropriate.

18.4 VM Chains

A VM file is just a collection of blocks. These blocks are not kept in any particular order. If the application wants to keep the blocks in some kind of order, it must set up some mechanism to do this.

One common mechanism is the VM chain. This is simply a way of arranging blocks in a sequence, in which each block contains the handle of the next block. GEOS has a standard format for VM chains and provides utility routines which work with chains of this format.

In general usage, a “chain” is a simple tree in which each block has a link to at most one other block. However, the GEOS VM chain can also contain special “tree blocks,” which may have links to any number of child blocks. By using these blocks, an application can set up VM trees of unlimited complexity.

18.4

18.4.1 Structure of a VM Chain

A VM chain is composed of two kinds of blocks: chain blocks (which are linked to at most one other block), and tree blocks (which may be linked to any number of other blocks). One block is the head of the chain; chain utility routines can be passed the handle of this block, and they will act on all the blocks in the chain. If a block is a “leaf” block, it should contain a null handle. An example of a VM chain with tree blocks is shown in Figure 18-5 on page ● 701.

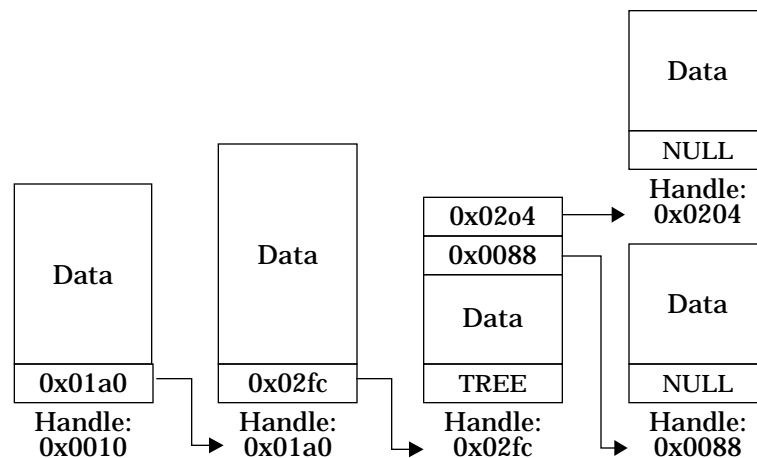
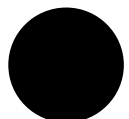


Figure 18-5 A VM Chain

This chain contains a tree node, which allows it to branch. A utility routine can be passed the handle of the head; the utility will then work on the entire chain.





Warning

VM chains must not contain loops.

18.4

Be warned that a VM chain must not contain any circuits. That is, by following links, you should not be able to go from any block back to itself; and there should not be two different routes from any one block to any other. If you create such a VM chain and pass it to a chain utility, the results are undefined. It is your responsibility to make sure no loops occur.

A VM chain block is the same as any other VM block, with one exception: The block must begin with a **VMChain** structure. This structure contains a single data field, *VMC_next*, which is the handle of the next block in the chain. If the block is in a chain but has no next link, *VMC_next* is a null handle. This means, for example, that LMem heaps cannot belong to a VM chain (since LMem heaps must begin with an **LMemHeader** structure).

In addition to chain blocks, a VM chain may contain a tree block. A tree block may have several links to blocks. The structure of a tree block is shown in Figure 18-6. A tree block begins with a **VMChainTree** structure. This structure has three fields:

VMCT_meta

This is a **VMChain** structure. Every block in a VM chain, including a tree block, must begin with such a structure. However, to indicate that this is a tree block, the *VMC_next* field must be set to the special value `VM_CHAIN_TREE`.

VMCT_offset

This is the offset within the block to the first link. All data in the tree block must be placed between the **VMChainTree** structure and the first link. If you will not put data in this block, set this field to **sizeof(VMChainTree)**.

VMCT_count

This is the number of links in the tree block.

Any of the links may be a null handle. To delete the last link in the block, just decrement *VMCT_count* (and, if you wish, resize the block). To delete a link in the midst of a block, just change the link to a null handle without decrementing *VMCT_count*. To add a new link to a VM tree block, you can either add the handle after the last link and increment *VMCT_count*, or you can replace a null handle (if there are any) with the new handle, without changing *VMCT_count*.

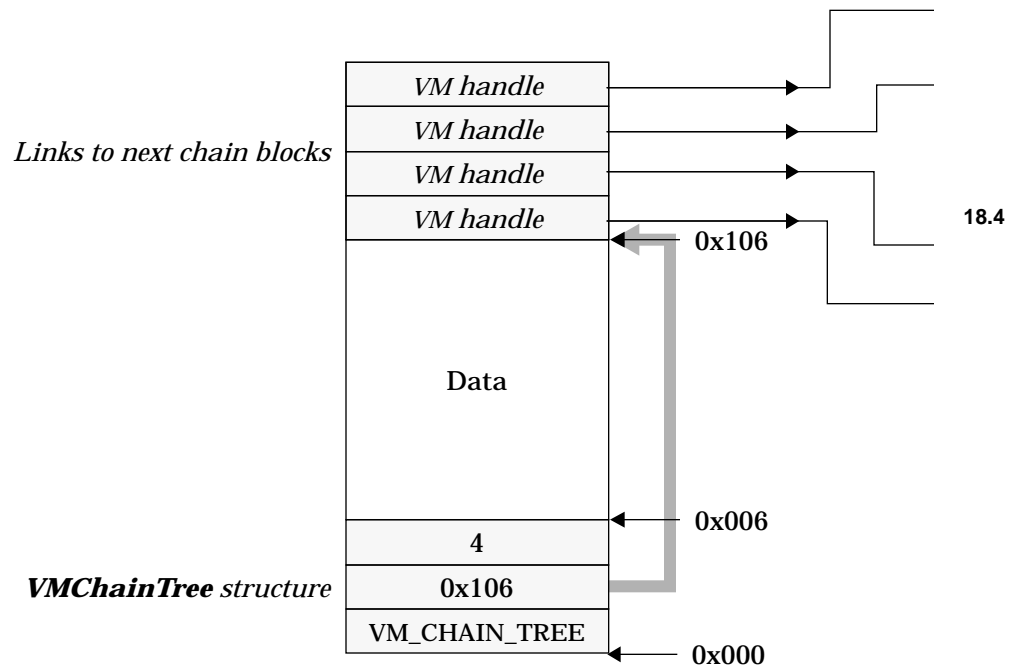


Figure 18-6 Structure of a VM tree block

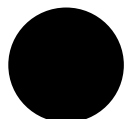
This is a sample VM tree block. It contains links to four other link (or chain) blocks, and enough extra space for 0x100 bytes of data.

18.4.2 VM Chain Utilities

```
VMChainHandle, VMFreeVMChain(), VMCompareVMChains(),
VMCopyVMChain(), VMCHAIN_IS_DBITEM(),
VMCHAIN_GET_VM_BLOCK(), VMCHAIN_MAKE_FROM_VM_BLOCK()
```

Several utilities are provided for working with VM chains. They allow you to compare, free, or copy entire VM chains with a single command.

For convenience, all of these routines can work on either a VM chain or a database item. This is useful, because sometimes you will want to use the utility on a data structure without knowing in advance how large it will be.



This way, if there is a small amount of data, you can store it in a DB item; if there is a lot, you can store it in a VM chain of any length. Whichever way you store the data, you can use the same chain utilities to manipulate it.

The routines all take, as an argument, a dword-sized structure called a **VMChain**. This structure identifies the chain or DB item. It is two words in length. If it refers to a DB item, it will be the item's **DBGroupAndItem** structure. If it refers to a VM chain, the less significant two bytes will be null, and the more significant two bytes will be the VM handle of the head of the chain. Note that none of the blocks in the VM chain need be locked when the routine is called; the routine will lock the blocks as necessary, and unlock them when finished. Similarly, a DB item need not be locked before being passed to one of these routines. However, the VM file containing the structure must be open.

If you want to free an entire VM chain at once, call the routine **VMFreeVMChain()**. This routine takes two arguments, namely the VM file handle and the **VMChain** structure. It frees every block in the VM chain, and returns nothing.

You can compare two different VM chains, whether in the same or in different files, by calling **VMCompareVMChains()**. This Boolean routine takes four arguments, namely the handles of the two files (which may be the same) and the **VMChain** structures of the two chains or items. The geode must have both files open when it calls this routine. The routine returns *true* (i.e. non-zero) if the two chains are identical (i.e. the trees have the same structures, and all data bytes are identical). Note that if the chains contain tree blocks which are identical except in the order of their links, the chains will not be considered identical and the routine will return *false* (i.e. zero).

You can make a copy of a VM chain with the routine **VMCopyVMChain()**. This routine copies the entire chain to a specified file, which may be the same as the source file. The blocks in the duplicate chain will have the same user ID numbers as the corresponding original blocks. The routine takes three arguments: the handle of the source file, the **VMChain** of the source chain or item, and the handle of the destination file. It copies the chain or item and returns the **VMChain** handle of the duplicate structure. As noted, if this structure is a VM chain, the less significant word of the **VMChain** will be null, and the more significant word will be the VM handle of the head of the chain. The geode must have both files open when it calls this routine.

Several utilities are provided for working with **VMChain** structures. They are implemented as preprocessor macros, so they are very fast. The macro **VMCHAIN_IS_DBITEM()** is passed a **VMChain** structure. It returns non-zero if the structure identifies a DB item (i.e. if the less significant word is non-zero); it returns zero if the structure identifies a VM chain. The macro **VMCHAIN_GET_VM_BLOCK()** is passed a **VMChain** structure which identifies a VM chain. It returns the VM handle (i.e. the more significant word of the structure). Finally, the macro **VMCHAIN_MAKE_FROM_VM_BLOCK()** takes a **VMBlockHandle** value and casts it to type **VMChain**.

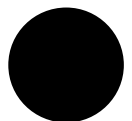
18.5

18.5 Huge Arrays

Sometimes a geode needs to work with an array that can get very large. Chunk arrays are very convenient, but they are limited to the size of an LMem heap, which is slightly less than 64K; furthermore, their performance begins to degrade when they get larger than 6K. Similarly, if a geode uses raw memory for an array, it is limited to the maximum size of a global memory block, again 64K.

For this reason, GEOS provides the Huge Array library. A Huge Array is stored in a VM file. All the elements are stored in VM blocks, as is the directory information. The application can specify an array element by its index, and the Huge Array routine will lock the block containing that element and return its address. Array indices are dword-sized, meaning a Huge Array can have up to 2^{32} elements. Since elements are stored in VM blocks, each element has a maximum size of 64K; however, the size of the entire array is limited only by the amount of disk space available. The blocks in a Huge Array are linked together in a VM chain, so the VM chain utilities can be used to duplicate, free, and compare Huge Arrays.

There are a couple of disadvantages to using Huge Arrays. The main disadvantage is that it takes longer to access an element: the routine has to lock the directory block, look up the index to find out which block contains the element, lock that block, calculate the offset into that block for the element, and return its address. (However, elements are consecutive within blocks; thus, you can often make many array accesses with a single Huge Array lookup command.) There is also a certain amount of memory overhead for



Huge Arrays. Thus, if you are confident that the array size will be small enough for a single block, you are generally better off with a Chunk Array (see section 16.4.1 of chapter 16).

Huge arrays may have fixed-size or variable-sized elements. If elements are variable-sized, there is a slight increase in memory overhead, but no significant slowdown in data-access time.

18.5

18.5.1 Structure of a Huge Array

The huge array has two different type of blocks: a single directory block, and zero or more data blocks. The blocks are linked together in a simple (i.e., non-branching) VM chain. The directory block is the head of the chain. (See Figure 18-7 below.) A Huge Array is identified by the handles of the VM file containing the array and the directory block at the head of the array.

The directory block is a special kind of LMem block. It contains a header structure of type **HugeArrayDirectory** (which begins with an **LMemBlockHeader** structure), followed by an optional fixed data area, which is followed by a chunk array. The chunk array is an array of **HugeArrayDirEntry** structures. There is one such structure for each data block; the structure contains the handle of the data block, the size of the block, and the index number of the last element in the block.

Each data block is also a special LMem block. It contains a **HugeArrayBlock** structure (which begins with an **LMemBlockHeader** structure) and is followed by a chunk array of elements. If the Huge Array has variable-sized elements, so will each data block's chunk array.

When you want to look up a specific element, the Huge Array routines lock the directory block. They then read through the directory chunk array until they find the block which contains the specified element. At this point, the routine knows both which data block contains the element and which element it is in that block's chunk array. (For example, if you look up element 1,000, the Huge Array routine might find out that block *x* ends with element 950 and block *y* ends with element 1,020. The routine thus knows that element 1,000 in the Huge Array is in the chunk array in block *y*, and its element number is 50 in that block's array.)

18.5

The routine then unlocks the directory block, and locks the data block containing that element. It returns a pointer to that element. It also returns the number of elements occurring before and after that element in that chunk array. The calling geode can access all the elements in that block without further calls to Huge Array routines.

When you insert or delete an element, the Huge Array routines look up the element index as described above. They then call the appropriate chunk array routine to insert or delete the element in that data block. They then go through the directory and adjust the element numbers throughout. If inserting an element in a data block would bring the block's size above some

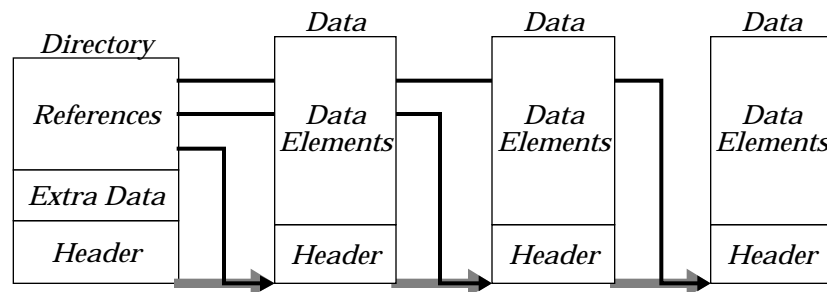
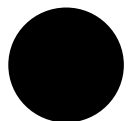


Figure 18-7 Huge Array Structure

The Huge Array comprises a VM chain, which begins with a single directory block, followed by one or more data blocks. The directory block has information on which blocks contain which elements. Elements are sequential within a data block.



18.5



A Chain of Heaps?

The Huge Array routines take special steps to link LMem heaps in a VM chain. Applications can't do this directly.

system-defined threshold, the Huge Array routine automatically divides the data block.

When the VM routines resize an element block, they automatically make the block larger than necessary. This leaves extra space for future insertions in that block, so the block won't have to be resized the next time an element is added. This improves efficiency, since you may often be adding several elements to the same block. However, this also means that most Huge Arrays have a fair amount of unused space. If you won't be adding elements to a Huge Array for a while, you should compact the Huge Array with **HugeArrayCompressBlocks** (see section 18.5.3 on page 713).

Ordinarily, VM Chains may not contain LMem heaps. Huge Arrays are an exception. The reason LMem blocks cannot belong to VM chains is simple. Each block in a VM chain begins with the handle of the next block in the chain (or VM_CHAIN_TREE if it is a tree block). However, each LMem heap has to begin with an **LMemBlockHeader** structure, the first word of which is the global handle of the memory block. In order for these blocks to serve as both, special actions have to be taken. When a Huge Array block is unlocked, its first word is the handle of the next block in the chain. It is thus a VM chain block and not a valid LMem heap. When the Huge Array routine needs to access the block, it locks the block and copies the block's global handle into the first word, storing the chain link in another word. This makes the block a valid LMem heap, but it (temporarily) invalidates the VM chain.

Although VM chain utilities work on Huge Arrays, you must be sure that the Huge Array is a valid VM chain when you call the utility. In practice, this means you cannot use a VM chain utility when any block in the chain is locked or while any thread might be accessing the array. If more than one thread might be using the array, you should not use the chain utilities.

18.5.2 Basic Huge Array Routines

`HugeArrayCreate()`, `HugeArrayDestroy()`, `HugeArrayLock()`,
`HugeArrayUnlock()`, `HugeArrayDirty()`, `HugeArrayAppend()`,

```
Huge ArrayInsert(), HugeArrayReplace(), HugeArrayDelete(),
HugeArrayGetCount()
```

GEOS provides many routines for dealing with Huge Arrays. The basic routines are described in this section. Some additional routines which can help optimize your code are described in “Huge Array Utilities” on page 713.

Note that you should never have more than one block of a Huge Array locked at a time. Furthermore, when you call any routine in this section (except **HugeArrayUnlock()**, **HugeArrayDirty()**, and **HugeArrayGetCount()**), you should not have *any* blocks locked. The next section contains several routines which may be used while a block is locked. Also, if you use any VM chain routines on a Huge Array, you should make sure that no blocks are locked.

18.5



Locked Blocks in Huge Arrays

You should never have more than one Huge Array block locked; for many routines, you must have no blocks locked.

To create a Huge Array, call **HugeArrayCreate()**. This routine allocates a directory block and initializes it. The routine takes three arguments:

- ◆ The handle of the VM file in which to create the huge array. This argument is ignored if a default VM file has been set for this thread.
- ◆ The size of each element. A size of zero indicates that arguments will be variable-sized.
- ◆ The size to allocate for the directory block's header. If you want to have a fixed data area between the **HugeArrayDirectory** structure and the chunk array of directory entries, you can pass an argument here. The size must be at least as large as **sizeof(HugeArrayDirectory)**. Alternatively, you can pass an argument of zero, indicating that there will be no extra data area, and the default header size should be used.

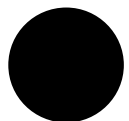
HugeArrayCreate() returns the VM handle of the directory block. This is also the handle of the Huge Array itself; you will pass it as an argument to most of the other Huge Array routines.

When you are done with a Huge Array, destroy it by calling **HugeArrayDestroy()**. This routine frees all of the blocks in the Huge Array. It takes two arguments, namely the global handle of the VM file and the VM handle of the Huge Array. It does not return anything. You should make sure that none of the data blocks are locked when you call this since all of the VM chain links must be valid when this routine is called.



Unlock All Blocks Before Freeing Array

While data blocks are locked, the VM links are invalid.



To access an element in the array, call **HugeArrayLock()**. This routine takes four arguments:

- ◆ The global handle of the VM file which contains the Huge Array.
- ◆ The VM handle of the Huge Array.
- ◆ The (32-bit) index number of the element.
- ◆ A pointer to a pointer to an element.

18.5

The routine figures out which block has the element specified (as described above). It then locks that block. It writes a pointer to that element in the location passed. It returns a dword. The more significant word is the number of consecutive elements in that block, starting with the pointer returned; the less significant word is the number of elements in that block before (and including) the element specified. For example, suppose you lock element 1,000. Let's assume that this element is in block *x*; block *x* has 50 elements, and element 1,000 in the huge array is the 30th element in block *x*. The routine would write a pointer to element 1,000 in the pointer passed; it would then return the dword 0x0015001e. The upper word (0x0015) indicates that element 1,000 is the first of the last 21 consecutive elements in the block; the lower word (0x001e) indicates that the element is the last of the first 30 consecutive elements. You thus know which other elements in the Huge Array are in this block and can be examined without further calls to **HugeArrayLock()**.

When you are done examining a block in the Huge Array, you should unlock the block with **HugeArrayUnlock()**. This routine takes only one argument, namely a pointer to any element in that block. It does not return anything. Note that you don't have to pass it the same pointer as the one you were given by **HugeArrayLock()**. Thus, you can get a pointer, increment it to work your way through the block, and unlock the block with whatever address you end up with.

Whenever you insert or delete an element, the Huge Array routines automatically mark the relevant blocks as dirty. However, if you change an element, you need to dirty the block yourself or the changes won't be saved to the disk. To do this, call the routine **HugeArrayDirty()**. This routine takes one argument, namely a pointer to an element in a Huge Array. It dirties the data block containing that element. Naturally, if you change several elements in a block, you only need to call this routine once.

If you want to add elements to the end of a Huge Array, call **HugeArrayAppend()**. If elements are of uniform size, you can add up to 2^{16} elements with one call to this routine. You can also pass a pointer to a template element; it will copy the template into each new element it creates. This routine takes four arguments:

- ◆ The global handle of the VM file containing the Huge Array. This argument is ignored if a default file has been set.
- ◆ The VM handle of the Huge Array. 18.5
- ◆ The number of elements to append, if the elements are of uniform size; or the size of the element to append, if elements are of variable size.
- ◆ A pointer to a template element to copy into each new element (pass a null pointer to leave the elements uninitialized).

The routine returns the index of the new element. If several elements were created, it returns the index of the first of the new elements. This index is a dword.

You can also insert one or more elements in the middle of a Huge Array. To do this, call the routine **HugeArrayInsert()**. As with **HugeArrayAppend()**, you can insert many uniform-sized elements at once, and you can pass a pointer to a template to initialize elements with. The routine takes five arguments:

- ◆ The global handle of the VM file containing the Huge Array. This argument is ignored if a default file has been set.
- ◆ The VM handle of the Huge Array.
- ◆ The number of elements to insert, if the elements are of uniform size; or the size of the element to insert, if elements are of variable size.
- ◆ The index of the first of the new elements (the element that currently has that index will follow the inserted elements).
- ◆ A pointer to a template element to copy into each new element (pass a null pointer to leave the elements uninitialized).

It returns the index of the first of the new elements. Ordinarily, this will be the index you passed it; however, if you pass an index which is out of bounds, the new elements will be put at the end of the array, and the index returned will thus be different.

To delete elements in a Huge Array, call **HugeArrayDelete()**. You can delete many elements (whether uniform-sized or variable-sized) with one call to **HugeArrayDelete()**. The routine takes four arguments:

- ◆ The global handle of the VM file containing the Huge Array. This argument is ignored if a default file has been set.
- ◆ The VM handle of the Huge Array.
- ◆ The number of elements to delete.
- ◆ The index of the first element to delete.

You can erase or replace the data in one or more elements with a call to **HugeArrayReplace()**. This is also the only way to resize a variable-sized element. You can pass a pointer to a template to copy into the element or elements, or you can have the element(s) initialized with null bytes. The routine takes five arguments:

- ◆ The global handle of the VM file containing the Huge Array. This argument is ignored if a default file has been set.
- ◆ The VM handle of the Huge Array.
- ◆ The number of elements to replace, if the elements are uniform-sized; or the new size for the element, if elements are variable-sized.
- ◆ The index of the first of the elements to replace.
- ◆ A pointer to a template element to copy into each new element (pass a null pointer to fill the elements with null bytes).

You can find out the number of elements in a Huge Array with a call to **HugeArrayGetCount()**. This routine takes two arguments, namely the file handle and the handle of the Huge Array. The routine returns the number of elements in the array. Since array indices begin at zero, if **HugeArrayGetCount()** returns x , the last element in the array has index $(x - 1)$.

18.5.3 Huge Array Utilities

```
HugeArrayNext(), HugeArrayPrev(), HugeArrayExpand(),  
HugeArrayContract(), HugeArrayEnum(),  
HugeArrayCompressBlocks(), ECCheckHugeArray()
```

The routines in the other section may be all you will need for using Huge Arrays. However, you can improve access time by taking advantage of the structure of a Huge Array. As noted above, you can use any of the VM Chain routines on a Huge Array, as long as none of the blocks are locked. 18.5

If you have been accessing an element in a Huge Array and you want to move on to the next one, you can call the routine **HugeArrayNext()**. The routine takes a pointer to a pointer to the element. It changes that pointer to point to the next element in the array, which may be in a different block. If the routine changes blocks, it will unlock the old block and lock the new one. It returns the number of consecutive elements starting with the element we just advanced to. If we were at the last element in the Huge Array, it unlocks the block, writes a null pointer to the address, and returns zero.

If you want to move to the previous element instead of the next one, call **HugeArrayPrev()**. It also takes a pointer to a pointer to an element. It changes that pointer to a pointer to the previous element; if this means switching blocks, it unlocks the current block and locks the previous one. It returns the number of consecutive elements ending with the one we just arrived at. If the pointer was to the first element in the Huge Array, it unlocks the block, writes a null pointer to the address, and returns zero.

If you have a block of the Huge Array locked and you want to insert an element or elements at an address in that block, call the routine **HugeArrayExpand()**. It takes three arguments:

- ◆ A pointer to a pointer to the location where we you want to insert the elements. This element must be in a locked block.
- ◆ The number of elements to insert, if the elements are uniform-sized; or the size of the element to insert, if elements are variable-sized.
- ◆ A pointer to a template to copy into each new element. If you pass a null pointer, elements will not be initialized.

The routine inserts the elements, dirtying any appropriate blocks. It writes a pointer to the first new element into the pointer passed. Since the data block may be resized or divided to accommodate the request, this address may be different from the one you pass. The routine returns the number of consecutive elements beginning with the one whose address was written. If the new element is in a different block from the address passed, the old block will be unlocked, and the new one will be locked.

18.5

If you have a block of a Huge Array locked and you want to delete one or more elements starting with one within the block, you can call **HugeArrayContract()**. This routine takes two arguments:

- ◆ A pointer to the first element to be deleted. The block with that element must be locked.
- ◆ The number of elements to delete. Not all of these elements need be in the same block as the first.

The routine deletes the elements, dirtying any appropriate blocks. It then changes the pointer to point to the first element after the deleted ones. If this element is in a different block, it will unlock the old block and lock the new one. It returns the number of consecutive elements following the one whose address was written.

You may wish to perform the same operation on a number of consecutive elements of a Huge Array. **HugeArrayEnum()** is a routine which lets you do this efficiently. **HugeArrayEnum()** takes six arguments:

- ◆ The VMFileHandle of the VM file containing the Huge Array.
- ◆ The VMBlockHandle of the Huge Array.
- ◆ The index of the first element to be enumerated (remember, the first element has index zero).
- ◆ The number of elements to enumerate, or -1 to enumerate to the end of the array.
- ◆ A void pointer; this pointer will be passed to the callback routine.
- ◆ A pointer to a Boolean callback routine. This callback routine should take two arguments: a void pointer to an element, and the void pointer which was passed to **HugeArrayEnum()**. The callback routine can abort the enumeration by returning *true* (i.e. non-zero).

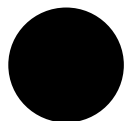
HugeArrayEnum() calls the callback routine for every element, in order, from the first element. It passes the callback a pointer to the element and the pointer passed to **HugeArrayEnum()**. The callback routine may not do anything which would invalidate any pointers to the Huge Array; for example, it may not allocate, delete, or resize any of the elements. The callback routine should restrict itself to examining elements and altering them (*without* resizing them). The callback routine can abort the enumeration by returning *true* (i.e. non-zero); if it does so, **HugeArrayEnum()** will return *true*. If **HugeArrayEnum()** finishes the enumeration without aborting, it returns *false* (i.e. zero).

18.5

If the count is large enough to take **HugeArrayEnum()** past the end of the array, **HugeArrayEnum()** will simply enumerate up to the last element, then stop. For example, if you pass a start index of 9,000 and a count of 2,000, but the Huge Array has only 10,000 elements, **HugeArrayEnum()** will enumerate up through the last element (with index 9,999) then stop. However, the starting index *must* be the index of an element in the Huge Array. You can also pass a count of -1, indicating that **HugeArrayEnum()** should enumerate through the last element of the array. Therefore, to enumerate the entire array, pass a starting element of zero and a count of -1.

As noted above, most Huge Arrays contain a fair amount of unused space. This makes it much faster to add and remove elements, since blocks don't need to be resized very often. However, if you have a Huge Array that is not frequently changed, this is an inefficient use of space. You can free this space by calling **HugeArrayCompressBlocks()**. This routine is passed two arguments: the handle of the VM file, and the **VMBlockHandle** of the Huge Array. The routine does not change any element in the Huge Array; it simply resizes the directory and data blocks to be no larger than necessary to hold their elements. The routine does not return anything.

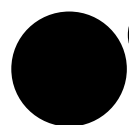
If you want to verify (in error-checking code) that a given VM block is the directory block of a Huge Array, you can call **ECCheckHugeArray()**. This routine is passed the VM file and block handles of the block in question. If the block is the directory block of a Huge Array, the routine returns normally; otherwise it causes a fatal error. The routine should not, therefore, be used in non-EC code.



Virtual Memory

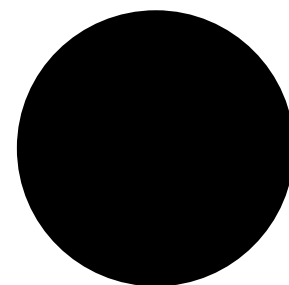
716

18.5



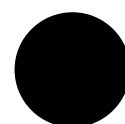
Concepts book

Database Library



19

19.1	Design Philosophy	719
19.2	Database Structure	720
19.2.1	DB Items.....	720
19.2.2	DB Groups	721
19.2.3	Allocating Groups and Items	722
19.2.4	Ungrouped DB Items.....	723
19.2.5	The DB Map Item.....	723
19.3	Using Database Routines	724
19.3.1	General Rules to Follow.....	724
19.3.2	Allocating and Freeing Groups	725
19.3.3	Allocating and Freeing Items	725
19.3.4	Accessing DB Items.....	726
19.3.5	Resizing DB Items	727
19.3.6	Setting and Using the Map Item.....	728
19.3.7	Routines for Ungrouped Items	729
19.3.8	Other DB Utilities.....	730
19.4	The Cell Library.....	731
19.4.1	Structure and Design.....	732
19.4.2	Using the Cell Library.....	734
19.4.2.1	The CellFunctionParameters Structure	734
19.4.2.2	Basic Cell Array Routines	736
19.4.2.3	Actions on a Range of Cells	738





Some applications need to keep track of many small pieces of data. For example, a database might use thousands of items of data, each of them only a paragraph long; in a spreadsheet, the data size might be only a few bytes. GEOS provides a Database (DB) library to make it easy to keep track of such data and store them conveniently in a GEOS Virtual Memory file.

The DB library manages Local Memory heaps in a VM file and uses these heaps to store items. It lets the geode associate items into groups; these groups can grow indefinitely, unlike LMem heaps.

19.1

Before you read this chapter, you should have read “Handles,” Chapter 14, and “Memory Management,” Chapter 15. You should also be familiar with basic LMem principles (see “Local Memory,” Chapter 16) and with Virtual Memory files (see “Virtual Memory,” Chapter 18).



19.1 Design Philosophy

A database manager should be flexible, allowing applications to store a variety of data items. It should be efficient, with minimal overhead in data-access time as well as in memory usage (whether in main memory or in disk space). Ideally, it ought to insulate applications from the details of memory allocation and data referencing. The GEOS database manager meets all of these requirements and several more:

- ◆ **Flexible Data Formats**

The DB Manager does not care about the content of a DB item. Consequently, a DB item can be anything that can fit in an LMem chunk. An application can use one file to store many different sizes of database item.

- ◆ **Speed and Efficiency**

The GEOS DB library uses the powerful GEOS memory management, Virtual Memory, and Local Memory routines. These enable it to store and access many database items with a minimal overhead in access time and storage space.

- ◆ **Uniform Data-Access Format**
Database items are stored in standard GEOS Virtual Memory files. All of the file-access utilities (such as the document control objects) can work unchanged with database files. Furthermore, VM files can contain both ordinary VM blocks and DB items in any combination.
- ◆ **Full Group Management**
Applications can divide DB items into groups. Access time is improved when items from the same group are accessed in succession. As an alternative, applications can let the DB manager create and assign groups for the items.
- ◆ **Sharable Data**
Since DB items are stored in VM files, the files can be shared between applications. All of the standard VM synchronization routines work for DB files.

19.2 Database Structure

The Database Library uses a Database Manager to access and create DB items. These items are stored in a standard VM file. This chapter will sometimes refer to a “Database File”; this simply means a VM file which contains DB items.

19.2.1 DB Items



Important

DB group- and item-handles are different from LMem heap- and chunk-handles.

The basic unit of data is the *item*. Items are simply chunks in special LMem heaps which are managed by the DB Manager; these heaps are called *item blocks*. You will not need to use any of the LMem routines; the DB manager will create and destroy LMem heaps as necessary and will call the appropriate routines to lock DB items when needed.

Each DB item in a DB file is uniquely identified by the combination of a *group-handle* and an *item-handle*. Note that these handles are not the same as the item's LMem Heap handle and its chunk handle. You will not generally need to use the item's heap and chunk handles; the DB routines store and use these automatically. However, you can retrieve them if necessary (for example, if you want to use an LMem utility on a DB item).

The DB Manager does not keep track of allocated items. Once you allocate an item, you must store the group- and item-handles. If you lose them, the item will remain in the file, but you will not be able to find it again.

Since DB items are chunks, their addresses are somewhat volatile. If you allocate an item in a group, other items in that group may move even if they are locked. (See section 16.2.2 of chapter 16.)

19.2.2 DB Groups

19.2

Each DB item is a member of a DB group. The DB group is a collection of VM blocks; the group comprises a single *group block* and zero or more item blocks.

The group block contains information about each item block and each item in the group. For each item block, it records the VM handle of the block and the number of DB items in the block. For each DB item, it records the VM handle of the item block in which the item is stored and the item's chunk handle within that item block. The item blocks are simply LMem heaps with a little extra information in the headers.

The item's group-handle is simply the VM handle of the group block for that group. The item's item-handle is an offset into the group block; the information about the item is stored at that offset. When you lock an item, the DB manager looks in that location in the group block and reads the handles of the item block and the chunk associated with that item; it then locks the item block and returns the address of the chunk. (In assembly code, it returns the segment address and the chunk handle.) The relationship between the different blocks and handles is shown in Figure 19-1 on page ● 722.

Whenever you access a DB item, the DB manager has to lock the block. If you access several items in a row, the overall access time is better if they belong to the same group since only one group block will need to be swapped in to memory. The items may also be in the same item-block since each item block contains items from only one group; again, this improves access time. Thus, it is a good idea to distribute items in groups according to the way they will be accessed; for example, an address-book database might group entries according to the first letter of the last name, thus speeding up alphabetical

access. If you have no logical way to group items, see “Ungrouped DB Items” on page 723.

19.2.3 Allocating Groups and Items

When you need a new DB group, call the DB routine **DBGroupAlloc()** (see page 725). This routine creates and initializes a DB group block.

When you allocate a DB item, you specify which group the item will go in. The DB manager sets up an entry for the item in the group block. It then decides which item block to put the item in. It tries to keep all the item blocks at the right size to optimize speed. If all of the group's item blocks are too full, it allocates a new item block and allocates the new item in that block. In either case, it returns the new item's item-handle.

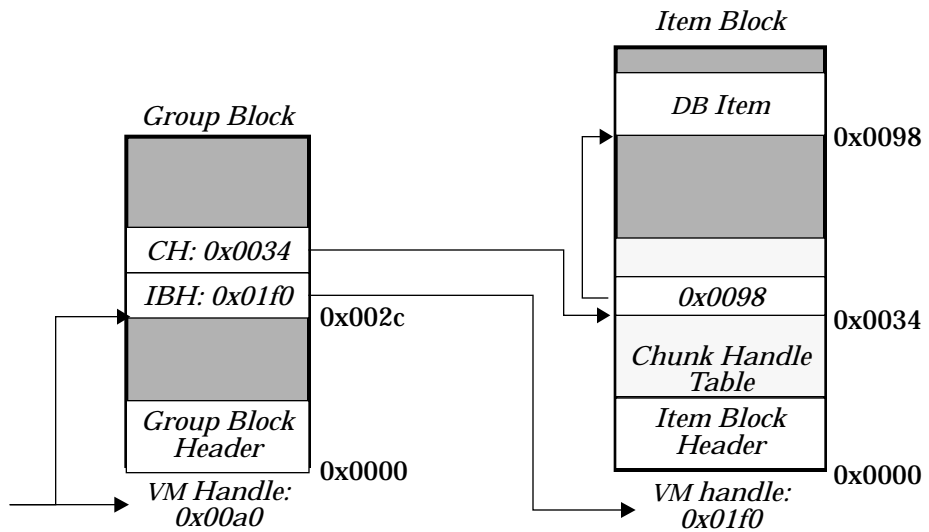


Figure 19-1 Dereferencing a DB Item
A DB item's group- and item-handles indicate an entry in the group block which has information on that item's item block handle and chunk handle. Here, we dereference the DB item with group-handle 0x00a0 and item-handle 0x002c. The blocks are shown with their VM handles (not their global handles). Addresses next to the blocks indicate offsets into the blocks. Note that the mechanics are transparent to the application, which simply passes the DB group- & item-handles to the DB routine and is returned the address of the DB item.

Once an item has been allocated, it will stay in the same item block (and have the same chunk handle) until it is freed or resized. If it is resized to a larger size, it may be moved to a different item block belonging to the same group.

19.2.4 Ungrouped DB Items

Sometimes there is no natural way to group DB items. For these situations, the DB manager allows you to allocate *ungrouped* items. These items actually belong to special groups which are automatically allocated by the DB manager. The DB manager tries to keep these groups at the right size for optimum efficiency.

19.2



“Ungrouped” Items

If you allocate an “ungrouped” item, the DB manager will assign it to a group. You will need that group-handle to access the item.

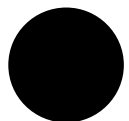
When you allocate an ungrouped item, the DB manager allocates an item in one of its “ungrouped groups.” If there are no such groups or if all of these groups have too many items already, the DB manager allocates a new “ungrouped” group.

For practical purposes, ungrouped DB items have a single, dword-sized handle. This “handle” is of type **DBGroupAndItem**. The upper word of this is the handle of the ungrouped group for this item; the lower word is the item’s item-handle within that group. There are special versions of most database routines for use with ungrouped items. These routines take a **DBGroupAndItem** argument instead of separate group-handle and item-handle arguments. These routines are discussed in “Routines for Ungrouped Items” on page 729. This section also describes macros which combine a group-handle and item-handle into a **DBGroupAndItem** and which break a **DBGroupAndItem** into its constituent parts.

19.2.5 The DB Map Item

You can designate a “map item” for a VM file with the routine **DBSetMap0**. You can recover the map item’s group and handle at will by calling **DBGetMap0**. This is entirely separate from the file’s map block; indeed, a VM file can have both a map block and a map item, and they may be set, locked, and changed independently.

The map routines are described in detail in section 19.3.6 on page 728.



19.3 Using Database Routines

GEOS provides a wide range of routines for working with databases. The routines all require that the calling thread have the VM file open. Most routines have to be passed the **VMFileHandle** of the appropriate VM file.

Almost all DB routines come in two forms. The standard form takes, among its arguments, the group-handle and the item-handle of an item to be affected. The other form is designed for use with “ungrouped” items. This form takes, as an argument, the item’s **DBGroupAndItem** structure.

In addition to the routines listed here, all of the VM chain routines can work on DB items. Simply cast the **DBGroupAndItem** structure to type **VMChain**, and pass it in place of the chain argument(s).

(**VMCopyVMChain()** will allocate the duplicate item as “ungrouped.”) For more information about **VMChain** routines, see section 18.4 of chapter 18.

19.3.1 General Rules to Follow

There are certain rules of “memory etiquette” you should follow when using DB files. For the most part, these rules are the same as the general rules of memory etiquette.

First and foremost, try to keep as few blocks locked as possible, and keep them locked for as short a time as possible. You should not usually need to keep more than one item locked at a time. If you need another item, unlock the first one first, even if they’re in the same item block. (This will cost very little time since the item block is unlikely to be swapped to disk right away.) The main reason you should have two or more items open at once is if you are directly comparing them or copying data from one to another. In this case, you should unlock each item as soon as you’re done with it.



Invalidating Pointers to Items

Allocating or expanding an item can invalidate all pointers to items in that group. Try to unlock items before doing an allocation or resizing.

Remember that items are implemented as chunks in LMem heaps. This means, for example, that when you allocate an item (or expand an existing one), the heap it resides in (i.e. the item block) may be compacted or moved on the global heap (even if it is locked). This will invalidate all pointers to items in that item block. As a general rule, you should not allocate (or expand) items if you have any items from that group locked. Do not allocate “ungrouped” items if you have any items from any of the “ungrouped” groups

locked. If you must keep an item locked, keep track of the item's memory block and chunk handle so you can use **DBDeref()** to get the address again.

Finally, try to keep the blocks small. Most of this is done for you. When you allocate an item, the DB manager will put it in an uncrowded item block. If all item blocks are too large, it will allocate a new one. However, you should keep items from getting too large. If individual items get into the multi-kilobyte range, you should consider storing them a different way; for example, you could make each of the larger items a VM block or a VM chain.

19.3

19.3.2 Allocating and Freeing Groups

`DBGGroupAlloc()`, `DBGGroupFree()`

You can improve DB access time by assigning items to groups such that items from the same group will generally be accessed together. This will cut down on the number of times group and item blocks will have to be swapped into memory.

To allocate a group, call **DBGGroupAlloc()**. This routine takes one argument, namely the handle of the VM file in which to create the group. It allocates the group and returns the group-handle (i.e., the VM handle of the group block). If it is unable to allocate the group, it will return a null handle.

If you are done with a DB group, call **DBGGroupFree()**. This routine frees the group's group block and all of its item blocks. Any attached global memory blocks will also be freed. Naturally, all items in the group will be freed as well. You can free a group even if some of its items are locked; those items will be freed immediately.

19.3.3 Allocating and Freeing Items

`DBAlloc()`, `DBFree()`

To allocate a DB item, call **DBAlloc()**. This routine takes three arguments: the handle of the VM file, the DB Group in which to allocate the item, and the size of the item (in bytes). The routine will allocate an item in one of that group's item blocks (allocating a new item block if necessary); it returns the new item's item-handle.

Remember that when you allocate a DB item, the DB manager allocates a chunk in an LMem heap (the item block). This can cause the item block to be compacted or resized; this will invalidate all pointers to items in that block. For this reason, you should not allocate items in a group while other items in that group are locked. Similarly, you should not allocate “ungrouped” items while any “ungrouped” items are locked. Instead, unlock the items, allocate the new one, and then lock the items again.

19.3



Never free a locked item

The item-block isn't unlocked until the last item is unlocked.

When you are done with an item, free it with **DBFree()**. This routine takes three arguments: the file handle, the group-handle, and the item-handle. It frees the item, making appropriate changes in the group block. If the item was the only one in its item block, that item block will be freed as well.

DBFree() does not return anything. Note that you should never free a locked item since the item-block's reference-count will not be decremented (and the block will never be unlocked). Always unlock an item before freeing it. (You need not, however, unlock items before freeing their *group*; when a group is freed, all of its items are automatically freed, whether they are locked or not.)

19.3.4 Accessing DB Items

`DBLock()`, `DBLockGetRef()`, `DBDeref()`, `DBUnlock()`, `DBDirty()`

To access a database item, lock it with **DBLock()**. This routine takes three arguments: the handle of the VM file, the item's group-handle, and the item's item-handle. The routine locks the item-block on the global heap and returns the address of the element. If the block is already locked (generally because another item in the item-block is locked), it increments the lock count.

In some circumstances it might be useful to know the global handle of the locked item-block and the chunk handle of the item. For example, if you want to set up an item as a chunk array, you will need this information. For this reason, the library provides the routine **DBLockGetRef()**. This routine is just like **DBLock()**, except that it takes one additional argument: the address of a variable of type `optr`. **DBLockItemGetRef()** writes global and chunk handles to the `optr` and returns the address of the locked DB item. You can now use any of the LMem routines on the item, simply by passing the `optr`.

Note that the memory block attached to the item block may change each time the block is locked unless you have instructed the VM manager to preserve the handle (see section 18.3.6 of chapter 18). The chunk handle will not change, even if the file is closed and reopened, unless the chunk is resized larger. (When an item is resized larger, the DB manager may choose to move the item to a different item-block, thus changing its chunk handle.) In general, if you will need this information you should get it each time you lock the item instead of trying to preserve it from one lock to the next.

19.3

If you have an *optr* to a *locked* DB item, you can translate it to an address with the routine **DBDeref()**. This is useful if you have to keep one item locked while allocating or expanding another item in that group. Since the locked item might move as a result of the allocation, you can get the new address with **DBDeref()**. In general, however, you should unlock all items in a group before allocating or resizing one there. Note that **DBDeref** is simply a synonym for **LMemDeref()**; the two routines are exactly the same.

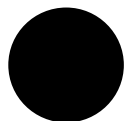
When you are done accessing an item, call **DBUnlock()**. This routine takes one argument, the address of a locked item. The routine decrements the reference count of the item's item-block. If the reference count reaches zero, the routine unlocks the block. Thus, if you lock two different items in an item block, you should unlock each item separately. As noted above, you should always unlock an item before freeing it.

If you change a DB item, you should mark the item's block as *dirty* by calling **DBDirty()**. This ensures that the changes will be copied to the disk the next time the file is saved or updated. The routine takes one argument, a pointer to an address in an item block (generally the address of an item). It will dirty the item-block containing that item. As with VM blocks, you must dirty the item *before* you unlock it, as the memory manager can discard any clean block from memory as soon as it is unlocked.

19.3.5 Resizing DB Items

`DBReAlloc(), DBInsertAt(), DBDeleteAt()`

Database items may be resized after allocation. They may be expanded either by having bytes added to the end or by having bytes inserted at a specified offset within the item. Similarly, items may be contracted by having bytes



truncated or by having bytes deleted from the middle of the item. When an item is resized, the DB manager automatically dirties the item block (or blocks) affected.

As noted above, when an item is expanded, its item block can be compacted or moved on the item heap (even if the item is locked). Thus, pointers to all items in that item block may be invalidated, even if they are locked. For that reason, you should unlock all items in the group before expanding any of them. If you must leave an item locked, be sure to get its new address with **DBDeref()**. If you *decrease* an item's size, the item-block is guaranteed not to move or be compacted. Thus, you can safely contract locked items (or items in the same block as locked items).

To set a new size for an item, call **DBReAlloc()**. This routine takes four arguments: the file handle, the group-handle, the item-handle, and the new size (in bytes). If the new size is smaller than the old, bytes will be truncated from the end of the item. If the new size is larger than the old, bytes will be added to the end of the item; these bytes will not be zero-initialized.

To insert bytes in the middle of an item, call the routine **DBInsertAt()**. This routine takes five arguments: the file handle, the group-handle, the item-handle, the offset (within the item) at which to insert the bytes, and the number of bytes to insert. The new bytes will be inserted beginning at that offset; they will be zero-initialized. Thus, if you insert ten bytes beginning at offset 35, the new bytes will be at offsets 35-44; the byte which had been at offset 35 will be moved to offset 45. To insert bytes at the beginning of an item, pass an offset of zero.

To delete bytes from the middle of an item, call **DBDeleteAt()**. This routine takes five arguments: the file handle, the group-handle, the item-handle, the offset (within the item) of the first byte to delete, and the number of bytes to delete. The routine does not return anything.

19.3.6 Setting and Using the Map Item

`DBSetMap()`, `DBGetMap()`, `DBLockMap()`

A VM file can have a *map block* and a *map item*. The map can be retrieved with a special-purpose routine, even if you don't know its handle (or handles); thus, the map usually keeps track of the handles for the rest of the file. The

map can be retrieved even if the file is closed and re-opened. To set a map block, use the routine **VMSetMap()** (see section 18.3.10 of chapter 18). To set a *map item*, use the routine **DBSetMap()**. **DBSetMap()** takes three arguments: the file handle, the item's group-handle, and the item's item-handle. The routine sets the file's map item to the DB item specified. A VM file can have both a map block and a map item; these are set independently.

Once you have set a map item, you can retrieve its handles with the command **DBGetMap()**. This routine takes one argument, namely the file's handle. It returns a **DBGroupAndItem** value containing the map item's handles. You can break this value into its constituent handles with **DBGroupFromGroupAndItem()** and **DBItemFromGroupAndItem()** (see section 19.3.7 on page 729). You can also lock the map directly without knowing its handles by calling the routine **DBLockMap()**. This routine takes one argument, namely the file handle. It locks the map item and returns the map's address. When you are done with the map item, unlock it normally with a call to **DBUnlock()**.

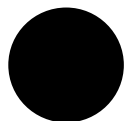
19.3

19.3.7 Routines for Ungrouped Items

```
DBAllocUngrouped(), DBFreeUngrouped(), DBLockUngrouped(),
DBLockGetRefUngrouped(), DBReAllocUngrouped(),
DBInsertAtUngrouped(), DBDeleteAtUngrouped(),
DBSetMapUngrouped()
```

Special routines are provided for working with ungrouped items. These routines are very similar to their standard counterparts. The routine **DBAllocUngrouped()** allocates an ungrouped item. It takes two arguments, the file handle and the size of the item to allocate. The DB manager allocates the item in one of the “ungrouped” groups and returns a **DBGroupAndItem** value containing the group-handle and item-handle. You can break this value into its components by calling the macros described in section 19.3.7 on page 729, or you can pass this value directly to the other “ungrouped” routines.

The rest of the routines listed above are exactly the same as their counterparts with one exception: whereas their counterparts take, among their arguments, the item's group-handle and item-handle, the ungrouped



routines take a **DBGroupAndItem** value. Each routine's other arguments are unchanged, as is the return value.

These routines are provided as a convenience. If you allocate an ungrouped item, you are perfectly free to break the **DBGroupAndItem** value into its component handles, and pass those handles to the standard DB routines. Conversely, if you allocate a normal “grouped” item, you are free to combine the two handles into a **DBGroupAndItem** token and pass that token to the “ungrouped” routines.

19.3

19.3.8 Other DB Utilities

```
DBCopDBItem(), DBCopDBItemUngrouped(),  
DBGroupFromGroupAndItem(), DBItemFromGroupAndItem(),  
DBCombineGroupAndItem()
```

You can duplicate a DB item with the routine **DBCopDBItem()**. This routine takes five arguments: the handle of the source file, the source item's group-handle, the source item's item-handle, the handle of the destination file (which may be the same as the source file), and the handle of the destination group. The routine will allocate a new item in the specified file and group. It will then lock both items and copy the data from the source item to the destination. Finally, it will unlock both items and return the item-handle of the duplicate item.

The routine **DBCopDBItemUngrouped()** is the same as **DBCopDBItem()**, except that it allocates an ungrouped item in the specified file. It is passed the source file handle, the **DBGroupAndItem** value for the source item, and the destination file handle. It allocates an ungrouped item and returns its **DBGroupAndItem** value.

Remember, if you are allocating the duplicate in the same group as the source, you should only call this routine when the source item is unlocked (since its item-block may be compacted when the new item is allocated). If the destination is in another block, the source item may be locked or unlocked at your preference. If it is locked when you call **DBCopDBItem()**, it will be locked when the routine returns.

All of the VM chain utilities work on DB items as well as VM chains. The routines are described in section 18.4 of chapter 18. To use a VM chain

routine, pass the item's **DBGroupAndItem** value. For example, **VMCopyVMChain()** will allocate an “ungrouped” duplicate item in the specified file and return its **DBGroupAndItem** value.

To build a **DBGroupAndItem** value from the group-handle and item-handle, use the macro **DBCombineGroupAndItem()**. This macro takes the two handles and returns a **DBGroupAndItem** value. To extract the component handles from a **DBGroupAndItem** value, use the macros **DBGroupFromGroupAndItem()** and **DBItemFromGroupAndItem()**. These macros are passed a **DBGroupAndItem** value and return the appropriate component.

19.4

19.4 The Cell Library

The database library lets applications organize data into groups. This is an intuitive way to organize data for many applications. However, for some applications, it is more natural to organize data in a two-dimensional array. The classic example of this is the spreadsheet, in which every entry (or *cell*) can be uniquely identified by two integers: the cell's row and its column.

The GEOS cell library lets you arrange data this way. With the cell library, you can organize data in rows and columns. The cell library saves the cells as DB items in a VM file. It insulates the application from the actual DB mechanism, letting the application behave as if its data is stored in a two-dimensional array. However, since the data is stored in DB items, it may be kept in any ordinary VM file. The library also provides routines to sort the cells by row or by column and to apply a routine to every cell in a range of rows and/or columns.

A collection of cells arranged into rows and columns is termed a *cell file*. Every cell file is contained in a VM file. There is often a one-to-one correspondence between cell files and the VM files which contain them. However, this correspondence is optional. There is nothing to stop an application from maintaining several distinct cell files in a single VM file.

19.4.1 Structure and Design

19.4

Most of the internal structure of a cell file is transparent to the geode which uses it. A geode can, for example, lock a cell with **CellLock()**, specifying the cell's row and column. The cell library will find the appropriate DB item and lock it, returning the locked item's address. For most operations, the geode does not need to know anything about the internal structure of the cell file. However, the internal structure does matter for some purposes. For this reason, we present a quick overview of the structure of a cell file.

A cell file can contain up to 16,512 rows, numbered from zero to 16,511. This is less than 2^{15} , so a row index can fit in a signed-word variable. Of these 16,512 rows, the last 128 are "scratch-pad" rows. They are intended to be used for holding information or formulae that will not be displayed or associated with a specific cell. The scratch-pad rows are never shifted; if you create a cell in the first scratch-pad row, it will always stay in that row. All other rows are called "visible" rows. Visible rows can be shifted when rows are created or deleted. For example, if you insert a new row 10, all the cells in the old row 10 will now be in row 11, and so on. The first scratch-pad row will be unchanged. Be aware that the database will not delete cells from rows that are shifted off the spreadsheet. For example, if you insert a new row, the last visible row will be shifted off the spreadsheet; the references to cells in that row will be removed, but the cells themselves will stay as DB items in the file. This is not generally a problem, since few cell files will need to use the last visible rows. If you add a row that will cause cells to be shifted off, you should delete those cells first.

The first row has an index number of zero. (See Figure 19-2.) The last visible row has an index equal to the preprocessor constant

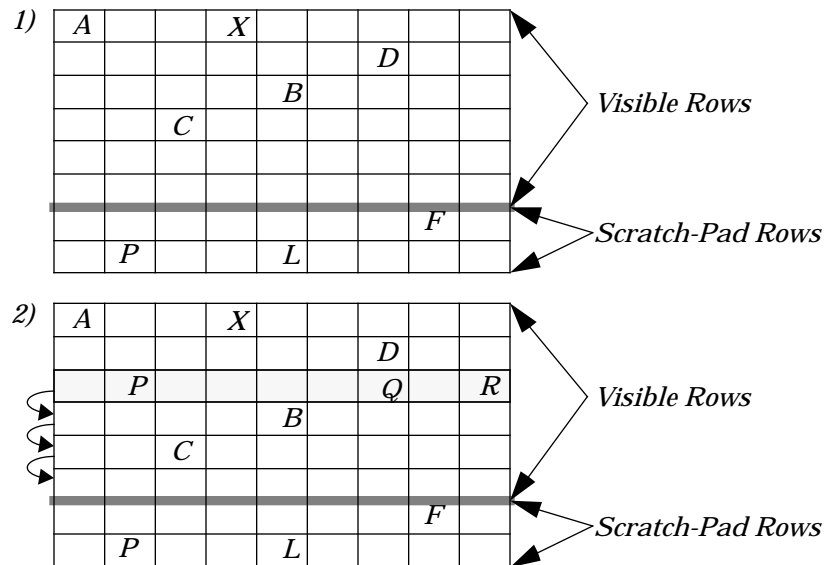
`LARGEST_VISIBLE_ROW`. The first scratch-pad row has an index equal to `(LARGEST_VISIBLE_ROW + 1)`. The last scratch-pad row has an index equal to `LARGEST_ROW` (which equals `(LARGEST_VISIBLE_ROW + 128)` or 16,511). The constants are all defined in **cell.h**.



Cells are Ungrouped DB Items

When you create or resize a cell, you invalidate pointers to all ungrouped items in that VM file.

The basic data unit in a cell file is the *cell*. The cell library treats cells as opaque data structures; their internal structure is entirely up to the geode using them. Cells are stored as ungrouped DB items. This restricts cell size to the size of a DB item; that is, a cell can theoretically be as large as 64K, but in practice should not grow larger than around 8K (and ideally should be under a kilobyte in size). Remember, whenever a DB item is created or



19.4

Figure 19-2 Inserting a New Row

1) A cell file before a row is inserted. This is an abstract representation; in fact the rows contain references to cells, not the cells themselves. Real cell files, of course, have many more visible and scratch-pad rows.

2) A new row is inserted at position 2. This shifts all the following visible rows down; it does not affect the scratch-pad rows.

resized, pointers to all other items in the group are invalidated. Since cells are ungrouped items, whenever you create or resize a cell, you invalidate any pointers to all other ungrouped items in that VM file. In particular, you invalidate pointers to all other cells in that VM file (even if the cells belong to another cell file in the VM file).

Cells are grouped into *rows*. A row can have up to 256 cells, numbered from zero to 255. Within a row, cells are identified by their column index. The column index can fit into an *unsigned* byte variable. The cell library creates a *column array* for every row which contains cells. The column array contains one entry for each cell in the row. A row often contains just a few widely scattered elements. For this reason, the column array is implemented as a *sparse array*. Each cell in the row has an entry consisting of two parts, namely the cell's column number and its **DBGGroupAndItem** structure. The advantage of this arrangement is that the column array need only contain



entries for those cells which exist in the row (instead of maintaining entries for the blank spaces between cells). The disadvantage is that when you lock a cell, the cell manager has to make a search through the column array to find its reference; however, this is generally a small cost.

The column arrays themselves belong to *row blocks*. Each row block is an LMem heap stored in the VM file, and each of its column arrays is a chunk in that heap. Row blocks contain up to 32 rows. These rows are sequential; that is, any existing rows from row zero to row 31 will always belong to the same row block, and none of them will ever be in the same row block as row 32. Since the row blocks and column arrays are not kept in DB items, they can be accessed and altered without causing any locked items to move. To keep track of the row blocks, you must have a **CellFunctionParameters** structure for each cell file. That structure need not be kept in the VM file (although it often is); rather, you must pass the address of the structure to any cell library routine you call.

Owing to the structure of a cell file, some actions are faster than others. The essential thing to remember is that cells are grouped together in rows, which are themselves grouped together to form a cell file. This means that you can access several cells belonging to the same row faster than you could access cells belonging to different rows. Similarly, if you insert a cell, it is much more efficient to shift the rest of the row over (which involves accessing only that one row) than to shift the rest of the column down (which involves accessing every visible row). Similarly, you can access groups of cells faster if they belong to the same row block.

19.4.2 Using the Cell Library

The cell library is versatile. The basic cell access routines are very simple, but more advanced utilities give you a wide range of actions. This section will explain the techniques used to set up and use a cell file, as well as the more advanced techniques available.

19.4.2.1 The CellFunctionParameters Structure

The cell library needs to have certain information about any cell file on which it acts; for example, it needs to know the handles of the VM file and of the row

blocks. That information is kept in a **CellFunctionParameters** structure. The geode which uses a cell file is responsible for creating a **CellFunctionParameters** structure. The C definition of the structure is shown below.

Code Display 19-1 CellFunctionParameters

```
typedef struct {
    CellFunctionParameterFlags
        CFP_flags;          /* Initialize this to zero. */
    VMFileHandle    CFP_file; /* The handle of the VM file containing
                               * the cell file. Reinitialize this each
                               * time you open the file. */
    VMBlockHandle   CFP_rowBlocks[N_ROW_BLOCKS]; /* Initialize these to zero.
*/
} CellFunctionParameters;
```

19.4

In order to create a cell file, you must create a **CellFunctionParameters** structure. Simply allocate the space for the structure and initialize the data fields. When you call a cell library routine, lock the structure on the global heap and pass its address. Geodes will usually allocate a VM block in the same file as the cell file, and use this block to hold the **CellFunctionParameters** structure; this ensures that the structure will be saved along with the cell file. They may often declare this to be the map block, making it easy to locate (see section 18.3.10 of chapter 18). However, this is entirely at the programmer's discretion. All that the cell library requires is that the structure be locked or fixed in memory every time a cell library routine is called.

The **CellFunctionParameters** structure contains the following fields:

CFP_flags The cell library uses this byte for miscellaneous bookkeeping. When you create the structure, initialize this field to zero. There is only one flag which you should check or change; that is the flag CFPF_DIRTY. The cell library routines set this bit whenever they change the **CellFunctionParameters** structure, thus indicating that the structure ought to be resaved. After you save it, you may clear this bit.

CFP_file This field must contain the file handle of the VM file containing the cell file. A VM file can have a new file handle every time it is opened; thus, you must reinitialize this field every time you open the file.

CFP_rowBlocks This field is an array of VM block handles, one for every existing or potential row block. If a row block exists in the cell file, its handle is recorded here. If it does not exist, a null handle is kept in the appropriate place. The length of this array is a number of words equal to the constant `N_ROW_BLOCKS` (defined in **cell.h**). When you create a cell file, initialize all of these handles to zero; do not access or change this field thereafter.

One warning: The cell library expects the **CellFunctionParameters** structure to remain motionless for the duration of a call. Therefore, if you allocate it as a DB item in the same VM file as the cell file, you must *not* have the structure be an ungrouped item. Remember, all the cells are ungrouped DB items; allocating or resizing a cell can potentially move any or all of the ungrouped DB items in that file.

19.4.2.2 Basic Cell Array Routines

`CellReplace()`, `CellLock()`, `CellLockGetRef()`, `CellDirty()`,
`CellGetDBItem()`, `CellGetExtent()`

The basic cell routines are simple to use. One argument taken by all of them is the address of the **CellFunctionParameters** structure. As noted, this structure must be locked or fixed in memory for the duration of a function call. You can also access cells in any of the ways you would access a DB item; for example, you can resize a cell with **DBReAlloc()**.

All of the routines use the VM file handle specified in the **CellFunctionParameters** structure.

To create, replace, or free a cell, call the routine **CellReplace()**. This routine takes five arguments:

- ◆ The address of the **CellFunctionParameters** structure.
- ◆ The element's row.

- ◆ The element's column.
- ◆ The address of the data to copy into the new cell. This must not move during the allocation; therefore, it must not be in an ungrouped DB item in the same VM file as the cell file. In particular, it must not be another cell. The data will not be changed.
- ◆ The size of the new cell. If you pass a size of zero, the cell will be deleted if it already exists; otherwise, nothing will happen.

If the cell file already contains a cell with the specified coordinates, **CellReplace()** will free it. **CellReplace()** will then allocate a new cell and copy the specified data into it. The routine invalidates any existing pointers to ungrouped DB items in the file.

19.4

Once you have created a cell, you can lock it with **CellLock()**. This routine takes three arguments: the address of the **CellFunctionParameters** structure, the cell's row, and the cell's column. It locks the cell and returns its address (the assembly version returns the cell's segment address and chunk handle). Remember, the cell is an ungrouped DB item, so its address may change the next time another ungrouped DB item is allocated or resized, even if the cell is locked.

Like all DB items, cells can (under certain circumstances) be moved even while locked. For this reason, a special locking routine is provided, namely **CellLockGetRef()**. This routine is just like **CellLock()** except that it takes one additional argument, namely the address of an optr. **CellLockGetRef** writes the locked item's global memory handle and chunk handle into the optr. You can translate an optr to a cell into a pointer by calling **CellDeref()**; this is another synonym for **LMemDeref()**, and is identical to it in all respects. For more information, see section 19.3.4 on page 726.

If you change a cell, you must mark it dirty to insure that it will be updated on the disk. To do this, call the routine **CellDirty()**. This routine takes two arguments, namely the address of the **CellFunctionParameters** structure and the address of the (locked) cell. The routine marks the cell's item block as dirty.

Sometimes you may need to get the DB handles for a cell. For example, you may want to use a DB utility to resize the cell; to do this, you need to know its handles. For these situations, call the routine **CellGetDBItem()**. The routine takes three arguments: the address of the

CellFunctionParameters structure, the cell's row, and the cell's column. It returns the cell's **DBGGroupAndItem** value. You can pass this value to any of the **DB...Ungrouped()** routines (described in "Routines for Ungrouped Items" on page 729), or you can break this value into its component handles by calling **DBGGroupFromGroupAndItem()** or **DBItemFromGroupAndItem()**.

19.4

If you want to find out the bounds of an existing cell file, call the routine **CellGetExtent()**. This routine takes two arguments: the address of the **CellFunctionParameters**, and the address of a **RangeEnumParams** structure. For the purposes of this routine, only one of its fields matters, namely the field *REP_bounds*. This field is itself a structure of type **Rectangle**, whose structure is shown below in Code Display 19-2. **CellGetExtent()** writes the bounds of the utilized section of the cell file in the *REP_bounds* field. The index of the first row which contains a cell will be written in the rectangle's *R_top* field; the index of the last row will be written in *R_bottom*; the index of the first column will be written in *R_left*; and the index of the last column will be written in *R_right*. If the cell file contains no cells, all four fields will be set to -1.

Code Display 19-2 Rectangle

```
typedef struct {
    sword      R_left;           /* Index of first column written here. */
    sword      R_top;            /* Index of first row written here. */
    sword      R_right;          /* Index of last column written here. */
    sword      R_bottom;         /* Index of last row written here. */
} Rectangle;
```

19.4.2.3 Actions on a Range of Cells

`RangeExists()`, `RangeInsert()`, `RangeEnum()`, `RangeSort()`,
`RangeInsertParams`

The cell library provides a number of routines which act on a range of cells. All of these routines take the address of a **CellFunctionParameters** structure as an argument. Many of these routines also take the address of a special parameter structure; for example, **RangeInsert()** takes the address

of a **RangeInsertParams** structure. In these cases, the structure should be in locked or fixed memory. If the routine might allocate or resize cells, the structure must not be in an ungrouped DB item.

You may want to find out if there are any cells in a specified section of the cell file. To do this, call the routine **RangeExists()**. This routine takes five arguments:

- ◆ The address of the locked **CellFunctionParameters** structure
- ◆ The index of the first row in the section to be checked
- ◆ The index of the first column in the section to be checked
- ◆ The index of the last row in the section to be checked
- ◆ The index of the last column in the section to be checked

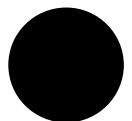
19.4

If any cells exist in that section, the routine returns *true* (i.e. non-zero). Otherwise, it returns *false*.

You may wish to insert several cells at once. For this reason, the cell library provides the routine **RangeInsert()**. This routine does not actually allocate cells; instead, it shifts existing cells to make room for new ones. You specify a section of the cell file to shift. Any cells in that section will be shifted over; the caller specifies whether they should be shifted horizontally or vertically. The routine takes two arguments, namely the address of the **CellFunctionParameters** and the address of a **RangeInsertParams** structure. It does not return anything. The definition of the **RangeInsertParams** structure is shown in Code Display 19-3. The calling geode should allocate it and initialize it before calling **RangeInsert()**.

Code Display 19-3 The RangeInsertParams and Point structures

```
typedef struct {           /* defined in cell.h */
    Rectangle              RIP_bounds;    /* Range of cells to shift */
    Point                  RIP_delta;     /* Specify which way to shift */
    CellFunctionParameters *RIP_cfp;
} RangeInsertParams;
```



```
typedef struct {          /* defined in graphics.h */
    sword                P_x;    /* Distance to shift horizontally */
    sword                P_y;    /* Distance to shift vertically */
} Point;
```

The **RangeInsertParams** structure has three fields. The calling geode should initialize the fields to determine the behavior of **RangeInsert()**:

19.4

RIP_bounds This field specifies which cells should be shifted. The cells currently in this range will be shifted across or down, depending on the value of *RIP_delta*; this shifts more cells, and so on, to the edge of the visible portion of the cell file. The field is a **Rectangle** structure. To insert an entire row (which is much faster than inserting a partial row), set *RIP_bounds.R_left* = 0 and *RIP_bounds.R_right* = LARGEST_COLUMN.

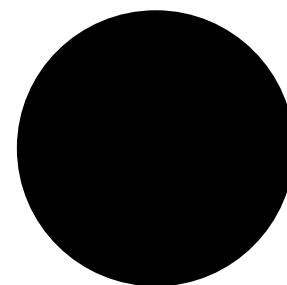
RIP_delta This field specifies how far the cells should be shifted and in which direction. The field is a **Point** structure (see Code Display 19-3). If the range of cells is to be shifted horizontally, *RIP_delta.P_x* should specify how far the cells should be shifted to the right, and *RIP_delta.P_y* should be zero. If the cells are to be shifted vertically, *RIP_delta.P_y* should specify how far the cells should be shifted down, and *RIP_delta.P_x* should be zero.

RIP_cfp This is the address of the **CellFunctionParameters** structure. You don't have to initialize this; the routine will do so automatically.

You may need to perform a certain function on every one of a range of cells. For this purpose, the cell library provides the routine **RangeEnum()**. This routine lets you specify a range of cells and a callback routine; the routine will be called on each cell in that range.

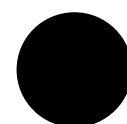
You can sort a range of cells, by row or by column, based on any criteria you choose. Use the routine **RangeSort()**. This routine uses a QuickSort algorithm to sort the cells specified. You supply a pointer to a callback routine which is used to compare cells.

Parse Library



20

20.1	Parse Library Behavior	743
20.1.1	The Scanner	745
20.1.1.1	Scanner Tokens	746
20.1.1.2	Strings	747
20.1.1.3	Cell References.....	748
20.1.1.4	Operators.....	748
20.1.1.5	Identifiers	752
20.1.2	The Parser.....	753
20.1.2.1	The Parser's Grammar	754
20.1.2.2	Parser Tokens.....	754
20.1.2.3	An Example of Scanning and Parsing	756
20.1.3	Evaluator.....	759
20.1.4	Formatter	761
20.2	Parser Functions	761
20.2.1	Internal Functions.....	762
20.2.2	External Functions.....	765
20.3	Coding with the Parse Library	766
20.3.1	Parsing a String.....	766
20.3.2	Evaluating a Token Sequence	767
20.3.3	Formatting a Token Sequence.....	770





The Parse Library was originally created to provide a parser for a spreadsheet language. However, it will also fit the needs of a programmer who wants to implement a language based on mathematical expressions.

The Parse Library takes an expression as text, converts it to an expression using tokens, and evaluates the expression. When finished, it converts the result back into text and returns it. The Parse Library recognizes a special grammar and set of expressions that include an interface to the Cell Library's data structures. Therefore, you can use the Cell and Parse Libraries together to form the basic underlying engine of a spreadsheet application.

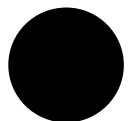
20.1

Applications will generally not use the Parse library directly; instead, they will use higher-level constructs like the Spreadsheet library (see "Spreadsheet Objects," Chapter 20 of the Object Reference Book). Thus, most programmers can just read "Parse Library Behavior" on page 743. The few programmers who will be using the parse library directly can read "Coding with the Parse Library" on page 766. This chapter often says that "the application" must do something; the application pass a callback routine to the parser, for example. If you use a higher-level interface, most of this will be taken care of for you; for example, the spreadsheet library takes care of the bookkeeping chores associated with the Parse Library.

You may want to familiarize yourself with how compilers work before you read this section. In particular, you should understand how scanners use regular expressions to translate raw text into token streams; and you should be familiar with the parsing of context-free grammars. A good book to look at is *Compilers: Principles, Techniques, and Tools* by Aho, Sethi, and Ullman (a.k.a. "The Red Dragon Book").

20.1 Parse Library Behavior

The Parse Library takes a string of characters and evaluates it. In many ways, it acts like a compiler; it translates a string into tokens, evaluates the tokens, and returns the result. It can also reverse the process, translating a sequence of tokens into the equivalent text string. Finally, it can simplify a



string of tokens, performing arithmetic simplifications and calling functions. The parse library provides many useful functions; furthermore, applications can define their own functions.

The different functions are separated into different parts of the parse library. The parse library contains the following basic sections:

20.1

◆ **Scanner**

The scanner reads a text string and converts it into a series of tokens. It does not keep track of the context of the tokens. Its behavior is partially determined by the localization settings; for example, it uses the localization setting to tell whether the decimal separator is a period, a comma, or some other character or string. It is called by the parser; it is not used independently.

◆ **Parser**

The parser interprets the stream of tokens returned by the scanner. It initializes the scanner and uses it to read tokens from the input strings; it also makes sure that the string of tokens is legally formatted. It does not do any type-checking.

◆ **Evaluator**

The evaluator simplifies a token string. It does this by replacing arithmetic expressions with their results, by making function calls, by reading current values of cells, and by replacing identifiers with their values. The result is another token string; usually this string consists of a single token (a number or string).

◆ **Formatter**

The formatter translates a token string into a text string. It is used to display the evaluator's output. Its behavior is influenced by the localization settings.

For example, suppose an application used the parse library to evaluate the string "(5*6)+SUM(A2:C6)". The following steps would be taken:

- 1** The parser would parse the string. It would do this by calling the scanner to read tokens from the string. It would then parse the token sequence to see that it evaluated to a well-formed expression. (It would not do any simplifying or type-checking.)
- 2** The evaluator would simplify the expression. It would reduce the token sequence for "(5*6)" to the single token for "30". It would then call the SUM function, passing it the specifier for the range of cells "A2:C6". The

SUM function would check the type of its arguments, then perform the appropriate action (in this case, adding the values of the cells together). The SUM function would return a value (e.g., it might return 999.9). The evaluator would thus be able to simplify the entire token sequence to the single token for the number 1029.9.

- 3 When the application needed to display the result, it would call the formatter. The formatter would check the localization settings, finding out what the thousands separator and decimal point character are. It would create the string "1,029.9".

20.1

Token strings are usually more compact than the corresponding text strings. There are several reasons for this; for example, cell references are much more compact, functions are specified by an ID number instead of a string, and white space is removed. When translated into a token string, it is only three bytes long: one token byte to specify that this is a number, and two data bytes to store the value of the number. For this reason, applications which use the parse library will generally not store the text entered by the user; instead, they can store the equivalent token string, and use the formatter to display the string when necessary.

The parse library routines often need to request information from the calling application or instruct it to perform a task. For example, when the Parser encounters a name, it needs to get a name ID from the calling application. For this reason, every Parse Library routine is passed a callback routine. The library routine calls this callback routine when necessary, passing a code indicating what action the callback routine should take. The beginning section will just describe this in general terms; for example, "the Evaluator uses the callback to find out the value of a cell." The advanced section provides a more detailed explanation.

20.1.1 The Scanner

The scanner translates a text string into a sequence of tokens. The tokens can then be processed by the parser. Every token is associated with some data.

The scanner can be treated as a part of the parser. It is never used independently; instead, the parser is called on to parse a string, and the parser calls the scanner to translate the string into tokens.

The scanner does not keep track of tokens after it processes them. For this reason, it will not notice if, for example, parentheses are not balanced. It returns errors only if it is passed a string which does not scan as a sequence of tokens.

20.1.1.1 Scanner Tokens

20.1

The scanner recognizes the tokens listed below. Note that applications will never directly encounter the scanner tokens; the tokens translates them into parser tokens before returning them. A complete list of parser tokens (with their names) is given in section 20.1.2.2 on page 754.

- | | |
|-------------------|--|
| NUMBER | This is some kind of numerical constant. The format in the string is determined by the localization settings. The data section of the token is a floating-point number (even if the string contained an integer). |
| STRING | This is a sequence of characters surrounded by "double-quotes." All characters within double quotes are translated into their ASCII equivalents, with the exceptions noted below in section 20.1.1.2 on page 747. The data section is a pointer to the ASCII string specified. |
| CELL | This is a reference to a cell in a database. The format is described in section 20.1.1.3 on page 748. |
| END_OF_EXPRESSION | The scanner returns this token when it has examined and translated an entire text string and reached its end. |
| OPEN_PAREN | This is simply a left parenthesis character, i.e. "(". There is no data section associated with this token. |
| CLOSE_PAREN | This is simply a right parenthesis character, i.e. ")". There is no data section associated with this token. |
| OPERATOR | This is a unary or binary operator. The operators are described in section 20.1.1.4 on page 748. The data section specifies which operator was encountered. |

LIST_SEPARATOR

This is a comma, i.e. “,”. It is used to separate arguments to functions. There is no data section associated with this token.

IDENTIFIER This is a sequence of characters, not in quotation marks, which does not match the format for cell references. Identifiers may be functions (built-in or application-defined) or variables; see section 20.1.1.5 on page 752. The data section is a string containing the identifier.

20.1

20.1.1.2 Strings

The string passed to the scanner may, itself, contain strings. These inner strings are not further analyzed; rather, their contents are associated with the string token. Strings are delimited by double-quotes. All characters within the double-quotes are copied directly into the token's data, with the exception of the backslash, i.e. “\”. This character signals that the character (or characters) which immediately follow it are to be interpreted literally. Backslash-codes include the following:

\"	This code represents a double-quote character (i.e. ASCII 0x22, or "); it indicates that the double-quote should be copied into the string, instead of read as a string delimiter.
\n	This code represents a newline control code (i.e. ASCII 0x0A, or control-J).
\t	This code represents a hard-tab control code (i.e. ASCII 0x09, or control-I).
\f	This code represents a form-feed control code (i.e. ASCII 0x0C, or control-L).
\b	This code represents a backspace control code (i.e. ASCII 0x10, or control-H).
\\	This code represents a backslash character (i.e. ASCII 0x5C, or “\”).
\nnn	This code is a literal octal value. The backslash must be followed by three digits, making up an octal integer in the range 0-177o (i.e. 0-255). The byte specified is inserted directly into the string. Thus, for example, "\134" is functionally identical to "\\\".



20.1.1.3 Cell References

20.1

The parse library is often used in conjunction with cell files; for example, the spreadsheet objects use the two libraries together. For this reason, the scanner recognizes cell references. Cell references are described by the regular expression `[A-Z]+[0-9]+`; that is, one or more capital letters, followed by one or more digits. The capital letters indicate the cell's column. The first column (the column with index 0) is indicated by the letter A; column 1 is B, column 2 is C, and so on, up to column 25 (which is Z). Column 26 is AA, followed by AB, AC, and so on to AZ (column 51); this column is followed by BA, and so on, to the largest column, IV (column 255). The rows are indicated by number, with the first row having number 1.

The data portion of a cell reference token is a **CellReference** structure. This structure records the row and column indices of the cell; the scanner translates the cell reference to these indices. For more information about the cell library, see the Cell Library section in "Database Library," Chapter 19 of the Concepts Book.

When the evaluator needs to get the value of a cell, it calls a callback routine, passing the cell's **CellReference** structure. The application is responsible for looking up the cell's value and returning it to the evaluator. If you manage a cell file with a Spreadsheet object, this work is done for you; the Spreadsheet will be called by the evaluator, returning the values of cells as needed. (The spreadsheet returns zero for empty or unallocated cells.)

Note that while the cell library numbers both rows and columns starting from zero, the Parse library numbers rows starting from one. This is because historically, spreadsheets have had the first row be row number 1. Therefore, if the parser encounters a reference to cell A1, it will translate this into a cell reference which specifies row zero, column zero.

20.1.1.4 Operators

The scanner recognizes a number of built-in operators. Neither the scanner nor the parser does any simplification or evaluation of operator expressions; this is done by the evaluator. All operators are represented by the token `SCANNER_TOKEN_OPERATOR`. The token has a one-byte data section, which is a member of the enumerated type **OperatorType**; this value specifies which operator was encountered. This section begins with a listing of

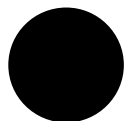
currently supported operators in order of precedence, from highest precedence to lowest; this is followed by a detailed description of the operators. All operators listed here will always be supported; other operators may be added in the future.

Note that neither the scanner nor the parser does any evaluation of arguments. All type-checking is done at evaluation time. Thus, if parse the text “(3 * "HELLO")”, the parser will not complain; the evaluator, however, will return a “bad argument type” error.

20.1

Table 20-1 on page ● 750 lists the operators in order of precedence. Highest-precedence operators are listed first. Operators with the same precedence are listed together; a blank line implies a drop in precedence. Operators of the same precedence level are grouped from left to right; that is, “1 - 2 - 3” is the same as “(1 - 2) - 3”.

:	This is a range separator. The range separator is a binary infix operator. The parser recognizes expressions of the format <i>Cell1:Cell2</i> as describing a rectangular range of cells, with the two specified cells being diagonally opposite corners. The data portion of this token is the constant OP_RANGE_SEPARATOR.
...	This is another range separator. It is functionally identical to the colon operator. The data portion of this token is the constant OP_RANGE_SEPARATOR. (The formatter will turn this back into a colon.)
-	This can be either of two different operators. It can be a negation operator. This is a unary prefix operator which reverses the arithmetic sign of the operand. It can also be a subtraction operator. This is a binary infix operator. The parser determines which operator is represented. For example, in “(-1)”, the hyphen is a negation operator; in “(1-2)”, it is a subtraction operator. The data portion of this token is either OP_NEGATION or OP_SUBTRACTION; the scanner assigns the neutral OP_SUBTRACTION_NEGATION, and the parser decides (from context) which value is appropriate.
%	This can be either of two operators. It can be a percent operator. This is a unary postfix operator which divides its operand by 100; that is, “50%” evaluates to 0.5. It can also be a modulo arithmetic operator. This is a binary infix operator which returns the remainder when its first operand is divided by its



second operand; that is, “11%4” evaluates to 3.0. The parser determines which operator is represented. The data portion of this token is either OP_PERCENT or OP_MODULO; the scanner assigns the neutral OP_PERCENT_MODULO, and the parser decides (from context) which constant is appropriate.

^ This is the exponentiation operator. It is a binary infix operator; it raises its first operand to the power of the second operand (e.g. “2^3” evaluates to 8.0). The data portion of this token is the constant OP_RANGE_EXPONENTIATION.

20.1

Table 20-1 *Parse Library Operators*

Operator	Description
:	range separator
...	range separator (alternate form)
#	range intersection
-	unary negation
%	unary percent
^	exponentiation
*	multiplication
/	division
%	modulo division
+	addition
-	subtraction
&	string concatenation
=	Boolean or string equality
<>	Boolean or string inequality
<	Boolean or string less-than
<=	Boolean or string less-than-or-equal-to
>	Boolean or string greater-than
>=	Boolean or string greater-than-or-equal-to



* This is the multiplication operator. It is a binary infix operator. It multiplies the two operands. The data portion of this token is the constant OP_MULTIPLICATION.

/ This is the division operator. It is a binary infix operator. It divides the first operand by the second. The data portion of this token is the constant OP_DIVISION. The constant OP_DIVISION_GRAPHIC is functionally equivalent; however, the formatter will display the operator as “÷”.

+ This is the addition operator. It is a binary infix operator. It adds the two operands. The data portion of this token is the constant OP_RANGE_ADDITION.

20.1

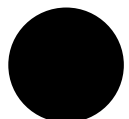
Several Boolean operators are also provided. In every case, if a Boolean expression is true, it evaluates to 1.0; if it is false, it evaluates to 0.0. (There is no Boolean negation operator; however, there is a Boolean negation function, NOT, which returns 1.0 if its argument is zero, and otherwise returns zero.) Boolean operators may be used for numbers or strings. They work in the conventional way for numbers. Strings are “equal” if they are identical. One string is said to be “less than” another if it comes first in lexical order. The parse library uses localized string comparison routines to compare strings; thus, the local lexical ordering is automatically used. (For more information, see “Localization,” Chapter 8.)

= This is the equality operator. It is a binary infix operator. An expression evaluates to 1.0 if both operands evaluate to identical values. The data portion of this token is the constant OP_EQUAL.

<> This is the inequality operator. It is a binary infix operator. An expression evaluates to 1.0 if the two operands evaluate to different values. The data portion of this token is OP_NOT_EQUAL. The constant OP_NOT_EQUAL_GRAPHIC is functionally equivalent; however, the formatter will display the operator as “≠”.

> This is the “greater-than” operator. It is a binary infix operator. It returns 1.0 if the first operand evaluates to a larger number than the second operand. The data portion of this token is OP_GREATER_THAN.

< This is the “less-than” operator. It is a binary infix operator. It returns 1.0 if the first operand evaluates to a smaller number



than the second operand. The data portion of this token is `OP_LESS_THAN`.

`>=` This is the “greater-than-or-equal-to” operator. It is a binary infix operator. It returns 1.0 if the first operator evaluates to a number that is greater than or equal to the value of the second operand. The data portion of this token is `OP_GREATER_THAN_OR_EQUAL`. The constant `OP_GREATER_THAN_OR_EQUAL_GRAPHIC` is functionally equivalent; however, the formatter will display the operator as “≥”.

20.1

`<=` This is the “less-than-or-equal-to” operator. It is a binary infix operator. It returns 1.0 if the first operator evaluates to a number that is less than or equal to the value of the second operand. The data portion of this token is `OP_LESS_THAN_OR_EQUAL`. The constant `OP_LESS_THAN_OR_EQUAL_GRAPHIC` is functionally equivalent; however, the formatter will display the operator as “≤”.

Some special-purpose operators are also provided:

`&` This is the “string-concatenation” operator. It is a binary infix operator. The arguments must be strings. The result is a single string, composed of all the characters of the first string (without its null terminator) followed by all the characters of the second string (with its null terminator); for example, (“Franklin” & “Poomm”) evaluates to “FranklinPoomm”. The data portion of this token is `OP_STRING_CONCAT`.

`#` This is the “range-intersection” operator. It is a binary infix operator. Both arguments must be cell ranges. The result is the range of cells which falls in both of the operand cell ranges. Note that cell ranges must be rectangular; there is, therefore, no “range-union” operator. The data portion of this token is `OP_RANGE_INTERSECTION`.

20.1.1.5 Identifiers

Any unbroken alphanumeric character sequence which does not appear in quotes, and which is not in the format for a cell reference, is presumed to be

an identifier. Identifiers serve two roles: they may be function names, or they may be labels.

The scanner merely notes that an identifier has been found; it does not take any other action. The parser will find out what the identifier signifies. If the identifier's position indicates that it is a function (but the name is not that of a built-in function), the parser will prompt its caller for a pointer to a callback routine which will perform this function. If its position indicates that it is an identifier, the parser will request the value associated with the identifier; this may be a string, a number, or a cell reference.

20.1

20.1.2 The Parser

Applications will never call the scanner directly. Instead, if they access the parse library directly (instead of through the spreadsheet objects), they will call the parser and pass it a string, and the parser will in turn call the scanner to process the string into tokens. This section will not discuss how to call the parser, since few applications will need to do that; it will instead describe the general workings of the parser.

The parser translates a well-formed string into a sequence of tokens. It calls the scanner to read tokens from the string. It then uses a context-free grammar to make sure the string is well formed. The context-free grammar is described below. The scanner outputs a sequence of parser tokens. The parser tokens are almost identical to the scanner tokens, with a few exceptions; those exceptions are noted below.

The parser is passed a callback routine. The parser calls this routine when it needs information about a token; for example, if it encounters a function it does not recognize, it calls the callback to get a pointer to the function. The details of this are provided in the advanced section.

If the parser is not passed a well-formed expression, or if it is unable to successfully parse the string for some other reason, it returns an error code. The error codes are described at length in the advanced section.

20.1.2.1 The Parser's Grammar

The parser uses a context-free grammar to make sure the string is well-formed. The grammar is listed below. The basic units of the grammar are listed in ALL-CAPS; higher-level units are listed in *italics*. The string must parse to a well-formed *expression*.

20.1

expression: '(' *expression* ')'
NEG_OP *expression*
IDENTIFIER '(' *function_args* ')'
base_item *more_expression*

more_expression:
<empty>
PERCENT_OP *more_expression*
BINARY_OP *expression*

function_args:
<empty>
arg_list

arg_list: *expression*
expression ',' *arg_list*

base_item: NUMBER
STRING
CELL_REF
IDENTIFIER

20.1.2.2 Parser Tokens

The parser does not return scanner tokens; instead, it returns a sequence of parser tokens. The parser tokens are almost directly analogous to the scanner tokens. However, a few additional token types are added.

The parser tokens have the same structure as the scanner tokens. The first field is a constant specifying what type of token this is. The second field contains specific information about the token; this field may be blank. The parser has the following types of tokens:

PARSER_TOKEN_NUMBER
This is the same as the scanner NUMBER token.

PARSER_TOKEN_STRING

This is the same as the scanner STRING token.

PARSER_TOKEN_CELL

This is the same as the scanner CELL token.

PARSER_TOKEN_END_OF_EXPRESSION

This is the same as the scanner END_OF_EXPRESSION token.

PARSER_TOKEN_OPEN_PAREN

This usually replaces the scanner OPEN_PAREN token.

However, it is not used if the parenthesis is delimiting function arguments; it is only used if the parenthesis is changing the order of evaluation.

20.1

PARSER_TOKEN_CLOSE_PAREN

This usually replaces the scanner CLOSE_PAREN token.

However, it is not used if the parenthesis is delimiting function arguments; it is only used if the parenthesis is changing the order of evaluation.

PARSER_TOKEN_NAME

This replaces some occurrences of the scanner IDENTIFIER token; specifically, those where the identifier is not a function name. The data portion is the number for that name.

PARSER_TOKEN_FUNCTION

This replaces some occurrences of the scanner IDENTIFIER token, specifically those in which the identifier is a function name. The data portion is the function ID number.

PARSER_TOKEN_CLOSE_FUNCTION

This replaces some occurrences of the scanner CLOSE_PAREN token; specifically, those where the closing parenthesis delimits function arguments.

PARSER_TOKEN_ARG_END

The parser inserts this token after every argument to a function call; thus, it replaces occurrences of SCANNER_TOKEN_LIST_SEPARATOR, and also occurs after the last argument to a function.

PARSER_TOKEN_OPERATOR

This is the same as the parser's OPERATOR token. The data section is an operator constant, as described above in

“Operators” on page 748. Note the parser replaces occurrences of `OP_PERCENT_MODULO` with either `OP_PERCENT` or `OP_MODULO`, as appropriate; similarly, it replaces `OP_SUBTRACTION_NEGATION` with either `OP_SUBTRACTION` or `OP_NEGATION`.

20.1

When the parser encounters an identifier that is in the appropriate place for a function name (that is, an identifier followed by a parenthesized argument list), it does not write an identifier token. Instead, it writes a “function” token, which has a one-word data section. This section is the function ID (described in section 20.2 on page 761). If the function’s name is not one of a built-in function, it will call the application’s callback routine to find out what the function’s ID number is; the evaluator will pass this ID when it needs to have the function called.

When the parser encounters an identifier, it asks its caller for an ID number for the identifier. It can then store the ID number instead of the entire string. The evaluator will use this ID number when requesting the value of the identifier. The formatter will use the ID number when requesting the original identifier string associated with the ID number.

When the parser encounters a scanner parenthesis token, it does not necessarily translate it into a parser parenthesis token. This is because parentheses fulfill two separate roles: they specify the order of evaluation, and they delimit function arguments. When the parser encounters parenthesis tokens which specify order of evaluation, it translates them into parser parenthesis tokens. If, however, it encounters argument-delimiting parentheses, it does not need to translate them literally; after all, the presence of a function token implies that it will be followed by an argument list. Thus, the parser does not need to copy the parenthesis tokens. Instead, it copies the tokens of the argument list. When it reaches a list separator, it replaces that with an “end-of-argument” token; when it reaches the closing parenthesis for the function call, it replaces that with a “close-function” token.

20.1.2.3 An Example of Scanning and Parsing

Suppose that you call the parser on the text string `"3 + SUM(6.5, 3 ^ (4 - 1), C5...F9)"`. The parser will evaluate the string, one token at a time. When it needs to process a token, it will call the scanner to

return the next token in the string. It will then replace these tokens with parser tokens, and write out the sequence of tokens to its output buffer.

For simplicity, this example treats the scanner as if it scanned the entire text stream at once, and returned the entire sequence of tokens to the scanner. In this case, the scanner would translate the text into the following sequence of tokens:

Token	Data	Comment	
NUMBER	3.0	All numbers are floats	20.1
OPERATOR	OP_ADDITION		
IDENTIFIER	"SUM"		
OPEN_PAREN		delimits function args	
NUMBER	6.5		
LIST_SEPARATOR			
NUMBER	3.0		
OPERATOR	OP_EXPONENTIATION		
OPEN_PAREN			
NUMBER	4.0		
OPERATOR	OP_SUBTRACTION_NEGATION	Parser figures out which operator this is	
NUMBER	1.0		
CLOSE_PAREN			
LIST_SEPARATOR			
CELL	C5	Actually stored as "4,2"; row index 4, column index 2	
OPERATOR	OP_RANGE_SEPARATOR		
CELL	F9		
CLOSE_PAREN			
END_OF_EXPRESSION			



20.1

The parser reads these tokens, one at a time, and writes out an analogous sequence of parser tokens:

Token	Data	Comment
NUMBER	3.0	All numbers are floats
OPERATOR	OP_ADDITION	
FUNCTION	FUNCTION_ID_SUM	
NUMBER	6.5	
END_OF_ARG		
NUMBER	3.0	
OPERATOR	OP_EXPONENTIATION	
OPEN_PAREN		
NUMBER	4.0	
OPERATOR	OP_SUBTRACTION	
NUMBER	1.0	
CLOSE_PAREN		
END_OF_ARG		
CELL	C5	Actually stored as "4,2"; row index 4, column index 2
OPERATOR	OP_RANGE_SEPARATOR	
CELL	F9	
END_OF_ARG		
CLOSE_FUNCTION		
END_OF_EXPRESSION		

The application does not need to save the original text string. Instead, it can save the buffer containing the parser tokens, and use the formatter to translate the token sequence back into a character string.

20.1.3 Evaluator

The evaluator simplifies a token string returned by the parser. If the input token sequence was well-formed (as are all token sequences generated by the Parser), the evaluator will produce a token sequence consisting of two tokens: a single “result” token (which may be an error token), followed by the “end-of-expression” token. It does this by doing two main things: simplifying arithmetic expressions, and making function calls.

20.1

The evaluator maintains two stacks, an Operator stack and an Argument stack. It reads the tokens from beginning to end. Each time it reads a token, it takes an action; this may involve pushing something onto a stack, or processing some of the tokens on the tops of the stacks.

If an error occurs, the parser may take two different actions. Some errors are pushed on the argument stack; these may be handled by functions. For example, if the result of an expression is too large to be represented, the evaluator will just push PSEE_FLOAT_POS_INFINITY on the argument stack. Any function or operator which is passed an error code as an argument can either handle the error, propagate the error, or return a different error. For example, if the division operator is passed PSEE_FLOAT_POS_INFINITY as the divisor, it will simply return zero.

The actual evaluation of tokens is straightforward. The evaluator pops the top token from the Operator stack. This is either a function or an operator. If the token is an operator, the evaluator pops either one or two arguments from the top of the argument stack, takes the appropriate action, and pushes the result on the argument stack. If the token is a function, the evaluator calls the function directly, passing it a pointer to the argument stack and the number of arguments to the function call. The function is responsible for popping off all of the arguments and pushing the return value on the argument stack.

Special actions have to be taken if an operand or argument is a cell reference. If the cell is an argument to a function, or an operand (and the operator is not a range-separator or range-intersection), the evaluator will call its callback routine to get the value contained by the cell; this value will be put on the argument stack in place of the cell reference. If the operand is a range-separator or range-intersection, the cells or ranges will be combined into a single range, which is pushed on the Argument stack.

The evaluator reads tokens, one at a time, from the buffer provided by the parser. For each token it takes an appropriate action:

OPEN_PAREN	Push an OPEN_PAREN token on the Operator stack.
CLOSE_PAREN	Evaluate tokens from the Operator stack until an OPEN_PAREN reaches the top of the operator stack; then pop the OPEN_PAREN off the stack.
OPERATOR	If the top token on the Operator stack is an OPERATOR of higher precedence than this OPERATOR, then evaluate top of Operator stack. Repeat until top of operator stack is either not an operator, or is an operator of lower precedence. Finally, push the operator token on the operator stack.
FUNCTION	Push FUNCTION token on the Operator stack. The evaluator FUNCTION token contains the function ID and the number of arguments to the function (starting at zero).
CLOSE_FUNCTION	Call function on top of Operator stack, passing it the pointer to the Argument stack and the number of arguments to the function call. (The arguments will be on the top of the argument stack.) The function should pop the arguments off the Argument stack, then push the return value (or error code) on the Argument stack.
ARG_END	Evaluate the Operator stack until a FUNCTION token is at the top of the Operator stack; then increment the argument count of that function.
NUMBER	Push number on Argument stack. (Actually, what is pushed is a reference to the thread's floating-point stack, which contains the number itself.)
STRING	Push string on Argument stack.
CELL_REF	Push the cell reference on the Argument stack.
NAME	Call the callback function to find the value associated with the name; act on the value appropriately.

END_OF_EXPRESSION

Evaluate the Operator stack until it is empty; the result will be on the top of the Argument stack.

20.1.4 Formatter

In order to display a token sequence, you must call the Formatter. The formatter is very straightforward. It is passed a buffer containing a token sequence; it returns a character array containing the result. The formatter makes use of the localization routines to format the result according to the local language and the user's Preferences settings.

20.2

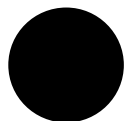
If the token sequence consists of an error token, the formatter will generate an appropriate error string.

20.2 Parser Functions

The Parse library provides many built-in functions. Furthermore, each application can define its own functions. Every function is associated with a function ID number. Built-in functions have ID numbers assigned to them in the library code; application-defined functions are given ID numbers by the application. ID numbers are word-sized unsigned integers. All built-in ("internal") functions have ID numbers which are less than the constant `FUNCTION_ID_FIRST_EXTERNAL_FUNCTION_BASE`; all application-defined ("external") functions have ID numbers which are greater than this constant.

When the Parser reads an identifier token whose position indicates that it is a function, it converts the identifier to a function token (containing a function ID). The parser first checks to see if the identifier is the name of a built-in function. If so, it looks up the function's ID number and stores it in the function token.

If the identifier is not the name of a built-in function, the Parser calls the application's callback routine to get the function's ID number. The application must assign each function a word-sized ID which is greater than or equal to the constant `FUNCTION_ID_FIRST_EXTERNAL_FUNCTION_BASE`. This constant is defined as `0x8000`, which leaves 2^{15} ID numbers available.



When the Evaluator needs to evaluate a function, it checks to see if the function is external or internal. If the function is internal, it looks up the functions address and calls it. If the function is external, it calls the application's callback routine and passes the function ID. In either case, it passes a pointer to the argument stack and the number of arguments. The function is responsible for popping all the arguments off the stack and pushing the result. It can also push an error message on the stack. All of this is discussed at length in the advanced section.

20.2

20.2.1 Internal Functions

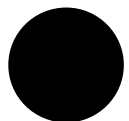
The Parse library provides many internal functions, and more are continually being added. Any application which uses the parse library automatically makes use of these functions. Some of these functions take a single argument; others take a set number of arguments or a variable number.

A listing of currently available functions follows, along with a short description of each one.

ABS	Absolute value
ACOS	Arc-cosine
ACOSH	Hyperbolic arc-cosine
AND	Boolean AND
ASIN	Arc-sine
ASINH	Hyperbolic arc-sine
ATAN	Arc-tangent
ATAN2	Four-quadrant arc-tangent
ATANH	Hyperbolic arc-tangent
AVG	Average of arguments
CHAR	Translates character-set code into character
CHOOSE	Finds value in list at specified offset
CLEAN	Removes control characters from a string

CODE	Translates character into character-set code
COLS	Returns # of columns in range
COS	Cosine
COSH	Hyperbolic cosine
CTERM	Returns time for an investment to reach a specified value
DDB	Depreciation over a period
DEGREES	Converts radians to degrees
ERR	Returns error PSEE_GEN_ERR
EXACT	Tests if two strings match
EXP	Exponentiation
FACT	Factorial
FALSE	Returns false (0.0)
FIND	Returns position in string where substring first occurs
FV	Future value of investment
HLOOKUP	Finds a value in a horizontal lookup table
IF	IF(<cond>,x,y) = x if <cond> is true, else y (like C's "<cond> ? x : y")
INDEX	Finds value at specified offset in a range
INT	Rounds to next lowest integer
IRR	Internal rate of return
ISERR	True if argument is error
ISNUMBER	True if argument is number
ISSTRING	True if argument is string
LEFT	Returns first characters in string
LENGTH	Returns length of string
LN	Natural log

20.2



Parse Library

764

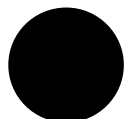
20.2

LOG	Log to base 10
LOWER	Converts string to all-lowercase
MAX	Returns largest of arguments
MID	Returns characters from middle of string
MIN	Returns smallest of arguments
MOD	Modulo arithmetic
N	Returns value of first cell in range
NA	Returns error PSEE_NA
NPV	Returns net present value of future cash flows
OR	Boolean OR
PI	Returns 3.1415926...
PMT	Calculates # of payments to pay off a debt
PRODUCT	Returns product of arguments
PROPER	Converts string to "proper capitalization"
PV	Calculates present value of an investment
RADIANS	Converts radians to degrees
RANDOM	Generates random number between 0 and 1
RANDOMN	Generates random integer below a specified ceiling
RATE	Calculates interest rate needed for investment to reach specified value
REPEAT	Returns string made of repeated argument string
REPLACE	Replaces characters in a string
RIGHT	Returns last characters in a string
ROUND	Rounds number to specified precision
ROWS	Returns number of rows in range

SIN	Sine	
SINH	Hyperbolic-sine	
SLN	Calculates straight-line depreciation	
SQRT	Square-root	
STD	Calculates standard deviation	
STDP	Standard deviation of entire population	
STRING	Converts number into string	20.2
SUM	Returns sum of arguments	
SYD	Sum-of-years'-digits depreciation	
TAN	Tangent (sine/cosine)	
TANH	Hyperbolic tangent (sinh/cosh)	
TERM	Returns number of payments needed to reach future value	
TRIM	Removes leading, trailing, and consecutive spaces from a string	
TRUE	Returns TRUE (1.0)	
TRUNC	Removes fractional part; rounds towards zero	
UPPER	Converts all letters in string to uppercase	
VALUE	Converts string to number	

20.2.2 External Functions

Applications which use the Parse library may write their own functions. Whenever the formatter encounters a function name which it does not recognize, it calls the application to get an ID for the function. When the evaluator needs to evaluate that function, it calls the application, passing the arguments and the function ID. The application should return a single value. If it cannot produce a value, it should return an error code. The error codes are described in section 20.3.2 on page 767.



20.3 Coding with the Parse Library

This section describes how to use the Parser directly, instead of using intermediaries (like the Spreadsheet library). Most applications will not need to use these routines.

20.3

20.3.1 Parsing a String

`ParserParseString()`

To parse a string, all you do is call **ParserParseString()**. This routine takes three arguments: A pointer to a null-terminated string, a pointer to a buffer, and a pointer to a **ParserReturnStruct**. This structure contains a pointer to a callback routine. **ParserParseString()** parses the string into a sequence of tokens and writes the tokens to the buffer. Whenever the parser encounters an identifier, it calls the callback routine and requests an ID number for the identifier. Similarly, when the parser encounters a function whose name it does not recognize, it calls the callback routine to get a function ID number. The ID numbers are stored in the token sequence.

The Parser can return the following errors:

PSEE_BAD_NUMBER

The string contained a badly-formatted number.

PSEE_BAD_CELL_REFERENCE

The string contained a badly-formatted cell reference.

PSEE_NO_CLOSE_QUOTE

The string contained an opening quote with no matching closing quote.

PSEE_COLUMN_TOO_LARGE

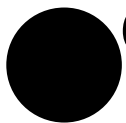
The string contained a cell whose column index was out of bounds (greater than 255).

PSEE_ROW_TOO_LARGE

The string contained a cell whose row index was out of bounds.

PSEE_ILLEGAL_TOKEN

The string contained a character sequence which was not a legal token.



PSEE_TOO_MANY_TOKENS	The expression was too complex.	
PSEE_EXPECTED_OPEN_PAREN	A function call lacked an open-parenthesis.	
PSEE_EXPECTED_CLOSE_PAREN	A function call lacked a close-parenthesis.	
PSEE_BAD_EXPRESSION	The string contained a badly-formed expression.	20.3
PSEE_EXPECTED_END_OF_EXPRESSION	An expression ended improperly.	
PSEE_MISSING_CLOSE_PAREN	Parentheses were mismatched.	
PSEE_UNKNOWN_IDENTIFIER	An identifier or external function name was encountered, and the callback routine would not provide an ID for it.	
PSEE_GENERAL	General parser error.	

20.3.2 Evaluating a Token Sequence

`ParserEvalExpression()`

To format an expression, call **ParserEvalExpression()**. This routine is passed a token sequence; it evaluates it and writes the result, another token sequence, to a passed buffer. It calls a supplied callback routine to perform the following tasks:

- ◆ Return the value of a specified cell
- ◆ Return the value associated with a given identifier, specified by ID number
- ◆ Evaluate an external function, given the arguments and the function ID number

The evaluator produces a sequence two tokens long, including the “end-of-expression” token. The first token might be an error token. Two

errors are so serious that if they occur, the evaluation is immediately halted and the error is returned:

PSEE_OUT_OF_STACK_SPACE

The evaluator ran out of stack space. Evaluation was halted when this occurred.

PSEE_NESTING_TOO_DEEP

The nesting grew too deep for the evaluator. Evaluation was halted when this occurred.

20.3

The following errors may be propagated; that is, if an expression returns an error, that error would be passed, as a value, to outer expressions. For example, if the evaluator were evaluating

“SUM(1, (PROD(1, 2, "F. T. Poomm")))”, PROD would return PSEE_WRONG_TYPE, since it expects numeric arguments. SUM, in turn, would be passed two arguments: the number 1 and the error PSEE_WRONG_TYPE. That function might, in turn, propagate the error upward, return a different error, or return a non-error value. (SUM, as it happens, would propagate the error; that is, it would return PSEE_WRONG_TYPE.)

PSEE_ROW_OUT_OF_RANGE

A cell's row index was out of range.

PSEE_COLUMN_OUT_OF_RANGE

A cell's column index was out of range.

PSEE_FUNCTION_NO_LONGER_EXISTS

The callback routine did not recognize the function ID for an external function.

PSEE_BAD_ARG_COUNT

A function was passed the wrong number of arguments.

PSEE_WRONG_TYPE

A function was passed an argument of the wrong type.

PSEE_DIVIDE_BY_ZERO

A division by zero was attempted.

PSEE_UNDEFINED_NAME

The callback would not provide a value for an identifier ID.

PSEE_CIRCULAR_REF

A circular reference occurred. This error will only occur if it is returned by the callback routine.

PSEE_CIRCULAR_DEP

The value is dependant on a cell whose value is PSEE_CIRCULAR_REF.

PSEE_CIRC_NAME_REF

The expression uses a name which is defined circularly.

20.3

PSEE_NUMBER_OUT_OF_RANGE

The result was a number which could not be expressed as a float.

PSEE_GEN_ERR

General error; this is returned when no other error code is appropriate.

PSEE_NA The value for a cell was not available.**PSEE_FLOAT_POS_INFINITY**

A float routine returned the error FLOAT_POS_INFINITY.

PSEE_FLOAT_NEG_INFINITY

A float routine returned the error FLOAT_NEG_INFINITY.

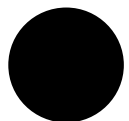
PSEE_FLOAT_GEN_ERR

A float routine returned the error FLOAT_GEN_ERR.

PSEE_TOO_MANY_DEPENDENCIES

The formula contained too many levels of dependency. This is generally returned by the callback routine; the Parse library routines do not return this error, they merely propagate it.

The application may also define its own error codes, beginning with the constant PSEE_FIRST_APPLICATION_ERROR. All internal functions, and all operators, always propagate application-defined errors.



20.3.3 Formatting a Token Sequence

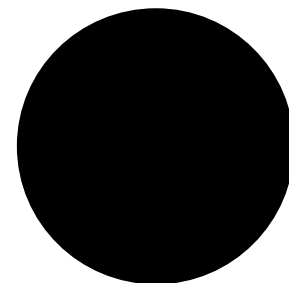
`ParserFormatExpression()`

The routine **ParserFormatExpression()** is passed a token buffer; it returns a character string. The formatter uses the localization routines to format numbers. The formatter also formats error codes as appropriate error messages. These error messages are stored in a localizable resource, so the formatter library will produce error messages in the appropriate language.

20.3

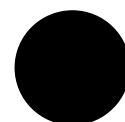
If the formatter encounters a token ID or an external function ID, it will call the callback routine to find out what character sequence is associated with that ID number. If it encounters an application-defined error code, it will request an appropriate error string. Applications should store these error strings in localizable resources; this will simplify translating the application into another language.

Using Streams



21

21.1	Using Streams: The Basics	773
21.1.1	Initializing a Stream.....	775
21.1.1.1	Creating a Stream.....	775
21.1.1.2	Assigning Readers and Writers.....	776
21.1.2	Blocking on Read or Write	777
21.1.3	Writing Data to a Stream.....	778
21.1.4	Reading Data from a Stream	779
21.1.5	Shutting Down a Stream.....	781
21.1.6	Miscellaneous Functions	782
21.2	Using the Serial Ports	782
21.2.1	Initializing a Serial Port.....	783
21.2.1.1	Opening a Serial Port	783
21.2.1.2	Configuring a Serial Port.....	784
21.2.2	Communicating.....	787
21.2.3	Closing a Serial Port	788
21.3	Using the Parallel Ports	789
21.3.1	Initializing a Parallel Port.....	789
21.3.2	Communicating.....	790
21.3.3	Closing a Parallel Port	790





It is often useful for an application to be able to write out data in an orderly manner to be read by another program or sent to a device such as a printer or modem. GEOS provides a mechanism called a *stream* to allow an orderly flow of data between two programs, between two threads of a multi-threaded program, or between a program and a device such as a serial or parallel port. The stream interface includes various ways to notify a program that bytes are available to read, or that there is room to write additional data.

21.1

The GEOS parallel and serial port drivers use the stream mechanism, so programs that will use these ports must do so via this mechanism; however, libraries exist to handle some of the more common uses for port communications. If you wish to monitor a serial line for pccom communications from another machine, you will be working with the PCCom library and should read “PCCom Library,” Chapter 22. Programs do not need to access the serial or parallel ports in order to print because the spooler does it for them. For more information about the spooler and printing, see “The Spool Library,” Chapter 17 of the Object Reference Book.

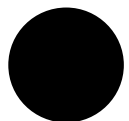


This chapter may be read with only a general understanding of GEOS and Object Assembly. Messaging is used heavily in informing the application using a stream of the stream's status, so a good understanding of object-oriented programming will be helpful. You should also be familiar with the GEOS memory manager. The chapter is divided into three main sections:

- ◆ A general description of a GEOS stream and how the two sides (writers and readers) access it.
- ◆ A description of the specific routines and conventions for using streams to send data to and receive data from a serial port.
- ◆ A description of the specific routines and conventions for using a stream to send data to a parallel port.

21.1 Using Streams: The Basics

A stream is a path along which information flows in one direction. At one end of the stream is a *writer*, who puts data into the stream. At the opposite end



of the stream is a *reader*, who receives data from the stream. The writer and reader can be GEOS programs or device drivers. Data flow in both directions can be achieved by using two streams, one in each direction; this is how the serial port driver is implemented. See Figure 21-1 for an illustration.

Even though the stream driver is loaded automatically when a stream is created, you will need to initialize, configure, and destroy any streams you use. The specific steps involved in this process are

21.1

- 1** Get the handle of the stream driver.
You will need to get this handle to use most of the stream-library routines. You can get this handle by calling **GeodeGetInfo()**.
- 2** Create the stream.
You must create each stream you plan on using. When a stream is initialized, it is designated a token that is used when calling the stream driver's strategy routine.
- 3** Configure the stream.
Arrange how your geode will be notified by the stream driver when certain situations (error received or buffer full/empty) arise, and make sure that all geodes accessing the stream have been given the stream's token.
- 4** Use the stream.
- 5** Shut down the stream.
Not a trivial task, shutting down a stream can involve several synchronization issues.

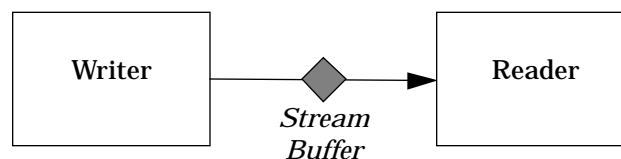


Figure 21-1 *Stream Writer and Reader Interaction*

Every stream has a writer on one end and a reader on the other, and each has access to the stream's central buffer. Note that two-way data transfer requires the use of two separate streams, one in each direction.

Streams are created and managed by the *Stream Driver*. Programs written in Object-Assembly can call the driver directly. Goc programs cannot do this; instead, they make calls to the *Stream Library*, which in turn calls the Stream Driver, and passes back any return values.

21.1.1 Initializing a Stream

A stream is essentially a first-in-first-out data buffer, in which the writer is different from the reader. When the writer writes data to the stream, the kernel stores it in the buffer; when the reader requests information from the stream, the kernel retrieves the oldest data not yet read. The data is stored in a memory block; this block may be either fixed or movable. If it is movable, both the reader and the writer must lock the block before calling any stream routines.

21.1

Note that the kernel does not enforce who is the reader or writer to a stream. Any geode may call the appropriate stream library routine, passing in the token for a stream, and read or write data. However, in practice, only those threads with a legitimate interest in a stream will know the stream's token.

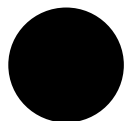
The serial and parallel drivers are built on top of the stream driver. There are separate routines to access the serial and parallel ports; these routines are discussed in “Using the Serial Ports” on page 782 for the serial driver and “Using the Parallel Ports” on page 789 for the parallel driver.

21.1.1.1 Creating a Stream

`StreamOpen()`

To create and initialize a new stream, call the routine **StreamOpen()**. This routine takes five arguments:

- ◆ The handle of the stream driver.
- ◆ The size of the stream buffer, in bytes. This may not be larger than 32767.
- ◆ The **GeodeHandle** of the geode that will own this stream. When this geode exits, the stream will be freed; however, you should call **StreamClose()** before this happens.



- ◆ A set of **HeapFlags**. The routine will have to allocate a block to hold the stream. The **HeapFlags** specify whether that block will be fixed or movable. If it is fixed, this argument should contain the flag `HF_FIXED`; otherwise it should be blank.
- ◆ The pointer to a **StreamToken** variable. **StreamOpen()** will create the stream and write its token to this variable. You will need this token whenever you access the stream, for reading or writing.

21.1

If the creation is successful, **StreamOpen()** will return zero and store the stream's token in the **StreamToken** variable. You must see to it that both the reader and the writer have this token. If the stream cannot be created, the strategy routine will set an error flag and return either `STREAM_CANNOT_ALLOC` (if the memory for the stream's buffer cannot be allocated) or `STREAM_BUFFER_TOO_LARGE` (if the requested stream size was greater than 32767).

21.1.1.2 Assigning Readers and Writers

Once a stream is created, you must make sure that both ends will be managed—a stream that has only a writer or only a reader is not a useful stream.

When communicating with a device such as a serial or parallel port, the port is considered to be the entity on the other end. However, if two threads are communicating via a stream, you must make sure the other thread can gain access to the stream. The best way to do this is to set up a message that will be sent by the creator to the other geode. This message should contain as an argument the token of the stream and probably the direction of the stream (whether the creator will be reading or writing).

Once both geodes have the stream's token, each can access the stream normally. The next several sections explain how to access a stream for writing and reading.

21.1.2 Blocking on Read or Write

`StreamBlocker`, `StreamError`

A stream is a data buffer of limited size. When a thread writes to the stream, there is a chance it could run out of space. Similarly, when a thread reads from the stream, there is a possibility that it will try to read more data than is available; for example, it might try to read 500 bytes, when only 250 bytes of data are sitting in the stream.

21.1

There are two ways you can deal with these situations. One way is, you can instruct the thread to block. For example, if you try to write 500 bytes to a stream and there is only 200 bytes of space available, the driver will write the first 200 bytes to that space, then have the writing thread block until more space is available (i.e. until the reading thread has read some data). The writing thread will not resume execution until all the data has been written. Similarly, a reading thread could block until the stream provided all the data it requested.

The other approach is to have the stream driver write or read all it can, then return an appropriate error code. This requires a little more work by the calling thread, as it cannot assume that all the data is always read or written; however, it avoids the risk of deadlock.

All read and write routines are passed a member of the **StreamBlocker** enumerated type. This type has two members: `STREAM_BLOCK`, indicating that the calling thread should block in the situations described above; and `STREAM_NO_BLOCK`, indicating that the routine should immediately return with an error if enough space is not available. A single thread may, if it wishes, pass `STREAM_BLOCK` sometimes and `STREAM_NO_BLOCK` sometimes.

If a stream routine returns an error, the error will be a member of the **StreamError** enumerated type. The possible error values are described in the section for each routine.

21.1.3 Writing Data to a Stream

`StreamWrite()`, `StreamWriteByte()`

To write data into a stream, call the routine **StreamWrite()**. This routine takes six arguments:

21.1

- ◆ The **GeodeHandle** of the stream driver.
- ◆ The **StreamToken** of the stream.
- ◆ A member of the **StreamBlocker** enumerated type, as described in section 21.1.2 on page 777.
- ◆ The amount of data to be written, in bytes.
- ◆ A pointer to a data buffer; the data will be copied from that buffer to the stream.
- ◆ A pointer to an integer. **StreamWrite()** will write the number of bytes actually copied to that integer.

If all the data was written successfully, **StreamWrite()** will return zero and write the number of bytes written (i.e. the size of the data buffer passed) to the integer pointed to by the sixth argument. If it could not successfully write all the data, it will return one of the following **StreamError** values:

STREAM_WOULD_BLOCK

STREAM_BLOCK had been passed, and there was no room to write any data to the stream. The sixth argument will be set to zero.

STREAM_SHORT_READ_WRITE

If **STREAM_NOBLOCK** had been passed, this means there was not enough room to write all the data. If **STREAM_BLOCK** had been passed, this means the stream was closed before all the data could be written. The sixth argument will contain the number of bytes actually written to the stream.

STREAM_CLOSING

The stream is in the process of being closed; no writing is permitted while this is happening. The sixth argument will be set to zero.

STREAM_CLOSED

The stream has already been closed. The sixth argument will be set to zero.

You may often want to write a single byte to the stream. There is a special routine to do this, **StreamWriteByte()**. This routine takes four arguments:

- ◆ The **GeodeHandle** of the stream driver.
- ◆ The **StreamToken** of the stream.
- ◆ A member of the **StreamBlocker** enumerated type, as described in section 21.1.2 on page 777.
- ◆ The byte to be written.

21.1

If the byte is written successfully, **StreamWriteByte()** will return zero. Otherwise, it will return one of the following error values:

STREAM_WOULD_BLOCK

STREAM_BLOCK had been passed, and there was no room to write any data to the stream.

STREAM_CLOSING

The stream is in the process of being closed; no writing is permitted while this is happening.

STREAM_CLOSED

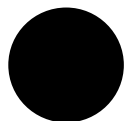
The stream has already been closed.

21.1.4 Reading Data from a Stream

`StreamRead()`, `StreamReadByte()`

To write data into a stream, call the routine **StreamRead()**. This routine takes six arguments:

- ◆ The **GeodeHandle** of the stream driver.
- ◆ The **StreamToken** of the stream.
- ◆ A member of the **StreamBlocker** enumerated type, as described in section 21.1.2 on page 777.
- ◆ The amount of data to be read, in bytes.



- ◆ A pointer to a data buffer; the data will be read from the stream into that buffer.
- ◆ A pointer to an integer. **StreamRead()** will write the number of bytes actually read to that integer.

If the requested amount of data was read successfully, **StreamRead()** will return zero and write the number of bytes read (i.e. the size of the data buffer passed) to the integer pointed to by the sixth argument. If it could not successfully read the requested amount of data, it will return one of the following **StreamError** values:

STREAM_WOULD_BLOCK

STREAM_BLOCK had been passed, and there was no data waiting in the stream. The sixth argument will be set to zero.

STREAM_SHORT_READ_WRITE

If STREAM_NOBLOCK had been passed, this means the stream did not have the amount of data requested. If STREAM_BLOCK had been passed, this means the stream was closed before the requested amount of data could be read. The sixth argument will contain the number of bytes actually read from the stream.

STREAM_CLOSING

The stream is in the process of being closed; no reading is permitted while this is happening. The sixth argument will be set to zero.

You may often want to read a single byte from the stream. There is a special routine to do this, **StreamReadByte()**. This routine takes four arguments:

- ◆ The **GeodeHandle** of the stream driver.
- ◆ The **StreamToken** of the stream.
- ◆ A member of the **StreamBlocker** enumerated type, as described in section 21.1.2 on page 777.
- ◆ A pointer to a byte-sized variable; the data byte read will be written to this variable.

If the byte is written successfully, **StreamReadByte()** will return zero. Otherwise, it will return one of the following error values:

STREAM_WOULD_BLOCK

STREAM_BLOCK had been passed, and there was no data waiting in the stream.

STREAM_CLOSING

The stream is in the process of being closed; no reading is permitted while this is happening.

21.1.5 Shutting Down a Stream

21.1

`StreamClose()`

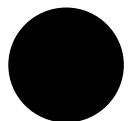
Either the writer or the reader can instigate stream shutdown by calling **StreamClose()**. When one of the two calls this routine, the shut-down process is started; it will not be completed until the other calls the routine.

If the writer calls **StreamClose()**, it may specify that the data already in the buffer be *flushed* (immediately cleared), or that it *linger*. If you specify that the data should linger, the data will be preserved as long as the reader has the stream open. The reader can continue to read data normally until it runs out of data. The last read-operation will most likely return `STREAM_SHORT_READ_WRITE`; after that, all attempts to read data will generate the error `STREAM_CLOSING`. At that point, the reader should call **StreamClose()**. (If the data was flushed by the writer, the next read attempt will return `STREAM_CLOSING`.)

To shut down the stream, call the routine **StreamClose()**. This routine is passed the following arguments:

- ◆ The **GeodeHandle** of the stream driver.
- ◆ The **StreamToken** of the stream.
- ◆ A Boolean value saying whether the data in the stream should be kept until it's read; *false* (i.e. zero) indicates it should be flushed.

If you are using the Serial or Parallel drivers (described later in this chapter), you do not have to coordinate the closure of a stream.



21.1.6 Miscellaneous Functions

`StreamFlush()`, `StreamQuery()`

To flush all the pending (written but unread) data from a stream, call the routine **StreamFlush()**. This routine is passed two arguments:

- ◆ The **GeodeHandle** of the stream driver.
- ◆ The **StreamToken** of the stream.

To find out how much free space is available in a stream, or how much data is waiting to be read, call **StreamQuery()**. This routine is passed four arguments:

- ◆ The **GeodeHandle** of the stream driver.
- ◆ The **StreamToken** of the stream.
- ◆ A member of the **StreamRoles** enumerated type. The only appropriate values here are `STREAM_ROLES_WRITER` (to find the amount of free space available for writing) or `STREAM_ROLES_READER` (to find the amount of data waiting to be read).
- ◆ A pointer to an integer variable.

If the call is successful, **StreamQuery()** returns zero and writes its return value to the fourth argument. If you pass `STREAM_ROLES_WRITER`, **StreamQuery()** writes the number of bytes of free space available in the stream buffer. If you pass `STREAM_ROLES_READER`, **StreamQuery()** returns the number of bytes of data waiting to be read. If the call is unsuccessful, **StreamQuery()** returns a **StreamError**.

21.2 Using the Serial Ports

The serial driver uses streams to control the flow of data to and from serial ports. The kernel automatically copies data from the serial port into one stream for reading, and sends data from another stream into the serial port. An application which wishes to use the serial port simply reads and writes data from those streams.

21.2.1 Initializing a Serial Port

Like the stream driver, the serial driver is not accessed directly from Goc code. Instead, a Goc application makes calls to the Stream Library, which passes the requests through to the Serial Driver's strategy routine. Each serial-port command must be passed the **GeodeHandle** of the Serial Library; again, you can find this handle by calling **GeodeGetInfo()**.

The serial driver uses two streams, one for data going out to the serial port (outgoing) and one for data coming in from the serial port (incoming). Your program is the writer of the outgoing and the reader of the incoming. (In both cases, the port acts as the opposite user.) 21.2

21.2.1.1 Opening a Serial Port

`SerialOpen()`

To open a serial port, call the routine **SerialOpen()**. This routine is passed the following arguments:

- ◆ The **GeodeHandle** of the serial-port driver.
- ◆ A member of the **SerialUnit** enumerated type; this specifies which serial port is being opened. The type's members are SERIAL_COM1, SERIAL_COM2, and so on up to SERIAL_COM8.
- ◆ A member of the **StreamOpenFlags** enumerated type, indicating what to do if the requested serial port is busy (either STREAM_OPEN_NO_BLOCK, indicating that the routine should return an error immediately; or STREAM_OPEN_TIMEOUT, indicating that the routine should wait a specified number of clock ticks to see if the port will free up).
- ◆ The total size of the stream to be used as an input buffer, in bytes.
- ◆ The total size of the stream to be used as an output buffer, in bytes.
- ◆ The maximum number of ticks to wait for the serial port to become available (if STREAM_OPEN_TIMEOUT was passed).

A flag is returned to indicate whether the serial port could be opened; if not, a value of type **StreamError** will be returned to indicate the reason. Possible stream error values are STREAM_BUFFER_TOO_LARGE and

STREAM_CANNOT_CREATE, and the additional values STREAM_NO_DEVICE (if the serial port does not exist) or STREAM_DEVICE_IN_USE (if the device is busy and the **StreamOpenFlags** passed indicate not to wait).

Note that when using the serial driver, you do not identify the stream by a stream token but rather by the serial port number, known as a *unit number*. When accessing a serial port, you simply pass the port's unit number along with either STREAM_READ (if reading from the stream) or STREAM_WRITE (if writing to the stream); because each port has two streams associated with it, you must specify both parameters. The serial driver will understand which stream you are accessing.

21.2.1.2 Configuring a Serial Port

```
SerialSetFormat(), SerialGetFormat(), SerialSetModem(),  
SerialGetModem(), SerialSetFlowControl()
```

Communication using a serial port requires that parity, speed, and flow control be properly set. To control these settings, call **SerialSetFormat()**, passing the following arguments:

- ◆ The **GeodeHandle** of the serial-port driver.
- ◆ The member of the **SerialUnit** enumerated type.
- ◆ A one-byte record of type **SerialFormat**, specifying the parity, word length, and number of stop bits to be used on the serial line; this record is described below.
- ◆ A member of the **SerialMode** enumerated type, set to indicate the level of flow control: SM_COOKED to indicate XON/XOFF flow control with characters stripped to seven bits, SM_RARE to indicate XON/XOFF flow control but incoming characters left alone, or SM_RAW to indicate no flow control.
- ◆ The baud rate to use, a member of the enumerated type **SerialBauds**, which has the following members:

```
typedef enum  
{  
    SERIAL_BAUD_115200    = 1,  
    SERIAL_BAUD_57600    = 2,  
    SERIAL_BAUD_38400    = 3,
```

```

        SERIAL_BAUD_19200      = 6 ,
        SERIAL_BAUD_14400      = 8 ,
        SERIAL_BAUD_9600       = 12 ,
        SERIAL_BAUD_7200       = 16 ,
        SERIAL_BAUD_4800       = 24 ,
        SERIAL_BAUD_3600       = 32 ,
        SERIAL_BAUD_2400       = 48 ,
        SERIAL_BAUD_2000       = 58 ,
        SERIAL_BAUD_1800       = 64 ,
        SERIAL_BAUD_1200       = 96 ,
        SERIAL_BAUD_600        = 192 ,
        SERIAL_BAUD_300        = 384
    } SerialBaud;

```

21.2

SerialFormat is a byte-sized record that specifies the parity, word-length, and number of stop bits for the serial line. The record has the following fields:

SERIAL_FORMAT_DLAB

This is for internal use only; it must be set to zero.

SERIAL_FORMAT_BREAK

If set, this causes a BREAK condition to be asserted on the line. You must explicitly clear this bit again to resume normal operation.

SERIAL_FORMAT_PARITY

This three-bit field holds the parity to expect on receive and use on transmit. It uses the **SerialParity** enumerated type, which has the following members:

```

typedef      enum {
    SERIAL_PARITY_NONE      = 0 ,
    SERIAL_PARITY_ODD       = 1 ,
    SERIAL_PARITY_EVEN      = 3 ,
    SERIAL_PARITY_ONE       = 5 ,
    SERIAL_PARITY_MARK      = 5 ,
    SERIAL_PARITY_ZERO      = 7 ,
    SERIAL_PARITY_SPACE     = 7
} SerialParity;

```



SERIAL_FORMAT_EXTRA_STOP

If this is set, extra stop-bits will be sent. One stop bit is always sent. However, if you set this flag, an extra 1/2 stop bit will be sent if the word-length is 5 bits long; an extra 1 stop bit will be sent if the frame is 6, 7, or 8 bits long.

SERIAL_FORMAT_LENGTH

This two-bit field holds the length of each data word, minus five (i.e. a five-bit word is represented with a zero, a six-bit word with a one).

To find out the current settings of a serial port, call **SerialGetFormat()**. This routine is passed five arguments:

- ◆ The **GeodeHandle** of the serial-port driver.
- ◆ The member of the **SerialUnit** enumerated type.
- ◆ A pointer to a **SerialFormat** variable. **SerialGetFormat()** will write the format data to this variable.
- ◆ A pointer to a **SerialMode** variable. **SerialGetFormat()** will write the appropriate mode constant (SM_COOKED, XON/XOFF, or SM_RARE) to this variable.
- ◆ A pointer to a **SerialBaud** variable. **SerialFormat()** will write the appropriate constant to this variable.

As with other serial port routines, if the routine is successful, it will return zero; if it is unsuccessful, it will return an error code.

If you are using a modem's hardware flow control, you will have to configure the modem appropriately. You can do this by calling **SerialSetModem()**. This routine is passed three arguments:

- ◆ The **GeodeHandle** of the serial-port driver.
- ◆ The member of the **SerialUnit** enumerated type.
- ◆ A record of type **SerialModem**. This record has four fields: SERIAL_MODEM_OUT2, SERIAL_MODEM_OUT1, SERIAL_MODEM_RTS, and SERIAL_MODEM_DTR. Set these fields to indicate how the control-bits should be set.

To find out what flow control is being used, call **SerialGetModem()**. This routine is passed three arguments:

- ◆ The **GeodeHandle** of the serial-port driver.
- ◆ The member of the **SerialUnit** enumerated type.
- ◆ A pointer to a record of type **SerialModem**. **SerialGetModem()** will set this record's SERIAL_MODEM_OUT2, SERIAL_MODEM_OUT1, SERIAL_MODEM_RTS, and SERIAL_MODEM_DTR bits appropriately.

You can also set the flow control without setting the other format options. Do this by calling **SerialSetFlowControl()**. This routine is passed the following arguments:

21.2

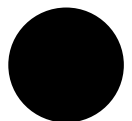
- ◆ The **GeodeHandle** of the serial-port driver.
- ◆ The member of the **SerialUnit** enumerated type.
- ◆ A record of type **SerialModem**. This record has four fields: SERIAL_MODEM_OUT2, SERIAL_MODEM_OUT1, SERIAL_MODEM_RTS, and SERIAL_MODEM_DTR. Set these fields to indicate how the control-bits should be set.
- ◆ A member of the **SerialMode** enumerated type, set to indicate the level of flow control: SM_COOKED to indicate XON/XOFF flow control with characters stripped to seven bits, SM_RARE to indicate XON/XOFF flow control but incoming characters left alone, or SM_RAW to indicate no flow control.
- ◆ A record of type **SerialModemStatus** to indicate which lines (chosen from DCD, DSR, and CTS) should be used to control outgoing data (if hardware flow control is selected). When one of the selected lines is de-asserted by the remote system, the serial driver will not transmit any more data until the state changes.

21.2.2 Communicating

```
SerialRead(), SerialReadByte(), SerialWrite(),
SerialWriteByte(), SerialQuery(), SerialFlush()
```

Communicating with a serial port is very much like using any other stream. Special versions of the stream routines are provided, but they function just like their stream counterparts.

To read data from a serial port, call **SerialRead()** or **SerialReadByte()**. These routines take the same arguments as their **Stream...()** counterparts,



except that each one must be passed the handle of the Serial Driver, not the Stream Driver, and each routine is passed the **SerialUnit** for the appropriate port, instead of being passed a stream token. These routines behave exactly like their **Stream...()** counterparts.

To write data to a serial port, call **SerialWrite()** or **SerialWriteByte()**. Again, these routines behave like their **Stream...()** counterparts, and take similar arguments.

21.2

To find out if you can read or write data to the port, call **SerialQuery()**. Again, this routine behaves like its **Stream...()** equivalent. To flush any data from the input or output stream, call **SerialFlush()**.

21.2.3 Closing a Serial Port

`SerialClose()`, `SerialCloseWithoutReset()`

To close a serial port, call the routine **SerialClose()**. This routine is passed three arguments:

- ◆ The **GeodeHandle** of the serial-port driver.
- ◆ The member of the **SerialUnit** enumerated type.
- ◆ Either `STREAM_LINGER` (to instruct the kernel to close the port after all outgoing data in the buffer has been sent), or `STREAM_DISCARD` (to instruct the kernel to close the port right away and discard all buffered data).

This function returns immediately whether the port was closed right away or not. However, if `STREAM_LINGER` is specified, the port may not be re-opened until all the data in the Serial Port's buffer has been dealt with.

You can also instruct the serial driver to close the stream to a port, without actually resetting the port. Do this by calling **SerialCloseWithoutReset()**. This routine is passed the same arguments as **SerialClose()**.

21.3 Using the Parallel Ports

Using a parallel port is simpler than using a serial port since data goes in only one direction. GEOS does not currently support reading data from a parallel port.

Parallel ports are used primarily for printing, which is handled by the Spool Object Library. The information in this section is useful only to programmers whose applications will need to send data out through the parallel port without using the spooler. Most applications, however, will use the spooler for any and all parallel port use.

21.3

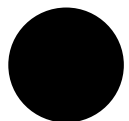
21.3.1 Initializing a Parallel Port

`ParallelOpen()`

To open a parallel port, call the routine **ParallelOpen()**. This routine is passed the following arguments:

- ◆ The **GeodeHandle** of the parallel-port driver.
- ◆ A member of the **ParallelUnit** enumerated type; this specifies which parallel port is being opened. The type's members are `PARALLEL_LPT1`, `PARALLEL_LPT2`, `PARALLEL_LPT3`, and `PARALLEL_COM4`.
- ◆ A member of the **StreamOpenFlags** enumerated type, indicating what to do if the requested parallel port is busy (either `STREAM_OPEN_NO_BLOCK`, indicating that the routine should return an error immediately; or `STREAM_OPEN_TIMEOUT`, indicating that the routine should wait a specified number of clock ticks to see if the port will free up).
- ◆ The unit number of the parallel port in question—this is a value of type **ParallelPortNums**.
- ◆ The total size of the stream to be used as an output buffer, in bytes.
- ◆ The maximum number of ticks to wait for the parallel port to become available (if `STREAM_OPEN_TIMEOUT` was passed).

A flag is returned to indicate whether the parallel port could be opened; if not, a value of type **StreamError** will be returned to indicate the reason.



Possible stream error values are `STREAM_BUFFER_TOO_LARGE` and `STREAM_CANNOT_CREATE`, and the additional values `STREAM_NO_DEVICE` (if the parallel port does not exist) or `STREAM_DEVICE_IN_USE` (if the device is busy and the **StreamOpenFlags** passed indicate not to wait (or not to wait any longer)).

Note that when using the parallel driver, you do not identify the stream by a stream token but rather by the parallel port number, known as a *unit number*. When accessing a parallel port, you simply pass the port's unit number along with either `STREAM_READ` (if reading from the stream) or `STREAM_WRITE` (if writing to the stream); because each port has two streams associated with it, you must specify both parameters. The parallel driver will understand which stream you are accessing.

Once the port is selected, the PC will assert the `SLCTIN` signal, which usually will place the device on-line.

21.3.2 Communicating

```
ParallelWrite(), ParallelWriteByte()
```

Writing to a parallel port is much like writing to any other stream. To write data, call **ParallelWrite()** or **ParallelWriteByte()**. These routines take the same arguments as their **Stream...()** components, except that each one must be passed the handle of the Parallel Driver, not the Stream Driver, and each routine is passed the **ParallelUnit** for the appropriate port, instead of being passed a stream token. These routines behave exactly like their **Stream...()** counterparts.

21.3.3 Closing a Parallel Port

```
ParallelClose()
```

To close a parallel port, **ParallelClose()**. This routine takes the following arguments:

- ◆ The **GeodeHandle** of the parallel-port driver.
- ◆ The member of the **ParallelUnit** enumerated type.

- ◆ Either `STREAM_LINGER` (to instruct the kernel to close the port after all outgoing data in the buffer has been sent), or `STREAM_DISCARD` (to instruct the kernel to close the port right away and discard all buffered data).

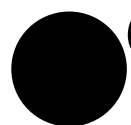
This function returns immediately whether the port was closed right away or not. However, if `STREAM_LINGER` is specified, the port may not be re-opened until all the data in the parallel port's buffer has been dealt with.

21.3

Using Streams

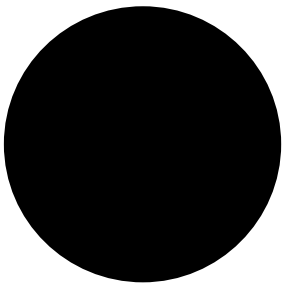
792

21.3



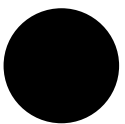
Concepts book

PCCCom Library



22.1	PCCom Library Abilities.....	795
22.2	What To Do.....	795
22.3	Staying Informed	796





The PCCom library provides a simple way to allow a geode to provide and monitor a PCCom connection. If you are familiar with the SDK, you probably think of **pccom** as a tool which allows the target machine to receive commands from the host machine. While the target machine runs the **pccom** tool, the host machine can upload and download files and otherwise manipulate the target machine.

22.1

The PCCom library allows a geode to start up a PCCom thread monitoring a serial port for purposes other than debugging. For instance, it allows the GEOS machine to receive files sent over a serial line by another machine running pccom or another program using pccom's file transfer protocol. This allows file transfers and other pccom operations to go on in the background while the user continues to interact with GEOS.

If you're not familiar with the **pccom** tool, you should probably read the pccom section of "Using Tools," Chapter 10 of the Tools Book—perhaps not the whole section, but at least enough to understand basic usage.

22.1 PCCom Library Abilities

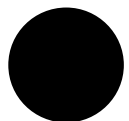
Your geode needn't do much to support PCCom. The PCCom library acts as a passive pccom machine—it will only accept orders from a remote machine.

All you need to do to support PCCom is start up a PCCom process, ideally freeing it when done. Geodes using the PCCom library have the option of receiving notification when pccom would display text.

22.2 What To Do

PCCOMINIT(), PCCOMEXIT(), PCCOMABORT()

When you are ready to start monitoring a serial port for pccom-style communications, call PCCOMINIT(). This routine is an entry point for the library, accessible via **ProcGetLibraryEntry()** as the first entry point in the



library, and it may also be called as a normal routine. PCCOMINIT() starts up a new thread which will monitor the passed serial port. If it cannot make the connection, it will return an error.

When you are done with PCCom, call PCCOMEXIT(), invocable as a library entry point or as a normal routine, which closes down the monitor thread and frees the serial port for other uses. After calling this routine, you must call PCCOMINIT() again if you wish to re-establish the pccom connection.

22.3

You may call the PCCOMABORT() routine at any time; this routine aborts any pccom file transfer that may be in progress, but leaves the PCCom connection intact.

22.3 Staying Informed

The sections above tell you everything you need to let your geode interact with pccom. It is possible to do more: your geode can receive notification whenever the **pccom** tool would display some text. The **pccom** tool displays text to show the user what's going on. Text signals the successful or unsuccessful completion of certain operations; a spinning baton shows that a file transfer is in progress. Your geode can also find out if the machine on the other side of the pccom link has quit pccom.

When calling PCCOMINIT(), your geode can specify an object which should receive notification when pccom has text to display or senses that the other side of the pccom link has quit. If an object is so specified, it will receive notification messages at these times. You must also set the **PCComInitFlags** argument to PCCOMINIT() such that the appropriate kinds of notification will be sent; there is one flag which asks for display text notification and another flag which asks for notification when the other side of the pccom connection exits.

Notification will come in the form of MSG_META_NOTIFY or MSG_META_NOTIFY_WITH_DATA_BLOCK. There are three possible forms of notification:

MSG_META_NOTIFY:GWNT_PCCOM_DISPLAY_CHAR
If the passed notification type is

GWNT_PCCOM_DISPLAY_CHAR, then the notification's data word contains a character that pccom would display.

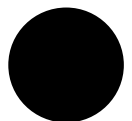
MSG_META_NOTIFY_WITH_DATA_BLOCK:GWNT_PCCOM_DISPLAY_STRING

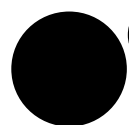
If the passed notification type is GWNT_PCCOM_DISPLAY_STRING, then the data in the notification's data block is a string of characters.

MSG_META_NOTIFY:GWNT_PCCOM_EXIT_PCCOM

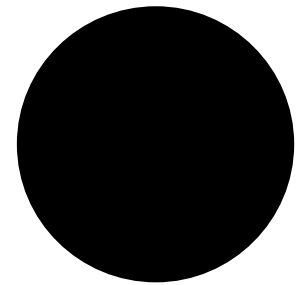
If the passed notification type is GWNT_PCCOM_EXIT_PCCOM, then the pccom process on the other side of the pccom connection has exited; the notification's data word contains a **PCComReturn** type indicating the success the other side had in exiting. The other side of the connection signals that it is exiting by sending an EX escape code over the pccom connection. A data value of PCCRT_NO_ERROR is a sign of a successful exit; other return values might signal that the exit was the result of an error.

22.3





Graphics Environment



23

23.1	Graphics Road Map	801
23.1.1	Chapter Structure	801
23.1.2	Vocabulary	803
23.2	Graphics Goals	806
23.3	Graphics Architecture	807
23.4	How To Use Graphics.....	808
23.5	Coordinate Space	810
23.5.1	Standard Coordinate Space.....	811
23.5.2	Coordinate Transformations.....	812
23.5.2.1	Simple Transformations	814
23.5.2.2	Complicated Transformations	816
23.5.3	Precise Coordinates	818
23.5.4	Device Coordinates	819
23.5.4.1	What the System Draws on the Device	819
23.5.4.2	Converting Between Doc and Device Coordinates.....	822
23.5.5	Larger Document Spaces.....	823
23.5.6	Current Position	824
23.6	Graphics State	825
23.6.1	GState Contents	826
23.6.2	Working with GStates	827
23.7	Working With Bitmaps	828
23.8	Graphics Strings.....	832
23.8.1	Storage and Loading	832
23.8.2	Special Drawing Commands	834
23.8.3	Declaring a GString Statically	836
23.8.4	Creating GStrings Dynamically	840
23.8.5	Drawing and Scanning.....	843



23.8.6	Editing GStrings Dynamically.....	846
23.8.7	Parsing GStrings.....	847
23.9	Graphics Paths	849
23.10	Working With Video Drivers	852
23.10.1	Kernel Routines.....	852
23.10.2	Direct Calls to the Driver.....	853
23.11	Windowing and Clipping.....	854
23.11.1	Palettes	854
23.11.2	Clipping	854
23.11.3	Signalling Updates	855



The GEOS graphics system provides many powerful tools to enhance your geode's appearance and graphical capabilities.

The kernel automatically displays much of what appears on the screen. Generic and Visual objects display themselves, so if these objects can display everything your geode needs, you can probably skip this section for now. On the other hand, geodes with any kind of specialized display needs will need imaging code, as will any geodes that print. 23.1

This, the first chapter of the Imaging Section, explains the thinking behind the graphics system and the work you'll have to get done ahead of time to set up the space you're going to be drawing to. For actual drawing commands, see "Drawing Graphics," Chapter 24.

Before reading this chapter, you should be familiar with the basics of the generic UI and messaging. You may also want to review high school geometry.



23.1

Graphics Road Map

Graphics is a big topic. It has many applications and thus many ways to apply it. If you will be writing graphics-intensive programs, you'll probably end up reading everything in these chapters. Chances are, however, that you'll only need to learn a few things about the graphics system.

This section is for those people who would like to skip around the GEOS imaging system. It contains an outline of the chapter as well as a list of definitions for terms that will be used throughout.

23.1.1

Chapter Structure

This chapter is divided into sections. Which sections you'll want to read depend on what you're interested in.



1 Road Map

You're reading the road map now.

2 Goals

This section goes into the design philosophy behind the GEOS graphics system.

3 Architecture

The Graphics Architecture section describes the basics behind how the graphics system works.

4 How to Use Graphics

This section is for programmers new to GEOS who know what sort of graphics they want their geode to have but don't know how to get it. This section discusses the different contexts in which graphics appear and how you can work within each of those contexts. After going through the section, you may wish to return to this road map to find out where to read more about whatever sort of graphics you'll be working with.

5 Coordinate Space

The GEOS graphics system describes locations and distances using a rectangular grid coordinate system. This section explains how to work with and manipulate this space.

6 Graphics State

This section describes the GState, a data structure used in many contexts and for many purposes throughout the graphics system.

7 Bitmaps

It is possible to specify an image by using a data structure in which an array of color values is used to represent a rectangular area of the screen. This section describes how such a data structure can be manipulated.

8 Graphics Strings and Metafiles

This section describes how to create and draw Graphics Strings, also known as GStrings. (See the Vocabulary section, below, for a simple definition of GString.)

9 Graphics Paths

The graphics system allows you to specify an arbitrary area of the display which can be used in powerful graphics operations. This section explains how to set up a path, a data structure describing the outline of the area you want to specify.

10 Video Drivers

Most programmers can safely skip this section, but video game writers might want to read it. This section includes some advanced techniques for getting faster drawing rates.

11 Windowing and Clipping

This section explains some of the work the graphics system does to maintain windows in a Graphical User Interface. Included is an in-depth look at clipping, along with some commands by which you can change the way clipping works.

23.1

23.1.2 Vocabulary

Several terms will crop up again and again in your dealings with graphics. Many of these terms will be familiar to programmers experienced with other graphics systems, but those not familiar with any of these terms are encouraged to find out what they mean.

GState, Graphics State

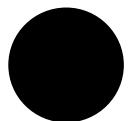
The graphics system maintains data structures called Graphics States, or GStates for short. GStates keep track of how a geode wants to draw things: the colors to use, current text font, scaling information, and so on. Most graphics routines, including all routines that actually draw anything, take a GState as one of their arguments. Thus to draw the outline of a red rectangle, you would first call the **GrSetLineColor()** routine to change the color used for drawing lines and outlines to red. Only after thus changing the GState would you call the **GrDrawRect()** command.

Standard Coordinate Space, Document Coordinate Space

Drawing commands in GEOS use the standard coordinate space, also known as the document coordinate space. This is a rectangular grid used for expressing locations and distances. This grid uses units of 1/72nd of an inch. Thus, drawing a point to (72, 72) will draw the point one inch to the right and one inch below the top left corner of an ordinary Content or document.

GString, Graphics String, Metafile

GString is short for Graphics String. A GString is a data structure representing a sequence of graphics commands. Graphics Strings are used many places in the system. GStrings



are used to pass graphics through the clipboard. They serve as the descriptors of graphical monikers, which are in turn used as application icons. The printing system uses GStrings to describe images to be printed. A graphics Metafile is just a file containing a GString, taking advantage of this data structure to store a graphics image to disk.

23.1

Bitmap

A bitmap is a picture defined as a rectangular array of color values. Note that since display devices normally have their display area defined as a rectangular grid for which each pixel can have a different color value, it is relatively easy to draw a bitmap to the screen, and in fact the graphics system normally works by constructing a bitmap of what is to be displayed and displaying it. Bitmaps are sometimes known as “raster images.”

Path

Paths provide a precise way to specify arbitrary areas of the display area. Like GStrings, they are described by a sequence of drawing commands. Instead of defining a picture, these commands describe a route across the graphics space. The enclosed area can be used in a variety of operations.

Region

Regions provide another approach to describing arbitrary display areas. Regions use a compressed scan-line data structure to represent a pixel-based shape on the display. Though lacking the mathematical precision of paths, region-based operations are very fast and thus ideally suited to certain simple tasks.

Palette, RGB Values

Many display devices can display a wide variety of colors. Of these, most cannot display all their colors at once. Typical configurations can show 16 or 256 colors at a time out of a possible 256K. A palette is one of these subsets of the set of possible colors. When drawing something in color, normally the color to draw with is specified by the palette index, that color's place in the table.

The value stored in each entry is an RGB value, a color described in terms of its red, green, and blue components. Each of these three values ranges between 0 and 255. Geodes can change the RGB values associated with palette entries and thus change the available colors.

Video Driver

These geodes stand between the graphics system and the actual display device. They maintain device independence and do a lot of the “behind the scenes” work of the graphics system.

Windowing, clipping, “marked invalid”

The windowing and graphics systems are heavily intertwined. The graphics system controls how windows should be drawn, while the windows system keeps track of which parts of various displays are visible. For the most part graphics programmers don't have to worry too much about the windowing system, but there are some terms worth knowing. *Clipping* is the process of keeping track of what things drawn to a window are actually visible. If a geode draws something beyond the edge of a window, the system can't just ignore the drawing, as the user might later reveal the drawing by scrolling onto that area. The graphics system “clips” the graphic, being sure to show only those parts that are in the visible part of a window. Further clipping takes place when a window is obscured behind another window. Any drawing on the lower window must not show up until the upper window is moved. The “clipping region” is that area of a window which is visible—anything drawn to the window outside the region will be clipped (see Figure 23-1). Programs can reduce the clipping areas of any of their associated windows.

23.1

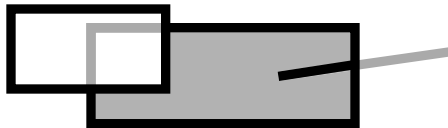


Figure 23-1 *Clipping*

Clipping is the process of determining what needs to be drawn. If something appears outside the bounds of its window or is hidden under another window, the system should not draw it. The clipping region is the area of the display where things will be drawn: in the window, but not obscured. In the figure, the shaded area shows the clipping region.

The windowing system is able to work much more quickly by assuming that not too many things are going to be changing at once. Normally it can assume that a given area of the screen

will look like it did a short while ago. When it's time to change an area of the screen, that area is said to be "marked invalid," since whatever is presently being shown there is no longer valid. The normal example is that if a pull-down menu has been obscuring part of a document, when the menu goes away, the part of the document that becomes visible must be redrawn.

MSG_VIS_DRAW, MSG_META_EXPOSED

23.2

These messages are sent to visible objects to instruct them to redraw themselves; if you are using graphics to display a visible object then you are probably intercepting MSG_VIS_DRAW. MSG_META_EXPOSED lets an object know that it has been marked invalid; either it has to be drawn for the first time, or part or all of it has been exposed (hence the name). The UI controller sends this message to the top object in the Vis hierarchy. MSG_VIS_DRAW, on the other hand, specifically instructs an object to redraw itself. Generally, a Vis object will respond to MSG_META_EXPOSED by sending MSG_VIS_DRAW to itself; it responds to MSG_VIS_DRAW by issuing the appropriate drawing commands, then sending MSG_VIS_DRAW to each of its children.

Graphic Objects

Graphic Objects provide the user with an interface for working with graphics in a manner similar to GeoDraw's. They are useful for programs which allow the users to construct some sort of graphical document or provide a sort of graphical overlay for a spreadsheet.

23.2 Graphics Goals

The graphics system is in charge of displaying everything in GEOS. Thus, it is vital that the GEOS graphics system be both powerful and easy to use. Features such as outline font support and WYSIWYG printing, usually afterthoughts on other operating systems, have been incorporated into the GEOS kernel. The graphics system was designed to be state of the art and thus had to achieve several goals:

- ◆ **Fast Operation**

The GEOS graphics system is heavily optimized for common operations

on low-end PCs. To allow a windowing system to run on an 8088 machine, it is vital that line drawing, clipping, and other common graphical operations run very quickly. They do.

- ◆ **Device Independence**

Applications are sheltered from the hardware. Coordinate system units are device independent so that all of your drawing commands use real-world measurements. These units are then translated into device coordinates by the kernel.

- ◆ **Complete Set of Drawing Primitives**

The graphics system must be able to draw a wide variety of shapes and objects to meet the needs of the UI and applications. It does.

- ◆ **Built In Support for Outline Fonts**

GEOS includes outline font technology supporting a variety of font formats. Outline fonts are text typefaces which are defined by their outline shape rather than by a bitmap representation. Outline fonts can be scaled with no apparent loss of smoothness. GEOS also supports bitmap-based fonts; however the very nature of these fonts makes them non-WYSIWYG.

- ◆ **Single Imaging Model for Screen and Hardcopy**

The graphics system uses the same high level graphics language for screen imaging as for printing. Because the GEOS system creates both screen and printed images through the same language, the screen can display a document just as it will be printed. Except for differences in resolution, what you see is what you get.

23.3

23.3 Graphics Architecture

The graphics system involves many pieces of GEOS working together to turn an application's UI and graphics commands into drawings on a display.

The graphics portion of the kernel automatically makes any requested transformations to the coordinate space dealing with scaling, rotation, etc. It transforms complicated drawing commands into simpler ones supported by the video drivers. (See Figure 23-2.)

The video driver can render these simplified commands directly on the device. The video driver checks for collisions, making sure that if whatever is

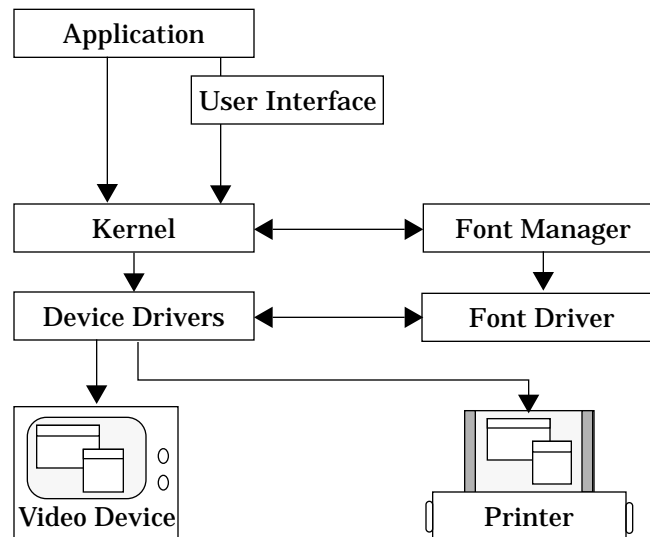


Figure 23-2 *Graphics System Architecture*

being drawn coincides with the mouse pointer, the mouse pointer will be drawn on top. The video driver does clipping so a geode doesn't draw outside its window.

Commands and information flow in this manner from application to the Video Device, though information also flows in the opposite direction, so that the kernel can get information about the display device's resolution, size, and capabilities.

If the geode is going to draw text, the graphics system makes calls to the font manager. If the requested font isn't already in memory, the font manager loads it. The font manager then makes calls to the individual font driver for information about specific characters.

23.4 How To Use Graphics

When looking at the source code of sample applications, it's usually not too hard to pick out the commands that do the actual drawing. Commands with names like **GrDraw...()** generally are self-explanatory. It's not so easy to

pick out the commands that set up an area in which the drawing will take place. Part of the problem is that there are many ways to display graphics; each is well suited for different tasks. This section of the chapter provides some practical knowledge about the various ways to display graphics and which situations are appropriate for each.

When possible, the best way to learn how to perform a graphics action is to look at code which performs a similar action. The sample program presented in “First Steps: Hello World,” Chapter 4 shows a simple graphics environment sufficient for many geodes. If you only need to change what is being displayed (as opposed to how it is displayed), you can work straight from the example, drawing different shapes using commands found in “Drawing Graphics,” Chapter 24. Most basic graphics techniques are used in one sample program or another. By combining and adapting code from the sample programs, you can take care of most simple graphics needs.

23.4

If you can’t find a sample geode to work from, there are several points to consider when deciding what sort of graphics environment to set up.

- ◆ Sometimes the only graphics commands in a geode will be those used to define that geode’s program icon. This is a common enough case that instructions for setting up your geode’s icon are in the program topics section, section 6.2 of chapter 6.
- ◆ Will existing generic UI gadgetry be sufficient for everything you want to display? If you’re writing a utility, it might be. If you’re writing an arcade game, it probably won’t be.
- ◆ If your geode will be displaying graphics, will the user ever interact directly with the graphics? A graphing program might draw a graph based on data the user types in. Such a program could draw the graph but might not actually allow the user to interact with the graph. On the other hand, an art program will probably expect the user to interact with the graphics directly.

Once you’ve figured out just what your geode’s graphical needs are, you’re ready to find out which pieces of graphics machinery are right for you.

For custom graphics that will appear in a view, the content object of the `GenView` must be prepared to issue graphics commands. A common tactic is to create a subclass of **VisContentClass** and let an object of this subclass act as the content for a view. The subclass would very likely have a specialized `MSG_VIS_DRAW`. The `Process` object is another popular choice for the view’s



output descriptor. In this case, the process must be prepared to intercept any messages the view is likely to send, with MSG_META_EXPOSED and MSG_VIS_DRAW of the most interest. Whichever object, process or content, is the content of a view can respond to MSG_META_EXPOSED or MSG_VIS_DRAW by calling kernel graphics routines. For more information on how to use these objects, see “GenView,” Chapter 9 of the Object Reference Book.

23.5

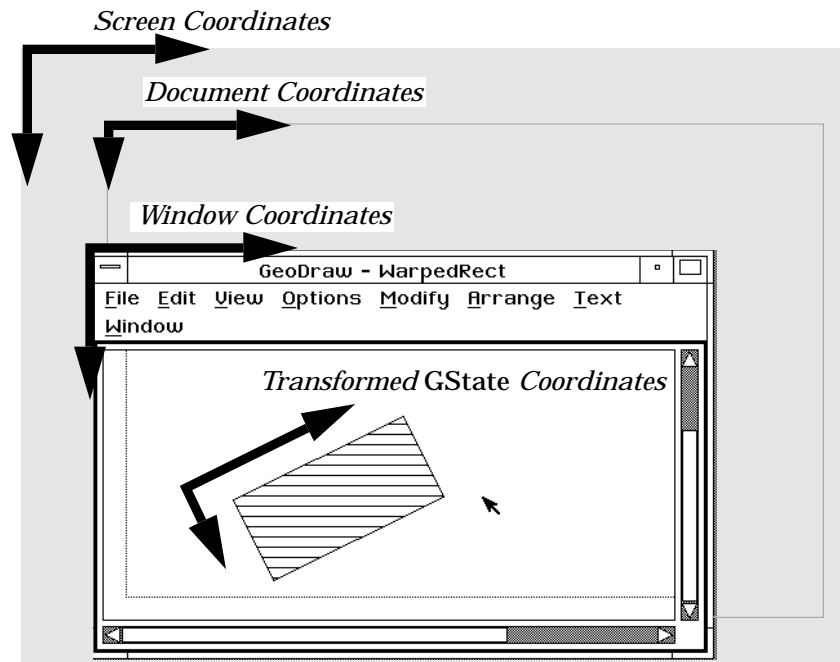
Geodes that wish to allow the user to edit graphical elements would do well to incorporate a Graphic Object into their hierarchies. These objects have considerable power and include UI to allow the user to work graphics within them. See “Graphic Object Library,” Chapter 18 of the Object Reference Book for more information about these object classes.

As you learn more advanced graphics concepts you may discover shortcuts. As you get deeper into graphics, you should keep a cardinal rule in mind. Any time the graphics space is obscured and then exposed, the geode must be able to draw correctly, no matter what changes have been made. If your geode only draws on a MSG_VIS_DRAW, it will automatically follow this rule. However, applications using shortcuts must take MSG_VIS_DRAW into account; it may be sent at any time, and what it draws may wipe out what was there before. An arcade game that moves a spaceship by blitting a bitmap may be fast; however, be sure that the spaceship will be drawn to the right place if the game’s window is obscured and then exposed. Don’t worry if this sounds confusing now, but keep these words in mind as you read on.

23.5 Coordinate Space

The graphics system uses a rectangular coordinate grid to specify the size and position at which drawing commands will be carried out. This is a logical choice as most display devices use a rectangular grid of pixels. “Smart” devices with built in graphics routines tend to use coordinate grids to set place, size, and sometimes movement. “Dumb” devices, capable only of displaying bitmaps, always have these bitmaps described in terms of pixels on a square grid. The GEOS graphics system expects the geode to use the provided device-independent grid to describe graphics commands (see Figure 23-3). The graphics system will then convert this information to

device coordinates: coordinates set up for the specific display device, using a grid of pixels.



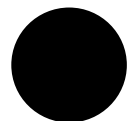
23.5

Figure 23-3 *Coordinate Spaces*

At any one time, the graphics system keeps track of several coordinate spaces. You only have to worry about the coordinate spaces for the GState you are working with; GEOS takes care of the rest.

23.5.1 Standard Coordinate Space

The standard coordinate space is what the application uses to describe where things are to be drawn. It is device independent, based on real world units: typographer's points, about 1/72nd of an inch. The origin of the coordinate system is normally the upper left corner of the document, but this can change with context and with changes made by the kernel or even your geode. The coordinate space normally extends from -16384 to 16383 horizontally and vertically. These constants may be referenced as MIN_COORD and



MAX_COORD. Note that if you're going to be printing your document, you should restrict yourself to coordinates between -4096 to +4096 to account for scaling to draw on high resolution printers. It is possible to define a 32-bit coordinate space, which gives more room but costs speed and memory. For details about 32-bit spaces, see section 23.5.5 on page 823.

Whenever you draw something, you must specify the coordinates where that thing will be drawn. The coordinates you pass specify where in the coordinate plane that thing will be drawn; the plane, in turn, may be translated, scaled, and/or rotated from the standard window coordinate system (see section 23.5.2 on page 812).

The system also maintains a device coordinate system, with a device pixel defined as the unit of the grid. This is the type of coordinate system most programmers are used to, but it is certainly not device independent. The graphics system will do all the worrying about device coordinates so your program doesn't have to. (Note, however, that **GrDrawImage0**, **GrDrawHugeImage0**, and **GrBrushPolyline0** are more device-dependant than most routines; see section 24.2.10 of chapter 24 and section 24.2.8 of chapter 24 for information on these routines).

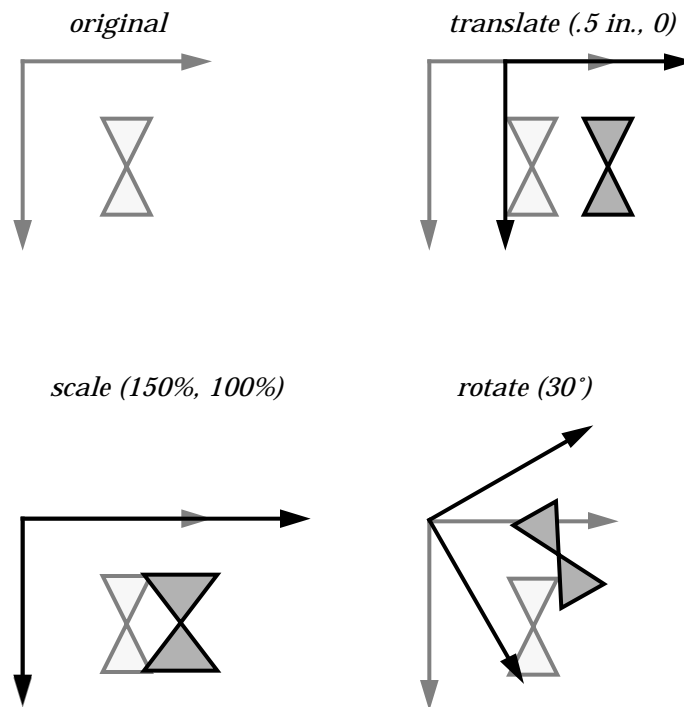
Standard GEOS coordinates depart from the device model, taking an approach closer to a pure mathematical Cartesian plane. Programmers used to working with device-based coordinates are especially encouraged to read section 23.5.4 on page 819 to learn about some of the differences.

23.5.2 Coordinate Transformations

GEOS provides routines which “transform” the coordinate space. These commands can shift, magnify, or rotate the coordinate grid, or perform several of these changes at once (see Figure 23-4). These transformations affect structures in the GState, and new transformations can be combined with or replace a GState's current transformation.

These transformations apply to the coordinate space, not to individual objects. As a result, if you apply a 200% scaling factor to a drawing not centered on the origin, not only will its size change, but its position will change as well. If you want these operations to affect an object but not its position, you should translate the coordinates so that the origin is at the

desired center of scaling or rotation, apply the scaling or rotation, draw the object at the translated origin, then change the coordinates back. For an example of this sort of operation, see Figure 23-5.



23.5

Figure 23-4 *Effects of Simple Transformations*

Since they are stored in the GState, these transformations endure—they do not go away after you’ve drawn your next object. If you apply a 90 degree rotation, you will continue drawing rotated to 90 degrees until you either rotate to some other angle or use another Graphics State. Transformations are also cumulative. If you rotate your space 30°, then translate it up an inch, the result will be a rotated, translated coordinate space. If you want to nullify your last transformation, apply the opposite transformation.

When applying a new transformation to a space which has already been transformed, the old transformations will affect the new one. Be careful, therefore, of the order of your transformations when combining a translation with any other kind of transformation. If you make your transformations in



the wrong order, you may not get what you expected (for an example, see Figure 23-6).

23.5

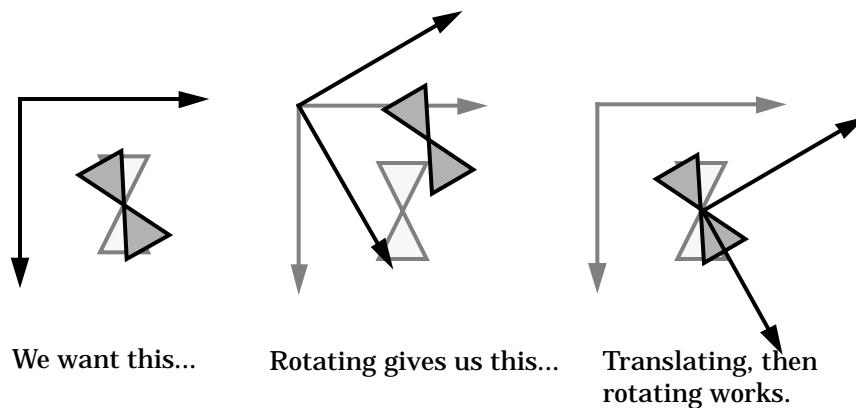


Figure 23-5 *Rotating an Object About Its Center*

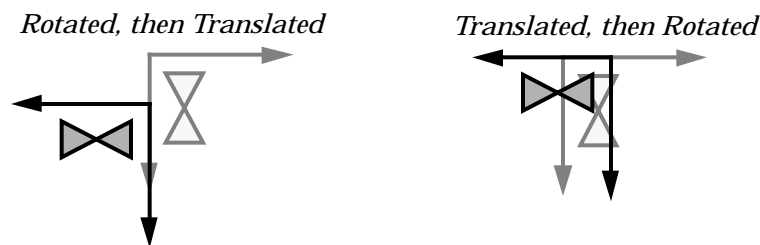
Remember that transformations apply to the whole coordinate space, their effects based around the origin. Thus, to draw a rotated shape at certain coordinates, if the space is rotated before drawing, the coordinates at which the shape will be drawn will be rotated about the origin as well. There is a simple way to draw a shape rotated about an arbitrary axis. First, apply a transformation so that the origin is at the planned center of rotation. Next, apply the rotation. Finally, draw the figure at the new origin.

23.5.2.1 Simple Transformations

```
GrApplyRotation(), GrApplyScale(), GrApplyTranslation(),
GrSetDefaultTransform(), GrSetNullTransform(),
GrInitDefaultTransform(), GrSaveTransform(),
GrRestoreTransform()
```

If you find yourself using transformations at all, they will probably all be rotations, scalings, and translations. The GEOS graphics system includes commands to apply these kinds of transformations to your coordinate system, taking the form **GrApplyTransformation()**. These commands work with a transformation data structure associated with the Graphics State, so

everything drawn in that Graphics State will be suitably transformed. Figure 23-4 on page 813 illustrates the effects of these transformations.



23.5

Figure 23-6 *Ordering Transformation Combinations*

When combining translations with other transformations, remember that the order of the transformations becomes important. The drawing on the left was rotated and then translated. The drawing on the right received the same transformations but in reverse order. Note that the drawings ended up being drawn to different positions.

GrApplyRotation() rotates the coordinate space, turning it counterclockwise around the origin. All objects drawn after the rotation will appear as if someone had turned their drawing surface to a new angle. With a 90° rotation, a shape centered at $(1, 1)$ would draw as if centered at $(1, -1)$. Anything drawn centered at the origin would not change position but would be drawn with the new orientation.

GrApplyScale() resizes the coordinate space; this is useful for zooming in and out. After applying a scale to double the size of everything in the x and y directions, everything will be drawn twice as big, centered twice as far away from the origin. Applying a negative scale causes objects to be drawn with the scale suggested by the magnitude of the scaling factor but “flipped over” to the other side of the coordinate axes.

GrApplyTranslation() causes the coordinate system to be shifted over. After a translation, everything will be drawn at a new position, with no change in orientation or size.

To undo the effects of a transformation, you can apply the opposite transformation: rotate the other way, translate in the opposite direction, or scale with the inverse factor.

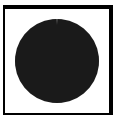
23.5

To undo the effects of all prior transformations, return to the default transformation matrix using the **GrSetDefaultTransform()** command. The routine **GrSetNullTransform()** sets the Graphics State transformation to the null transform—nullifying not only your transformations, but any the system may have imposed as well. For the most part, you should avoid using the **GrSetNullTransform()** command and use the **GrSetDefaultTransform()** instead. You can change the default transformation matrix using **GrInitDefaultTransform()**, but this is generally a bad idea since the windowing system works with the default transformation, and if a geode begins making capricious changes, this can produce strange images.

There are “push” and “pop” operations for transformations. To keep a record of the GState’s current transformation, call **GrSaveTransform()**. To restore the saved transformation, call **GrRestoreTransform()**.

23.5.2.2 Complicated Transformations

`GrApplyTransform(), GrSetTransform(), GrGetTransform(),
GrTransformByMatrix(), GrUntransformByMatrix()`



Advanced Topic

Most programmers will never use these transformations.

You may want to make some change to the coordinate system that has no simple correspondence to scaling, rotation, or translation. Perhaps you know some linear algebra and want to use your knowledge to combine several transformation functions into a single transformation (thus improving execution speed). All transformations on the coordinate system are expressed in the form of transformation matrices. A GEOS graphics system transformation consists of a matrix containing 6 variables and 3 constants (see Equation 23-1). The six variables allow for standard coordinate transformations. The constants (0, 0, and 1 respectively) allow these transformation matrices to be composed. For example, multiplying a scaling matrix with a rotation matrix creates a matrix which represents a combined scaling and rotation. The six variable matrix elements are stored in a **TransMatrix** structure.

The GEOS system uses one matrix to store the Graphics State transformation and one to store the Window transformation. When told to apply a new transformation, the graphics system constructs a matrix to represent the requested transformation change and multiplies this matrix by the old

$$\begin{pmatrix} x_t & y_t & 1 \end{pmatrix} = \begin{bmatrix} T_{11} & T_{12} & 0 \\ T_{21} & T_{22} & 0 \\ T_{31} & T_{32} & 1 \end{bmatrix} \begin{pmatrix} x & y & 1 \end{pmatrix}$$

$$\begin{pmatrix} x_t & y_t & 1 \end{pmatrix} = (T_{11}x + T_{21}y + T_{31}, T_{12}x + T_{22}y + T_{32})$$

Equation 23-1 Transformation Matrices

23.5

The left hand side shows the new coordinate pair resulting from applying the transformation to the old coordinates. The right hand side of the equation shows the formula used to compute these new coordinates. The "1" following each coordinate pair is a constant to allow matrix multiplications.

transformation matrix. To combine these matrices, GEOS multiplies them together to get the cross-product (See Equation 23-2).

$$P \times Q = \begin{bmatrix} P_{11} & P_{12} & 0 \\ P_{21} & P_{22} & 0 \\ P_{31} & P_{32} & 1 \end{bmatrix} \times \begin{bmatrix} Q_{11} & Q_{12} & 0 \\ Q_{21} & Q_{22} & 0 \\ Q_{31} & Q_{32} & 1 \end{bmatrix}$$

$$P \times Q = \begin{bmatrix} P_{11}Q_{11} + P_{12}Q_{21} & P_{11}Q_{12} + P_{12}Q_{22} & 0 \\ P_{21}Q_{11} + P_{22}Q_{21} & P_{21}Q_{12} + P_{22}Q_{22} & 0 \\ P_{31}Q_{11} + P_{32}Q_{21} + Q_{31} & P_{31}Q_{12} + P_{32}Q_{22} + Q_{32} & 1 \end{bmatrix}$$

Equation 23-2 Combining Transformations

Transformations are combined by taking the cross product of their matrices. In cases where order is important, the leftmost factor represents the more recent transformation.

If you know that there's a particular combination of transformations you're going to be using a lot, you can do some math in advance to compute your own transformation matrix, then apply the raw matrix as a transformation using **GrApplyTransform()**. Equation 23-3 shows the matrices corresponding to the simple transformations. To replace the GState's current transformation matrix with the matrix of your choice, use **GrSetTransform()**. To find out the current transformation, call **GrGetTransform()**.



Scaling

$$\begin{bmatrix} Scale_x & 0 & 0 \\ 0 & Scale_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation

$$\begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Translation

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Trans_x & Trans_y & 1 \end{bmatrix}$$

23.5

Equation 23-3 Matrices for Standard Transformations

GrTransformByMatrix() returns the results of transforming a passed coordinate pair using an arbitrary transformation matrix.

GrUntransformByMatrix() performs the reverse operation, returning the coordinates which would have transformed to the passed coordinates.

Sometimes you have to be careful about the order in which these transformations are supposed to occur. When multiplying transformation matrices together, the transformation that is applied later is the first in the multiplication pair. You can combine any number of rotations and scalings together without having to worry about order: the resulting matrices will be the same. When combining a translation with any other kind of operation, it makes a difference what order you make the transformations and thus makes a difference based on what order you multiply the matrices (See Figure 23-6 on page ● 815 and Equation 23-2 on page ● 817).

23.5.3 Precise Coordinates

As has been previously stated, coordinates are normally given in typographer's points. Most graphics commands accept coordinates accurate to the nearest point. This should be more than sufficient for most geodes, but for those specialized programs, more precise drawing is possible.

For simple cases, it is possible to create precise drawings by scaling the drawings. To make drawings accurate to one fourth of a point, for example, scale by 25% and multiply all coordinates accordingly. However, this approach is limited and may result in confusing code.

Another way to make precise drawings is to use the graphics commands which have been specially set up to take more precise coordinates. These commands will not be described in detail here, but keep them in mind when

planning ultra-high resolution applications. **GrRelMoveTo()**, **GrDrawRelLineTo()**, **GrDrawRelCurveTo()**, and **GrDrawRelArc3PointTo()** take **WWFixed** coordinates, and are thus accurate to a fraction of a point. To draw a precise outline, use these commands to draw the components of the outline. To fill an area, use the precise drawing commands to describe the path forming the outline of the area, then fill the path.

23.5

23.5.4 Device Coordinates

Most programmers can work quite well within the document space regardless of how coordinates will correspond to device coordinates. However, some programmers might need to know about the device coordinates as well. The system provides clever algorithms for going from document to device space for all programmers, as well as routines to get device coordinate information from the device driver.

23.5.4.1 What the System Draws on the Device

Consider a device whose pixels are exactly $1/72$ nd of an inch, such that no scaling is required to map document units to device units. The relationship of the coordinate systems is illustrated below. Note that a pixel falls between each pair of document units. This is a further demonstration of the concept that document coordinates specify a location in the document coordinate space, not a pixel.

Next consider a device that has a resolution of 108 dpi, which is 1.5 times greater than our default 72 dpi. That is, there are 1.5×1.5 pixels on the device for each square document unit. The basic problem here is that the coordinates that are specified in document space map to non-integer pixels in device space. The graphics system would like the pixels to be half-filled along two edges of the rectangle (see Figure 23-8). Unfortunately, a pixel must be either filled or empty, so the system needs a set of rules to follow in this situation. These rules are

- ◆ If the midpoint of a pixel (i.e., the device coordinate of that pixel) falls inside the area's boundary, that pixel is filled.

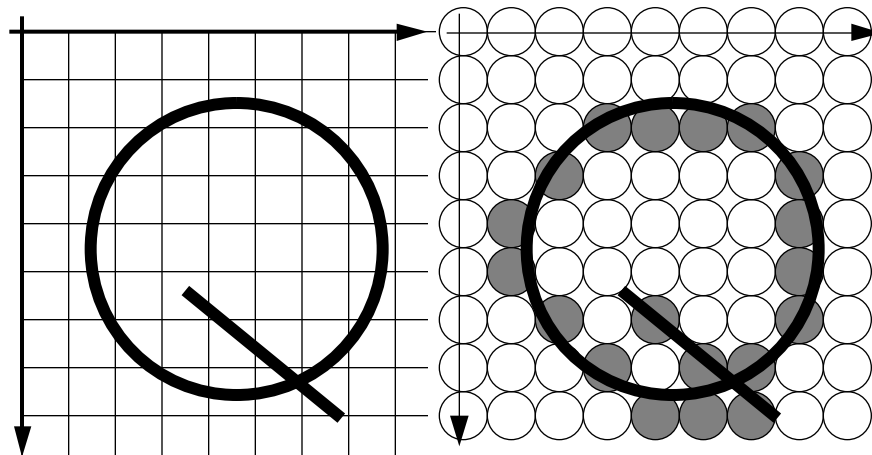


Figure 23-7 *Document and Device Coordinates*

When using a device with 72dpi resolution, it's pretty simple to figure out what will be drawn to the device.

- ◆ Conversely, if the midpoint of a pixel falls outside the area's border, the pixel is not filled.
- ◆ If the midpoint of the pixel falls exactly on the border of the area to be filled, the following rule is used:
 Pixels on the left or the top are not filled;
 Pixels on the right or the bottom are filled;
 Pixels in the left-bottom and top-right corners are not filled.

These rules might seem a little odd: Why not just fill all the pixels that would be touched by the area? One of the problems with this approach is that areas that did not overlap in the document space would overlap on the device. Or more specifically, they would overlap only on some devices (depending on the resolution), which is even worse. The rules have the property that adjoining areas in document space will not overlap in any device space.

Our next set of potential problems comes with lines. Lines can be very thin and thus might be invisible on some low-resolution devices. If the graphics system used the rules for filled objects then some thin lines would be only partially drawn on low resolution devices. GEOS uses Bresenham's algorithm for drawing straight thin lines, ensuring that a continuous group of pixels will be turned on for a line (see Figure 23-9). This continuity is insured due

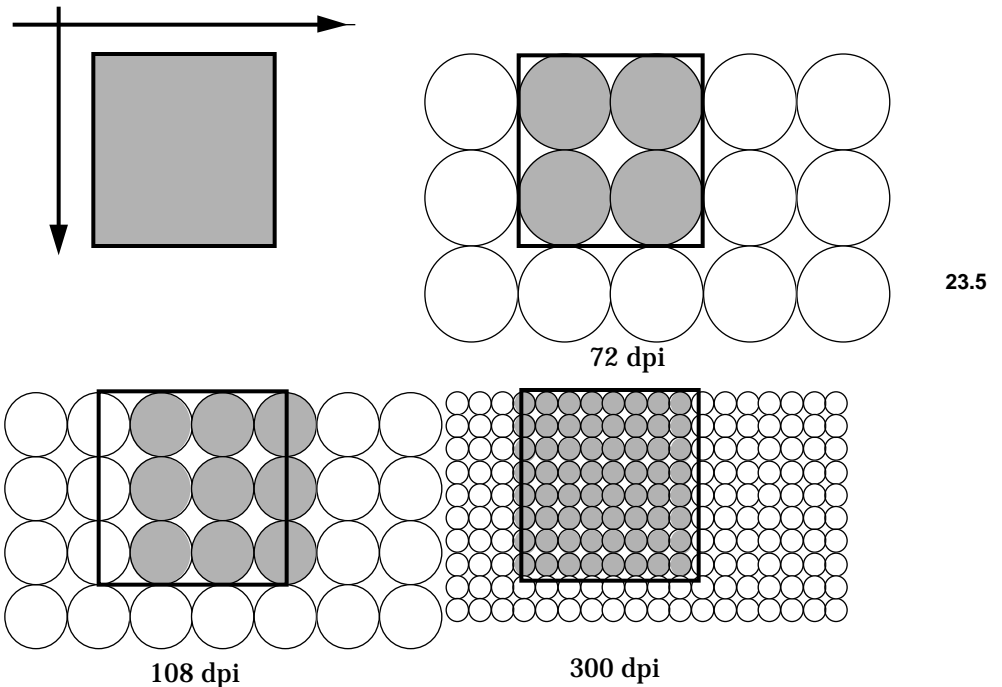


Figure 23-8 *Different Device Coordinates*

When using a device with a resolution that is not a multiple of 72dpi, GEOS has to decide which borderline pixels have to be filled.

- ◆ If the line is more horizontal than vertical, exactly one pixel will be turned on in each column between the two endpoints.
- ◆ If the line is more vertical than horizontal, exactly one pixel will be turned on in each row.
- ◆ If the line is exactly 45 degrees, exactly one pixel will be turned on in each column and row.

Since ellipses and Bézier curves are drawn as polylines, Bresenham's algorithm will work with them.



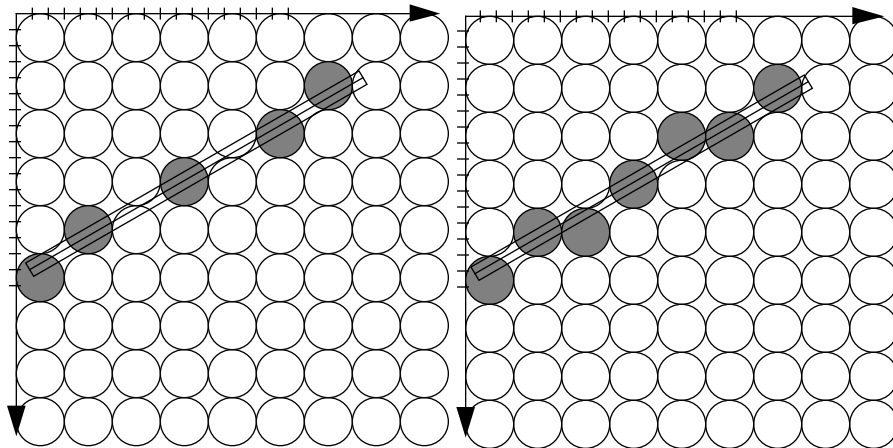


Figure 23-9 *Drawing Thin Lines*

If the thin line were drawn as if it were a skinny filled polygon, there would be gaps as shown to the left. Using Bresenham's line algorithm, GEOS eliminates the gaps, resulting in a continuous line drawn as on the right.

23.5.4.2 Converting Between Doc and Device Coordinates

```
GrTransform(), GrUntransform(), GrTransformWWFixed(),
GrUntransformWWFixed()
```

Given a coordinate pair, at times it's convenient to know the corresponding device coordinates. Sometimes the reverse is true. Use these functions to convert a coordinate pair to device coordinates or vice versa. **GrTransform()** takes a coordinate pair and returns device coordinates. **GrUntransform()** does the reverse. If you want to be able to get a more exact value for these coordinates you can use **GrTransformWWFixed()** and **GrUntransformWWFixed()**. These return fixed point values so you can do other math on them before rounding off to get a whole number that the graphics system can use.

To transform points through an arbitrary transformation instead of to device coordinates, use the **GrTransformByMatrix()** or **GrUntransformByMatrix()** routines, described previously.

23.5.5 Larger Document Spaces

```
GrApplyDWordTranslation(), GrTransformDWord(),
GrUntransformDWord(), GrTransformDWFixed(),
GrUntransformDWFixed()
```



Advanced Topic

Most programmers will only need standard coordinate spaces.

Some applications may need a graphics space larger than the 19 foot square of the standard coordinate space (10 foot square for printed documents).

Some spreadsheets can take up a lot of room, as can wide banners. Geodes that will need a large drawing space can request a 32-bit graphics space, i.e. a coordinate space in which each axis is 2^{32} points long (more than 4 billion points, or 900 miles). Most applications will be able to get by with standard 16-bit coordinate spaces, and their authors may safely skip this section. Note, however, that the Graphic and Spreadsheet objects use 32-bit graphics spaces for display.

23.5

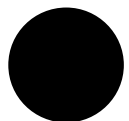
If a geode will use a 32-bit space in a view, it will need a VisContent object with the VCNA_LARGE_DOCUMENT_MODEL attribute.

For the most part, 32-bit documents use the same graphics commands as regular 16-bit documents. No single drawing command can cover more than a standard document area; each has been optimized to draw quickly in 16 bits. To draw to the outlying reaches of your document space, apply a special translation that takes 32-bit coordinates, then use normal drawing routines in your new location.

A model for working with 32-bit graphics spaces commonly used with spreadsheets is to break the graphics space into sections. To draw anything, the geode first uses a translation to get to the proper section, then makes the appropriate graphics calls to draw within that section. Similarly a GeoDraw-style application might translate to the appropriate page, then make normal graphics commands to draw to that page (see Figure 23-10).

If you wish to display this 32-bit coordinate graphics space to the screen, you'll probably want to do so in a 32-bit content.

The standard graphics commands can only draw to one 16 bit space at a time. You will need to translate the coordinate space to choose which 16-bit portion of the 32-bit space you want to draw to. To do so you need to use some special translation functions. **GrApplyDWordTranslation()** corresponds to the **GrApplyTranslation()** function normally used. You can use this routine to



make the jumps necessary to access far away portions of the graphics space. Since a coordinate can now be in a much larger area than before, all routines that deal with a point's position have 32-bit equivalents.

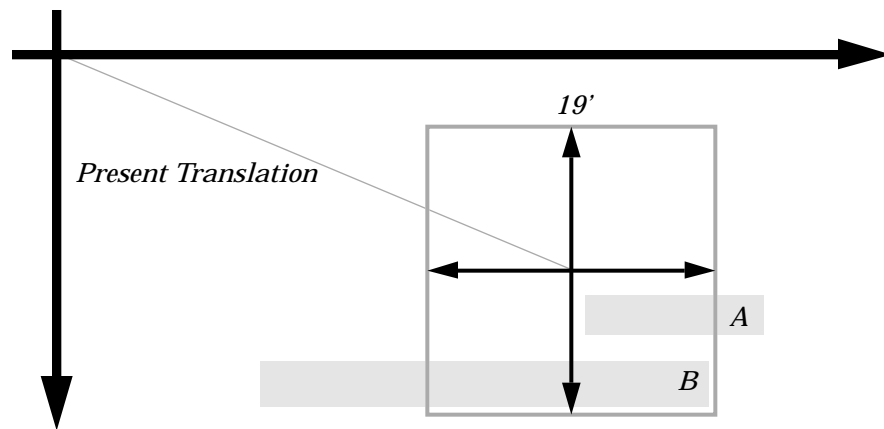


Figure 23-10 *Thirty-two Bit Graphics Spaces*

When drawing in a 32-bit space, remember that the drawing commands are only prepared to draw in a sixteen-bit subspace. In the figure, rectangle A isn't entirely within reach of the origin, so it will be necessary to translate the origin again before drawing this rectangle. Rectangle B is not only partially out of reach of the origin, it's too big to fit inside of a 16-bit space. In order to draw this rectangle, you would have to divide it into two pieces.

GrTransformDWord() and **GrUntransformDWord()** take the place of **GrTransform()** and **GrUntransform()**. **GrTransformDWFixed()** and **GrUntransformDWFixed()** take the place of **GrTransformWWFixed()** and **GrUntransformWWFixed()**, with two words in the integer part of the number instead of one.

23.5.6 Current Position

`GrMoveTo()`, `GrRelMoveTo()`, `GrGetCurPos()`,
`GrGetCurPosWWFixed()`

The graphics system supports the notion of a current position, sometimes called a pen position. Note that the pen position metaphor predates “pen computing”; please don't confuse these concepts. The concept behind the pen

position is that all graphics commands are executed by a pen that ends up at the last place drawn. So if the last command was to draw a curve ending at (20, 20), the pen will still be there. There are special drawing commands that work with the pen position, so you could then draw a line from the current position to (30, 30). The line would extend from (20, 20) to (30, 30), and the pen would then be at (30, 30).

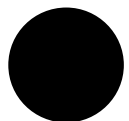
Calling **GrMoveTo()** moves the pen position to coordinates of your choice. **GrRelMoveTo()** moves the pen to a location relative to its present position. **GrRelMoveTo()** takes coordinates accurate to a fraction of a point, allowing for very precise placement. Call **GrGetCurPos()** to get the current position, **GrGetCurPosWWFixed()** to do so with greater accuracy.

23.6

There are some guidelines to follow when figuring out where your pen position is going to end up after a draw operation. If you call the **GrDrawLineTo()** procedure, the pen will end up at the point drawn to. If you call any draw/fill procedure that takes two points as arguments, the pen position usually ends up at the second point passed; if it does not, the reference entry will note where the pen position ends. When text draws at the current position, the current position is used to place the left side of the text. When done drawing the text, the current position has moved to the right side of the text.

23.6 Graphics State

The data structure known as the Graphics State, or GState, keeps track of changes your code makes about how it wants to draw things. If your geode changes its transformation, everything you draw from that point will be drawn transformed until you change the transformation again. If your geode sets the line width to two points, all lines to come will be drawn two points wide until the width is set to something else. Many graphics routines ask that you pass a GState along as one of the parameters so they know where and how to draw what you've requested.



23.6.1 GState Contents

Sometimes it's convenient to think of the GState as being analogous to the Properties boxes in GeoDraw. The GState keeps track of how the program wants to draw lines, text, and filled areas, just as the Line Properties, Text Properties, and Area Properties boxes keep track of how the GeoDraw user wants to draw these objects.

23.6 While handy, this analogy doesn't do the GState justice. The GState keeps track of many things:

- ◆ **Mix Mode**
The graphics system allows for different mix modes (sometimes known as copy modes). These drawing modes permit the geode to draw in such ways that it erases instead of drawing, draws using the inverse of whatever it's drawing on, or uses various other modes.
- ◆ **Current Position (also known as Pen Position)**
The graphics system keeps track of the position of the last thing drawn. If your geode wants to draw something else at this location, there are drawing commands that will draw at the current position.
- ◆ **Area Attributes**
The GState contains information that the graphics system will use when filling areas. This information includes the color and fill pattern to use.
- ◆ **Line attributes**
The graphics system uses information stored in the Graphics State to keep track of what color and pattern to use when drawing lines. The GState also keeps track of whether lines should be drawn as dotted, and if so what sort of dot-dash pattern to use. The GState contains the line width. It also contains line join information, which controls how lines will be drawn when they meet at an angle, as at the vertices of a polygon.
- ◆ **Text Attributes**
The Graphics State contains the ID of the current font, the font size, and the text style. It also contains a color and pattern to use, information about the font, and some esoteric text-drawing options.
- ◆ **Coordinate Space Transformations**
The GState keeps track of the current coordinate space transformation. If there is an associated window, the window's transformation matrix is maintained separately with the window's information.

- ◆ **Associated Window**
The GState knows the handle of the window associated with the GState. This is the window that will determine where drawings appear and how they are clipped.
- ◆ **Associated GString**
If the application is building a GString, the GState is aware of it. The GState contains a reference to the GString. When the graphics system turns graphics commands into GString elements, these elements will be appended to the referenced GString.
- ◆ **Associated Path**
If the application is building a path, the GState contains a pointer to the path along which new path elements are passed, similar to the way GString elements are passed. The GState also keeps track of whether the current path is to be combined with another path or should be used “as is.”
- ◆ **Clipping Information**
The GState keeps track of clipping information in addition to that maintained by the window.

23.6

The function of many of these parts may be fairly intuitive to someone used to working with graphics programs. Some of the others may require additional explanation, especially when it comes to how to work with them.

23.6.2 Working with GStates

```
GrCreateState(), GrDestroyState(), GrSaveState(),
GrRestoreState()
```

As has been mentioned, most graphics routines require that a GState handle be provided as an argument. Beginning programmers are often unclear on just where to get the GState to use.



If you're creating a GState in a MSG_VIS_DRAW handler, you probably shouldn't be.

Many drawing routines are called by MSG_VIS_DRAW. This message provides a GState, and all routines in the handlers for this message should use the provided GState. Creating a new GState under these circumstances is unnecessary and wasteful. However, sometimes you will need to create a GState. **GrCreateState()** creates a GState with the default characteristics. You must specify the window with which the GState will be associated.



Commands which change drawing attributes or the current position change the GState.

GrDestroyState() is used to get rid of a GState, freeing the memory to be used by other things. If GStates are created but not destroyed, eventually they will take too much memory. Normally, for each call to **GrCreateState()** there is a corresponding **GrDestroyState()**. MSG_VIS_DRAW handlers don't need to destroy the passed GState. Graphics states are cached so that **GrCreateState()** and **GrDestroyState()** don't normally need to call **MemAlloc()** or **MemFree()**. When GStates are freed, their space is added to the cache. When the memory manager needs to find space on the heap, it flushes the cache.

A geode is most likely to call **GrCreateState()** when about to draw a piece of geode-defined UI. Other than that, you'll probably be using GStates provided to you by the system. You might want to create a GState if you wanted to calculate something (perhaps the length, in inches, of a text string) when you had no appropriate GState.

GrSaveState() provides a sort of "push" operation that works with GStates. When you call certain functions, like **GrSetAreaColor()**, new values will wipe out the values of the old GState. But if you've previously called **GrSaveState()**, then any time you call **GrRestoreState()** on your saved state, it will come back and displace the current state. Your application can save a GState to save a commonly used clipping region, which could then be restored by restoring the state instead of repeating all the operations needed to duplicate the region. **GrSaveTransform()** and **GrRestoreTransform()** are optimizations of **GrSaveState()** and **GrRestoreState()**, but they only preserve the GState's transformation.

23.7 Working With Bitmaps

`GrCreateBitmap()`, `GrDestroyBitmap()`, `GrEditBitmap()`
`GrGetPoint()`, `GrSetBitmapMode()`, `GrGetBitmapMode()`,

```
GrSetBitmapRes(), GrGetBitmapRes(), GrClearBitmap(),  
GrGetBitmapSize(), GrCompactBitmap(), GrUncompactBitmap()
```

Bitmaps are useful for describing complicated pictures that don't have to be smooth at all resolutions. For example, coming up with all the lines and shapes necessary to describe a complicated photograph would be time-consuming and a waste of memory. It's much easier to set up a rectangular array of cells and to set a color value for each cell. Bitmaps are often used for defining program icons. GEOS includes a great deal of bitmap support. It has kernel routines to create, modify, and draw bitmaps.

23.7

There are three main ways to create a bitmap for an application to use. One often used is to embed the data of a desired bitmap directly into a graphics string. This is the way normally used for defining system icons. The other common way is to call the kernel graphics routine **GrCreateBitmap0**. Another way to create a bitmap, not used so often, is to manipulate memory directly: The formats used for describing bitmaps are public, and though it would be easier in most cases to work through **GrCreateBitmap0**, those with specialized needs might want to create their bitmap data structures from scratch.

The **GrCreateBitmap0** routine creates an offscreen bitmap and returns a Graphics State which can be drawn to; changes to this Graphics State become changes to the offscreen bitmap. For example, calling **GrDrawLine0** and passing the Graphics State provided with such a bitmap would result in a bitmap depicting a line. To display this bitmap, call the **GrFillBitmap0**, **GrDrawHugeBitmap0**, **GrDrawImage0**, or **GrDrawHugeImage0** commands in another graphics space (see section 24.2.10 of chapter 24).

When creating a bitmap, you must make choices about what sort of bitmap you want. Depending on your choices, GEOS will be able to use a variety of optimizations. For instance, it takes much less room to store a monochrome bitmap than a color bitmap the same size.

Whenever creating a bitmap, you must specify its dimensions and what sort of coloring it will use (monochrome, four bit, eight bit, or 24 bit).

You may store a mask with your bitmap. This mask works something like the mask used when drawing a view's mouse pointer. When the bitmap is drawn, blank areas of the bitmap will be drawn as black for those pixels where the



mask is turned on. For pixels where the mask is turned off, whatever was underneath the bitmap will be allowed to show through.

By asking for a Complex bitmap, you can specify even more information. Complex bitmaps may include their own palette and may specify their own horizontal and vertical resolution. They are very useful for working with bitmaps that may have been captured on other systems or for working with display devices.

23.7

The **GrDestroyBitmap()** routine destroys some or all of the information associated with a bitmap. You may use this function to free all memory taken up by the bitmap. You may also use this function to free only the GState associated with a bitmap by **GrCreateBitmap()**, but leave the bitmap's data alone; the **BMDestroy** argument will determine exactly what is destroyed. This usage comes in handy for those times when a bitmap will not be changing, but will be drawn. Large bitmaps are stored in HugeArrays so they won't take up inordinate amounts of RAM; of course it's always wise to free the memory associated with a bitmap when that bitmap is no longer needed.

If you have freed the GState associated with a huge bitmap using **GrDestroyBitmap()** but want to make changes to the bitmap, all is not lost. Call **GrEditBitmap()** to associate a new GState with the bitmap. Be careful, however; the bitmap will not recall anything about the old GState, so you must set up colors, patterns, and other such information again. To update the VM file used to store a bitmap (if any), call **GrSetVMFile()**.

GrClearBitmap() clears the data from a bitmap.

GrGetPoint() can retrieve information from a bitmap, returning its color value for some location. It works with all sorts of display areas, not just bitmaps. It is mentioned here because of its usefulness for those who wish to be able to exercise effects on their bitmaps.

GrSetBitmapMode() gives you control over how drawing commands will affect the bitmap. If your bitmap has a mask, use this routine to switch between editing the mask and the bitmap itself; the **BitmapMode** argument will specify what is to be edited.



Advanced Topic

Bitmaps are simple, but the following routines do advanced things with them.

This routine also gives you control over how monochrome bitmaps should handle color. When you draw something in color to a monochrome bitmap, the system tries to approximate the color using dithering. It turns some pixels on and some pixels off in an attempt to simulate the color's brightness. A

crimson area would appear with most pixels black, and thus rather dark. A pink area on the other hand would have mostly white pixels, and thus appear light. **GrSetBitmapMode()** can change which strategy GEOS will use when deciding which pixels to turn on. If you wish to use “dispersed” dithering, the pixels turned on will be spread out evenly, resulting in a smooth gray. However, some output devices (notably certain printers) have trouble drawing small, widely spaced dots accurately. Using “clustered” dithering causes the system to keep the pixels turned on close together, resulting in pictures reminiscent of newspaper photographs. Thus, devices that prefer a few big dots to lots of little dots will have an easier time with bitmaps so edited.

23.7

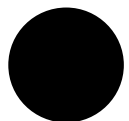
The bitmap mode information may also include a color transfer table. These tables are normally only used by printer drivers, but your geode is welcome to use them, if you can find a reason. Some models of printer have problems when mixing colors. Several printers have problems trying to mix dark colors, tending to end up with black. The table contains a byte value for each possible value of each color component. When mixing the colors, the graphics system will find the real color value to use in the table. Thus, if the reds in RGB bitmaps are looking somewhat washed out, you might set up a table so that reds would be boosted. `RGBT_red[1]` might be 16, `RGBT_red[2]` 23, so that the device would use more than the standard amount of the red component.

GrGetBitmapMode() returns the current bitmap editing mode.

GrSetBitmapRes() works with complex bitmaps, changing their resolution. Because simple bitmaps are assumed to be 72 dpi, their resolution cannot be changed unless they are turned into complex bitmaps. **GrGetBitmapRes()** returns the resolution associated with a complex bitmap.

GrGetBitmapSize() returns a bitmap's size in points. This function might be useful for quickly determining how much space to set aside when displaying the bitmap. **GrGetHugeBitmapSize()** retrieves the size of a bitmap stored in a **HugeArray**.

Use **GrCompactBitmap()** and **GrUncompactBitmap()** to compact and uncompact bitmaps. Compacted bitmaps take up less memory; uncompact bitmaps draw more quickly. Note that the bitmap drawing routines can handle compacted and uncompact bitmaps. These functions are here to aid programmers who wish more immediate control over their memory usage.



23.8 Graphics Strings

A GString is a data structure which represents a series of graphics commands. This structure may be stored in a chunk or VM file so that it may be played back later. An application may declare a GString statically or may create one dynamically using standard kernel drawing commands. They are used for describing application icons and printer jobs, among other things.

23.8

23.8.1 Storage and Loading

```
GrCreateGString(), GrDestroyGString(), GrLoadGString(),  
GrEditGString(), GrCopyGString(), GrGetGStringHandle(),  
GrSetVMFile()
```

GStrings may reside in a number of types of memory areas. Depending on the GString's storage, you will have to do different things to load it. One common case we have already discussed to some extent is when the GString is part of a visual moniker. In this case, the gstring will be stored in the *gstring* field of the **@visMoniker**'s implied structure. In this case the UI will do all loading and drawing of the GString.

The GString data itself consists of a string of byte-length number values. The graphics system knows how to parse these numbers to determine the intended drawing commands. You need not know the details of the format used—there are routines by which you may build and alter GStrings using common kernel graphics routines; however, macros and constants have been set up so that you may work with the data directly.

GString data may be stored in any of the following structures (corresponding to the values of the **GStringType** enumerated type):

Chunk This is the storage structure of choice for GStrings which will be used as monikers.

VM Block Virtual memory is normally used to store GStrings which may grow very large. GStrings residing in virtual memory may be dynamically edited.

Pointer-Referenced Memory

You may refer to GStrings by means of a pointer. However, this

will only work for reading operations (i.e. you may not change the GString). This is the ideal way to reference a GString which is statically declared in a code resource.

Stream Streams are not actually used to store data—they are used to transmit it between threads or devices. If you write a GString to a stream, it is assumed that some other application, perhaps on another device, will be reading the GString.

Note that a GString stored in a Stream, VM block, or in memory referenced only by a pointer is not quite ready to be drawn, only GStrings stored in a chunk may be drawn. Fortunately there is a routine which can load any type of GString into local memory so that it may be drawn.

23.8

If you are editing or creating the GString dynamically, it will have a GState associated with it. Any drawing commands made using this GState will be appended to the GString. This GState will not be stored with the GString; it is instead stored with the other GStates. You may destroy the GState when done editing, and hook up a new one if starting some other edit; this will not affect the GString's storage.

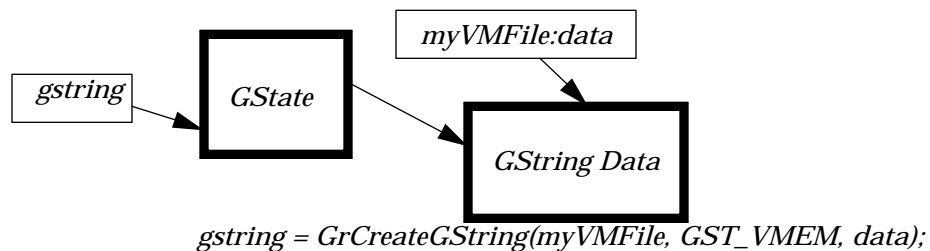
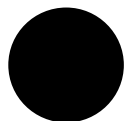


Figure 23-11 *GStrings and Associated GStates*

The GString's data may reside in a variety of places. In addition to this data, many GString operations will involve a GState. This special GState may be used to edit the contents of the GString. Don't confuse the GString's handle with that of its associated data.

To dynamically create an empty GString, call the **GrCreateGString()** routine. You must decide where you want the GString to be stored—either in a chunk, a VM block, or a stream. If you wish to store the GString in a chunk or VM block, a memory unit of the appropriate type will be allocated for you. This routines will return the GState by which the GString may be edited and the chunk or VM block created.



The **GrDestroyGString()** routine allows you to free up the GState handle associated with your GString. You may also destroy the GString's data if you wish; specify exactly what you want to destroy by means of the **GStringKillType** argument. In addition, you may destroy another GState. You must pass the global handle of the GString to destroy—this will be the handle returned by **GrCreateGString()**, **GrEditGString()**, or **GrLoadGString()**.

23.8

The **GrLoadGString()** command loads a GString into a global memory block so that it may be drawn. Actually, it doesn't load the entire GString into memory, but does initialize the data structure so that it may be referenced through the global memory handle which the routine returns.

The **GrEditGString()** command is very much like **GrCreateGString()**, except that instead of creating a new GString, it allows you to dynamically edit an existing GString. This command loads a VM-based GString into a special data structure. Like **GrCreateGString()**, it returns a GState to which you may make drawing commands. You may insert or delete drawing commands while in this mode, all using kernel drawing routines. For more information about using this routine, see "Editing GStrings Dynamically" on page 846.

The **GrCopyGString()** command copies the contents of one GString to another. At first you might think that you could do this by allocating the target GString with **GrCreateGString()**, then drawing the source GString to the provided GState. However, the GState may only have one GString associated with it, whether that GString is being used as a source or target.

To find the handle of the GString data associated with a GState, call **GrGetGStringHandle()**. To update the VM file associated with a GString (perhaps after calling **VMsave()**), use **GrSetVMFile()**.

23.8.2 Special Drawing Commands

```
GrEndGString(), GrNewPage(), GrLabel(), GrComment(),  
GrNullOp(), GrEscape(), GrSetGStringBounds()
```

There are certain kernel graphics commands which, though they could be used when drawing to any graphics space, are normally only used when

describing GStrings. Most of these commands have no visible effect, but only serve to provide the GString with certain control codes.

The most commonly used of these routines is **GrEndGString()**. This signals that you are done describing the GString, writing a `GR_END_GSTRING` to the GString. This routine will let you know if there was an error while creating the GString (if the storage device ran out of space). Its macro equivalent is **GSEndString()**.

The **GrNewPage()** routine signals that the GString is about to start describing another page. GStrings are used to describe things to be sent to the printer, and unsurprisingly, this routine is often used in that context. An example of a prototype multi-page printing loop is shown in Code Display 23-1. Whether or not it is drawing to a GString, **GrNewPage()** causes the GState to revert to the default. When calling this routine, specify whether or not a form-feed signal should be generated by means of the **PageEndCommand** argument.

23.8

Code Display 23-1 Multi-Page Printing Loop

```
/* The application this code fragment is taken from stores several pages of
 * graphics in one coordinate space. The pages are arranged each below the other.
 * To display page #X, we would scroll down X page lengths. */

for (curPage=0; curPage < numberOfPages; curPage++) {
    GrSaveState(gstate);
    GrApplyTranslation(gstate, 0, MakeWWFixed(curPage*pageHeight));

    /* ...Draw current page... */

    GrRestoreState(gstate);
    GrNewPage(gstate, PEC_NO_FORM_FEED );}
```

The **GrLabel()** routine inserts a special element into the GString. This element does not draw anything. However, these GString labels, as they are called, are often used like labels in code. By using GString drawing routines with a certain option (described below), you may “jump” to this label and start drawing from that point in the GString. Alternately, you could start at some other part of the GString and automatically draw until you encountered that label.

The **GrComment()** routine inserts an arbitrary-length string of bytes into the GString which the GString interpreter will ignore when drawing the GString. You might use this to store anything from custom drawing commands which only your geode has to be able to understand to a series of blank bytes which could act as a sort of placeholder in memory.

Code Display 23-2 GSComment Example

```
23.8 static const byte MyGString[] = {
    GSComment(20), 'C','o','p','y','l','e','f','t',' ','l','i','n','e','3',
    ' ','P','K','i','t','t','y',
    GSDrawEllipse(72, 72, 144, 108),
    GSEndString()};
```

The **GrNullOp()** routine draws nothing and does nothing other than take up space in the GString (a single-byte element of value GR_NOP). You might use it as a placeholder in the GString.

The **GrEscape()** writes an escape code with accompanying data to the GString. Most geodes do not use this functionality; you might use it to embed special information in a **TransferItemFormat** based on GStrings.

Depending on what the system will do with your GString, the bounds of the GString may be used for many purposes. Normally the system determines the bounds of a GString by traversing the whole GString and finding out how much space it needs to draw. The **GrSetGStringBounds()** allows you to optimize this, setting up a special GString element whose data contains the bounds of the GString. You should call this routine early on in your GString definition so that the system won't have to traverse very much of your GString to discover the special element.

23.8.3 Declaring a GString Statically

For most programmers, the first encounter with GStrings (often, in fact, their first encounter with any sort of graphics mechanism) is with a program icon. Often this program icon consists of one or more GStrings, each of which contains a bitmap. These monikers are often set up using the **@visMoniker** keyword. This automatically stores the GString to a chunk. For an example

of a GString stored this way, see the appSMMoniker GString in Code Display 23-3.

Code Display 23-3 GString in Visual Monikers

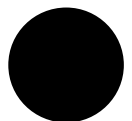
```
@start AppResource;
@object GenApplicationClass AppIconApp = {

/* The visual moniker for this application (an icon) is created by selecting the
* most appropriate moniker from the list below. Each moniker in this list is
* suitable for a specific display type. The specific UI selects the moniker
* according to the display type GEOS is running under. A text moniker is also
* supplied if the specific UI desires a textual moniker. */
23.8
GI_visMoniker = list {
    AppIconTextMoniker, /* simple text moniker */
    appLCMoniker, /* Large Color displays */
    appLMMoniker, /* Large Monochrome displays */
    appSCMoniker, /* Small Color displays */
    appSMMoniker, /* Small Monochrome displays */
    appLCGAMoniker, /* Large CGA displays */
    appSCGAMoniker /* Small CGA displays */ }
}

@visMoniker AppIconTextMoniker = "C AppIcon Sample Application";
@end AppResource

@start SAMPLEAPPICONAREASMMONIKERRESOURCE, data;
@visMoniker appSMMoniker = {
    style = icon;
/* Small monochrome icons use the standard size and 1 bit/pixel gray coloring. */
    size = standard; color = gray1; aspectRatio = normal; cachedSize = 48, 30;

/* The following lines hold GString data: */
@gstring {
    /* GSFillBitmapAtCP:
    * This macro signals that we want to fill a bitmap. The next few
    * bytes should consist of information about the bitmap; the data
    * after that holds the bitmap data. The GString reader will know
    * when it's done reading the bitmap data based on the size stated
    * in the bitmap's info—it will thus know where to look for the
    * next command (in this case a GSEndString() macro. */
    GSFillBitmapAtCP(186),
    /* Bitmap():
    * This macro is used to write basic information about the bitmap
    * to the GString. In this case, that information consists of:
```



23.8

```

        * The bitmap is 48x30. It is compressed using PackBits. It is
        * monochrome. */
Bitmap (48, 30, 0, BMF_MONO),
/* 0x3f, 0xff, 0xff, ...:
    * These numbers are the bitmap data itself. */
0x3f, 0xff, 0xff, 0xff, 0xff, 0x80, 0x7f, 0xff, 0xff, 0xff, 0xff, 0xc0,
0x60, 0x00, 0x00, 0x00, 0x00, 0x60, 0x60, 0x00, 0x00, 0x00, 0x60,
0x60, 0x00, 0x00, 0x80, 0x00, 0x60, 0x60, 0x00, 0x01, 0xc0, 0x00, 0x60,
0x60, 0x00, 0x03, 0xe0, 0x00, 0x60, 0x60, 0x00, 0x07, 0xf0, 0x00, 0x60,
0x60, 0x00, 0x03, 0xf8, 0x03, 0xfe, 0x60, 0x00, 0x01, 0xfc, 0x00, 0xf9,
0x60, 0x06, 0x04, 0xfe, 0x01, 0xfd, 0x60, 0x38, 0x06, 0x1f, 0x01, 0xfd,
0x60, 0x40, 0x03, 0x63, 0x81, 0x05, 0x60, 0x80, 0x01, 0xa3, 0xc1, 0x04,
0x60, 0x40, 0x00, 0xc3, 0x81, 0x04, 0x60, 0x38, 0x00, 0x6f, 0x01, 0x04,
0x60, 0x07, 0xfe, 0xf6, 0x00, 0x88, 0x60, 0x00, 0x01, 0xfc, 0x01, 0x8c,
0x60, 0x00, 0x03, 0xfe, 0x03, 0xde, 0x60, 0x00, 0x07, 0xf5, 0x07, 0x77,
0x60, 0x00, 0x03, 0xe2, 0x8f, 0xaf, 0x60, 0x00, 0x01, 0xc3, 0xdf, 0xdf,
0x60, 0x00, 0x00, 0x81, 0xff, 0xff, 0x60, 0x00, 0x00, 0x00, 0xff, 0xff,
0x60, 0x00, 0x00, 0x00, 0x7b, 0xff, 0x60, 0x00, 0x00, 0x00, 0x03, 0xff,
0x3f, 0xff, 0xff, 0xff, 0xff, 0x1f, 0xff, 0xff, 0xff, 0xff, 0xff,
0x00, 0x00, 0x00, 0x00, 0x03, 0xff, 0x00, 0x00, 0x00, 0x00, 0x03, 0xff,
/* GSEndString():
    * This macro lets the GString interpreter know we're done. */
GSEndString() } }

@end SAMPLEAPPICONAREASMMONIKERRESOURCE;
```

Notice that this example uses macros to set up the data for the GString. We could have just written the GString data as a series of numbers, as shown in Code Display 23-4, but the macros are usually easier to read. Each macro's name, you will notice, is taken from the corresponding graphics command name. Thus the **GSEndString()** macro corresponds to the **GrFillBitmap()** routine. There are no **GSEndString()** macros; GStrings have no return values or conditional statements, and thus have no use for retrieving this sort of information.

Code Display 23-4 GString Declared Without Macros

```

const char byte myGString[] = {
    GR_SET_LINE_STYLE,      /* First we set the line style. */
    LS_DASHED,              /* We want a dashed line. */
    0,                      /* Draw dashes starting with index 0 */
}
```

```

        GR_DRAW_RECT_TO,      /* Next we draw a rectangle from our current
                                * position to... */
        0, 72, 0, 72,         /* ...the point (72, 72). (We need the zeroes
                                * because GStrings are arrays of bytes, but
                                * we need a word to describe each
                                * coordinate. */

        GR_END_GSTRING
}; /* myGString */

```

23.8

Just as all monikers are not GStrings, not all GStrings need be declared as monikers. See Code Display 23-5 for an example of a statically declared GString taken from the Moniker sample application; here the declared GString is eventually used as a moniker, but could just as well have been passed to **GrDrawGString()** and drawn to an arbitrary graphics space.

Code Display 23-5 Statically Declared GString

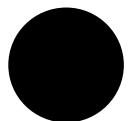
```

static void _near      SetPreDefinedGStringMoniker(void)
{
/*
 * A predefined graphics string that draws a
 * MONIKER_WIDTH by MONIKER_HEIGHT * light-blue rectangle at 0, 0.
 */
static const byte gstring[] = {
    GSSetAreaColorIndex(C_LIGHT_BLUE),
    GSFillRect(0, 0, MONIKER_WIDTH, MONIKER_HEIGHT),
    GSEndString() };

@call CycleMonikerTrigger::MSG_GEN_REPLACE_VIS_MONIKER(
    VUM_NOW,                      /* Update appearance immediately */
    MONIKER_HEIGHT, MONIKER_WIDTH, /* Since source is a GString, we need to
                                    * pass the height and width of the
                                    * GString. */

    sizeof(gstring),              /* Pass the size of sourceGString. */
    VMDT_GSTRING,                 /* Source is a gstring.... */
    VMST_FPTR,                    /* ...referenced by a far pointer */
    (dword) gstring);             /* Pointer to gstring */
} /* End of SetPreDefinedGStringMoniker() */

```



For more examples of statically declared GStrings, see the Moniker, GStest, and GStest2 sample applications.

23.8.4 Creating GStrings Dynamically

23.8

Sometimes it comes in handy to be able to create GStrings “on the fly.” To add elements to a GString, issue normal kernel drawing commands, but use a GState which is associated with the GString.

To create a new, empty GString ready for editing (i.e. with an attached GState), call **GrCreateGString()**. At this point, you may draw to the GString using normal drawing commands. For an example of creating a GString in this manner, see Code Display 23-6.

Code Display 23-6 Creating a GString Dynamically

```
#define LABEL_BOX    2
#define LABEL_CIRCLE 3

gstate = GrCreateGString(file, GST_VMEM, &myVMBlock);

GrSetLineColor(gstate, CF_INDEX, C_BLUE, 0, 0);
GrDrawRect(gstate, 0, 0, 620, 500);

GrLabel(gstate, LABEL_BOX);
GrSetAreaColor(gstate, CF_INDEX, C_RED, 0, 0);
GrFillRect(gstate, 10, 130, 610, 490);
GrSetLineWidth(gstate, MakeWWFixed(2));

GrLabel(gstate, LABEL_CIRCLE);
GrSetAreaColor(gstate, CF_INDEX, C_RED, 0, 0);
GrFillEllipse(gstate, 130, 10, 490, 370);
GrDrawEllipse(gstate, 130, 10, 490, 370);

GrEndGString(gstate);
GrDestroyGString(gstate, 0, GSKT_LEAVE_DATA);
```

Drawing to a GString in this manner is almost exactly like drawing in any other GEOS graphics environment. However, there are some important rules to keep in mind.

- ◆ The GString must end with a GR_END_GSTRING element; when the GString interpreter encounters this element, it knows to stop drawing. When creating a GString dynamically, the normal way to assure this is to call **GrEndGString()**. (Actually, this rule is not strictly true—when you learn more about drawing GStrings, you will see that it is possible to stop GString drawing based on other cues. However, it's always safest to end the GString with a GR_END_GSTRING in case some other application tries to draw the same GString.)
- ◆ Remember that you are creating a data structure which will be used later. The only commands which will affect the GString's contents are the kernel graphics routines, and only those which actually draw something or change the drawing properties. When creating a GString, it is tempting to include constructions like the following:

23.8

```
if (redFlag)
    {GrSetAreaColor(
        gstate, C_RED, CF_INDEX, 0, 0);}
else {GrSetAreaColor(
        gstate, C_BLUE, CF_INDEX, 0, 0);}
GrFillRect(gstate, 0, 0, 72, 72);
```

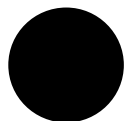
You might be surprised when you found out that the following code fragments both drew rectangles which were the same color:

```
redFlag = FALSE;
GrDrawGString(    screenGState,
                  myGString,
                  0, 0, 0, elem);

redFlag = TRUE;
GrDrawGString(    screenGState,
                  myGString,
                  0, 0, 0, elem);
```

The factor which would determine the color of the drawing in this example would be the value of redFlag when you were creating the GString, not when you were drawing it.

- ◆ Think carefully before making coordinate space transformations in GStrings. If you want to remove all transformation effects, you should always call **GrSetDefaultTransform()**, instead of **GrSetNullTransform()**. By using **GrSetDefaultTransform()**, an application that is including your GString can apply some other type of



transformation and make that the default; your application will then appear transformed as intended. However, if you call **GrSetNullTransform()**, you ignore that default transform and will appear in a strange way.

23.8

- ◆ If you use **GrInitDefaultTransform()**, you should probably bracket its use with calls to **GrSaveTransform()** and **GrRestoreTransform()**. This save/restore pair will also save the current default transformation, if there is one. By adding the save and restore, you will be preserving whatever default transform the application including yours has set up.
- ◆ If you think your Graphics String will never show up included in some other application, consider the following: The print-to-file feature creates a graphics string that can be imported into several other applications.
- ◆ If you think you have to use **GrSetTransform()**, try replacing it with a **GrSetDefaultTransform()/GrApplyTransform()** pair. This will most likely have the same effect, but will be more palatable to another application using the GString.
- ◆ If you are including some other externally-created Graphics String into your document, you probably want to bracket it with **GrSaveState()** and **GrRestoreState()**.
- ◆ If you're creating a multi-page GString which might be printed, make sure that each page is independent. There should be nothing assumed about the GState at the beginning of any page—you should instead assume that each will begin with the default GState. This applies to transformations, drawing properties, the GState's path, and so on. Keep in mind that each page should be able to print by itself if extracted from a multi-page document.

The following pseudocode is an example of a “bad” idea:

```
for(curPage=0; curPage < numberOfPages; curPage++) {  
    /* { draw current page } */  
    GrNewPage();  
    GrApplyTranslation(0, pageHeight); }
```

The following pseudocode provides a “better” way to do the same thing:

```

for (curPage=0; curPage < numberOfPages; curPage++)
{ GrSaveState();
  GrApplyTranslation(0,curPage*pageHeight);
  /* {draw current page} */
  GrRestoreState();
  GrNewPage(); }

```

Probably the most important piece of advice is to think about how the Graphics String will be used. If it will be used only under certain well-controlled circumstances, the above concerns may not affect you.

23.8

23.8.5 Drawing and Scanning

```

GrDrawGString(), GrDrawGStringAtCP(), GrSetGStringPos(),
GrGetGStringBounds(), GrGetGStringBoundsDWord()

```

There are several ways to use a GString. You've already seen how to use one as the visual moniker for a UI gadget. In that case, the UI is responsible for drawing the moniker. If you are learning about interfacing with printers, you probably know that you pass a GString to a PrintControl object to describe print jobs.

There is also a kernel graphics routine for drawing GStrings directly. The **GrDrawGString()** command draws a GString, or a part thereof. Remember that the GString must be loaded for drawing; you must call **GrLoadGString()** if you have not done so already (or if you have destroyed the GString since you last called **GrLoadGString()**).

Code Display 23-7 GrDrawGString() in Action

```

@method StaticContentClass, MSG_VIS_DRAW {
    word    lElem;
    static const byte gstringData[] = {
        GSSetAreaColorIndex(C_RED), GSFillRect(0, 0, 72, 72),
        GSEndString() };
    @callsuper();
    gstring = GrLoadGString (PtrToSegment(gstringData), GST_PTR,
        PtrToOffset(gstringData));
    GrDrawGString(gstate, gstring, 0, 0, 0, &lElem); }

```



```
23.8 @method DynamicContentClass, MSG_VIS_DRAW{
      Handle      gstring;
      VMFileHandle file;
      char        fileString[] = ".", 0;
      VMBlockHandle gstringData;
      GStringElement lElem;

      @callsuper();
      file = VMOpen(fileString,
                    (VMAF_FORCE_READ_WRITE | VMAF_USE_BLOCK_LEVEL_SYNCHRONIZATION),
                    VMO_CREATE_TRUNCATE,
                    0);

      gstring = GrCreateGString(file, GST_VMEM, &gstringData);
      GrSetAreaColor(gstring, CF_INDEX, C_RED, 0, 0);
      GrFillRect(gstring, 0, 0, 72, 72);
      GrEndGString(gstring);
      GrDestroyGString(gstring, NULL, GSKT_LEAVE_DATA);

      gstring = GrLoadGString(file, GST_VMEM, gstringData);
      GrDrawGString(gstate, gstring, 0, 0, 0, &lElem);
      GrDestroyGString(gstring, NULL, GSKT_DESTROY_DATA);

      FileDelete(fileString);
}
```

The **GrDrawGString()** routine has several arguments. A simple usage of the routine is shown in Code Display 23-7 on page ● 843. To take advantage of some of the more powerful features of **GrDrawGString()**, you should know what the purpose of the arguments.

- ◆ You must provide a GState to draw to. You may wish to call **GrSaveState()** on the GState before drawing the GString (and call **GrRestoreState()** afterwards). If you will draw anything else to this GState after the GString, you must call **GrDestroyGString()** on the GString, and pass this GState's handle as the gstate argument so that **GrDestroyGString()** can clean up the GState.
- ◆ You must provide a GString to draw. The GString must be properly loaded (probably by means of **GrLoadGString()**).
- ◆ You can provide a pair of coordinates at which to draw the GString. The graphics system will translate the coordinate system by these coordinates before carrying out the graphics commands stored in the GString.

- ◆ You can provide a **GSControl** argument which requests that the system stop drawing the GString when it encounters a certain type of GString element. If the GString interpreter encounters one of these elements, it will immediately stop drawing. The GString will remember where it stopped drawing. If you call **GrDrawGString()** with that same GString, it will continue drawing where you left off. Note that any time a GString-traversing function such as **GrDrawGString()** returns, it returns a **GSRetType** value which makes it clear exactly why it stopped traversing the GString.

23.8

- ◆ You must provide a pointer to an empty **GStringElement** structure. **GrDrawGString()** will return a value here when it is finished drawing. If the GString has stopped drawing partway through due to a passed **GSControl**, the returned **GStringElement** value will tell you what sort of command was responsible for halting drawing. For instance, if you had instructed **GrDrawGString()** to halt on an 'output' element (**GrDraw...()** or **GrFill...()** commands), then when **GrDrawGString()** returns, you would check the value returned to see what sort of output element was present.

Note that those last two arguments aren't very useful except when used in conjunction with some other GString routines which we will get to later.

The **GrDrawGStringAtCP()** routine functions in much the same way as **GrDrawGString()**, except that the current pen position will be used in the place of passed coordinate arguments.

The **GrSetGStringPos()** routine allows you to skip any number of GString elements, or go back to the beginning or end of the GString. You specify whether you want to skip to the beginning, end, or ahead by a few steps; this is specified by the **GStringSetPosType** argument. This routine is useful both when drawing and editing GStrings. Note that you may also use this routine to jump backwards in a GString, but this only works with VM-based GStrings. The GString must be loaded for drawing or editing, and you will pass in the GString's global handle, as supplied by **GrLoadGString()** or **GrEditGString()**.

Code Display 23-8 GrSetGStringPos() In Action

```
/* The following routine is used to allow creating an 'overlay' effect. Normally,
 * multi-page GStrings will contain form feed elements that signal the break
 * between pages. Here we see a section of code which will filter out form feeds by
 * skipping those elements. The result will be a single page in which all the pages
 * of the original GString are drawn on top of each other. */
```

23.8

```
for (   gsr = GrDrawGString(gstate, gstring, 0, 0, GSC_NEW_PAGE, gse);
      gsr == GSR_FORM_FEED;
      gsr = GrDrawGString(gstate, gstring, 0, 0, GSC_NEW_PAGE, gse) )
    {GrSetGStringPos(gstring, GSSPT_SKIP_1, 0); }
```

Because a GString remembers its place when you stop drawing partway through, if you wish to 'reset' the GString position, you should use **GrSetGStringPos()** to set it back to the beginning.

Occasionally you may be curious to know how much space is necessary to draw a GString. The **GrGetGStringBounds()** routine determines this, returning the coordinates describing the GString's bounding rectangle. If the GString may have very large bounds, you should use the **GrGetGStringBoundsDWord()** routine instead.

23.8.6 Editing GStrings Dynamically

GrEditGString(), **GrDeleteGStringElement()**

Applications may find cause to dynamically alter an existing GString. You might create a sort of template GString and want to fill in some parts at a later time. If you will draw several similar GStrings, it might be nice to use a single GString, but change only certain parts before drawing each time.

GrEditGString() allows you to edit an existing GString. It only works with VM-based GStrings. Calling **GrEditGString()** allocates a new GState and associates it with the GString. Any drawing commands to this GState will be appended to the GString. You may use **GrDrawGString()** (along with appropriate **GSControl** values) and **GrSetGStringPos()** to change position within the GString, allowing you to insert new commands into the middle of the GString.

The **GrDeleteGStringElement()** routine allows you to delete any number of GString elements. The elements deleted will be taken starting at your position in the GString. This command only works while editing the GString, and you must pass the GString's editing GState handle to this routine.

23.8.7 Parsing GStrings

`GrGetGStringElement()`, `GrParseGString()`

23.8

For complicated GString operations, you may find the following advanced routines helpful.

GrGetGStringElement() returns the raw data associated with the current GString element. To understand this stream of bytes, you must know what sorts of data are associated with each kind of GString element. For example **GrGetGStringElement()** might return `GR_DRAW_RECT` with the following buffer of bytes:

```
GR_DRAW_RECT, 0x00, 0x48, 0x00, 0x24,
0x00, 0x90, 0x00, 0x84
```

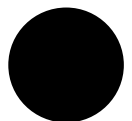
You must know enough about GString element structures to know that this will draw a rectangle with bounds {72, 36, 144, 108}. To find out this sort of information, examine the `GS...()` macros, or search **gstring.h** for macros containing the appropriate **GStringElement** value.

Code Display 23-9 GrGetGStringElement() In Action

```
/* Our application allows for a second kind of spline, a B-spline. This spline
 * looks similar to a regular Bézier spline, but is somewhat different and uses
 * a different mathematical formula. When this app creates a GString, it will
 * output Bézier splines in the normal way.

 * When outputting a B-spline to a GString, it outputs the GString element for a
 * regular spline. That way, other applications will be able to draw the GString
 * mostly correctly. However, all B-spline elements will be preceded by a
 * GString comment 'B''s''p''l'.':

 *      GSComment(4), 'B','s','p','l',
 *      GSDrawSpline(...), ...
```



* The following snippet of code will be used when this application draws a GString. It will look for the significant comments. When it finds them, it will know that the following GR_DRAW_SPLINE element should actually be treated differently. */

23.8

```
for (    gsr = GrDrawGString(gstate, gstring, 0, 0, GSC_MISC, gse);
        gsr == GSRT_MISC;
        gsr = GrDrawGString(gstate, gstring, 0, 0, GSC_MISC, gse) )
    {byte canonicalBuffer[] = {GR_COMMENT,'B','s','p','l'};
      byte buffer[20];
      int eSize;

      GrGetGStringElement(gstate, gstring,
                          sizeof(buffer),&buffer, &eSize);

      /* First check to see if this is the
       * comment we're looking for: */
      if (strncmp(buffer, canonicalBuffer, 5)) {
          /* Skip ahead to the GrDrawSpline element */
          GrSetGStringPos(gstring, GSSPT_SKIP_1, 0);
          GrGetGStringElement(gstate, gstring,
                              sizeof(buffer), &buffer, &eSize);
          /* Draw spline using our routine */
          MyDrawBSpline(gstate, buffer+3,
                        (eSize-3)/sizeof(Point))
          /* Advance GString so kernel won't draw a
           * Bézier spline over our B-spline. */
          GrSetGStringPos(gstring, GSSPT_SKIP_1, 0) }
    }
```

The **GrParseGString()** command calls a callback routine on all elements of a GString which match some criterion. The routine may save information about the elements, draw to a GState, or something completely different.

GrParseGString() takes the following arguments:

- ◆ GString to parse.
- ◆ GState handle. **GrParseGString()** itself will do nothing with this handle, and passing a NULL handle is permitted. However, this GState will be passed to the callback routine. If your callback routine will draw, it is thus convenient to pass a properly initialized GState to **GrParseGString()** which the callback routine may then draw to.

- ◆ A record of type **GSControl**. This will determine which elements will be passed on to the callback routine. If you set `GSC_OUTPUT`, the callback routine will be called only for those GString elements which draw something. If you set `GSC_ONE`, the callback routine will be called upon all of the GString elements.
- ◆ Far pointer to the callback routine itself.

The callback routine is passed a pointer to the GString element and the handle of the GState that was passed to **GrParseGString()**.

23.9

23.9 Graphics Paths

```
GrBeginPath(), GrEndPath(), GrCloseSubPath(),
GrSetStrokePath(), GrGetPathBounds(), GrTestPointInPath(),
GrGetPathPoints(), GrGetPathRegion(), GrGetPath(),
GrSetPath(), GrGetPathBoundsDWord(), GrTestPath()
```

A *path* is a data structure which describes a route across a graphics space. Normally, applications use the path to specify an arbitrary screen area, defining a path that describes the area's borders. Like a GString, a path is created by calling an appropriate initiation routine followed by a series of drawing commands.

Paths don't have to be continuous. An unconnected path is said to be made up of more than one path element.

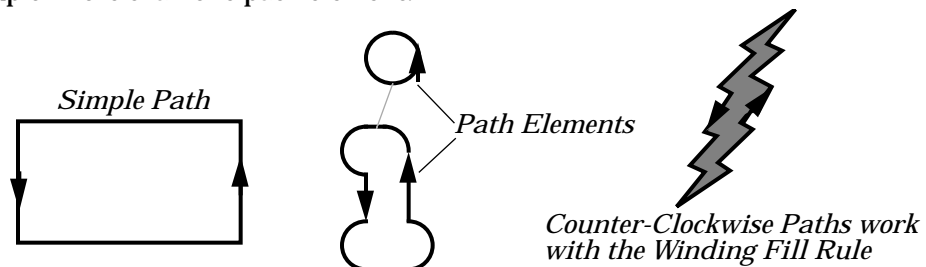
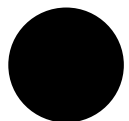
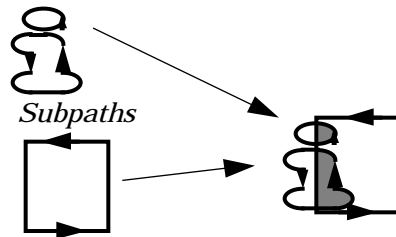


Figure 23-12 Graphics Paths

Paths can be created by taking the intersection or union of two or more paths. In this case, the paths which are combined to make the resulting path are called *sub-paths*. (See Figure 23-13).





23.9

Figure 23-13 *Combining Paths*

Paths are normally used to describe regions. However, regions are basically bitmaps, whereas paths are described in terms of the standard coordinate space. Thus, while regions don't scale well at all, it is possible to scale a path perfectly, then use the transformed path to more correctly compute the desired region. If this comparison sounds similar to that between bitmap- and outline-based fonts, it should. The "outline" of an outline-based font character is, in fact, a specialized form of path.

Paths are also used to describe clipping regions. It is possible to combine any path with a window's clipping region to further restrict the clipping area. Thus it is possible to clip to an ellipse or to a Bézier curve, or even to use text as a clipping region.

**Figure 23-14** *Geode-Defined Clipping Paths*

It is possible to clip using paths incorporating text, splines, and other elements.

When constructing a path, you should keep in mind what it will be used for. If the path is to be filled using the winding rule, it is important that all edges go the correct direction. The winding rule algorithm assumes that the region is described by edges that go around it counterclockwise. Edges going clockwise describe hollows. The odd/even fill rule will work independently of path direction. See section 24.2.11 of chapter 24 for more information about winding rules.

Both the winding and odd/even fill rules demand closed path elements. If any path elements are not filled, the routine will treat them as if they were, connecting the ends of each open path element with a straight line.

It is also possible to alter an existing path by combining it with another. The new path thus formed preserves both paths. If drawn, both paths will appear. If the path is filled or used as a clipping region, the geode can specify how the regions described by the paths should be combined, whether the intersection or union should be taken. Any number of paths can be combined in this manner.

23.9

GrBeginPath() signals that the geode is about to start describing a path. All drawing commands directed at the GState will go into constructing the path until **GrEndPath()** is called. **GrBeginPath()** is also the routine used to combine a path with an existing path. Calling **GrBeginPath()** when already constructing a path signals that further graphics commands should describe a new path to be combined with the existing one. The new path can either replace the existing one or combine to find the intersection or the union.

GrCloseSubPath() closes the current path element, adding a straight line segment from the current position to the starting point if necessary.

GrGetPathBounds() returns the bounding coordinates of a rectangle that completely encloses the current path. **GrTestPointInPath()** tests whether a passed point falls within the current path. **GrGetPathPoints()** returns all the points along a path in the order visited. **GrGetPathRegion()** returns the region defined by the path. **GrGetPathBoundsDWord()** returns the bounds of the path and works in a 32-bit graphics space. If you just want to know whether or not a given path exists, then call **GrTestPath()**, passing the **GetPathType** GPT_CURRENT.

GrSetStrokePath() replaces the current path with the one that would result from “stroking” (drawing) the current path with the current line attributes. For example, if the current line style is a dotted line, the result will most likely be a set of many skinny regions. At this time, stroke paths cannot be used for clipping purposes. However, geodes can still draw and fill these paths.

Paths can be drawn or filled using the **GrDrawPath()** and **GrFillPath()** commands. For more information about these routines, see section 24.2.11 of chapter 24.



GrGetPath() retrieves the handle of a block containing the path's data. You may pass this handle to **GrSetPath()** and thus copy a path to another GState.

23.10 Working With Video Drivers

23.10

The main benefit of the device independence is that the geode writer can issue graphics commands without worrying about the device. Working with the video driver is left to the graphics system.

23.10.1 Kernel Routines

`GrInvalRect()`, `GrInvalRectDWord()`, `GrGrabExclusive()`,
`GrGetExclusive()`, `GrReleaseExclusive()`, `GrBitBlt()`,
`GrGetBitmap()`, `GrGetWinHandle()`



Advanced Topic

These commands are beyond the scope of most programmers.

Sometimes the geode may want more power over the driver. It can see what device coordinates correspond to a set of standard GEOS coordinates or vice versa.

It is possible to update part of a drawing without exposing the whole window. Calling **GrInvalRect()** causes a passed rectangular area to be updated; the area outside the rectangle will be unaffected. **GrInvalRectDWord()** works the same way, but for large coordinate spaces.

Programs can also seize exclusive access to a video driver by calling the **GrGrabExclusive()** command, which allows only the passed GState to alter what is shown on the screen. This routine is useful for programs such as screen dumps who want to accomplish something concerning the screen without worrying that programs in other threads will change the screen in the meantime. **GrReleaseExclusive()** ends the exclusive access to the screen so that other GStates can update.

To find out if the video exclusive is presently grabbed, call **GrGetExclusive()**. This will return the handle of the GState presently in possession of the exclusive, or zero if there is no such GState.

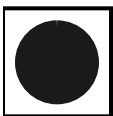
GrBitBlt() is an advanced function used to quickly copy or move data within video memory. Effectively, it can copy or move a rectangular part of the document space to another part of that space—the passed **BLTMode** will determine whether the area is copied or moved. This might be used in an arcade game or animation program to move simple pictures around the screen very quickly. Pass this function the source and destination rectangles, and whether you are copying or moving the block. After calling **GrBitBlt()**, the changes have been made to video memory but have not actually been drawn to the screen. Make sure that the affected areas are redrawn by invalidating the appropriate area. If speed is a concern, and if you're using **GrBitBlt()** it probably is, you'll probably want to restrict the clipping area when the drawing is refreshed.

23.10

GrGetBitmap() basically returns a dump of an arbitrary display area. It is an advanced function and should be used with some caution. If asked to dump an area from a **GState** being displayed to screen, **GrGetBitmap()** won't check to see if the dumped area is obscured by another window, and so your dump might include a picture of that other window. However, if you're working with some sort of offscreen bitmap, this function provides a way to look at large portions of it a time. Note that to look at smaller areas, you might prefer to use **GrGetPoint()**, an optimized, easier to use function to find out the color of a pixel.

At some times, it may prove useful to know what window, if any, is associated with a **GState**. To find out, call **GrGetWinHandle()**.

23.10.2 Direct Calls to the Driver



Advanced Topic

Rarely, a geode may wish to make direct calls to the video driver. In most cases, anything your code might want the video driver to do can be handled better by going through the appropriate graphics routine.

23.11 Windowing and Clipping

Windows are the interface between the graphics commands and the GEOS user interface. In this section we will discuss some of the graphical mechanisms associated with windows in GEOS.

23.11

23.11.1 Palettes

Each window has a color palette associated with it. For more information about manipulating palettes, see section 24.3.1.3 of chapter 24. The system will use the palette of whatever window is active. As a result, if two windows have different palettes, when one window is active, the other's colors will be distorted.

23.11.2 Clipping

```
GrSetClipPath(), GrSetClipRect(), GrGetClipRegion(),  
GrTestRectInMask(), GrSetWinClipRect(), GrGetMaskBounds(),  
GrGetMaskBoundsDWord(), GrGetWinBounds(),  
GrGetWinBoundsDWord()
```



Advanced Topic

The window system's clipping is sufficient for most geodes.

The graphics system provides some routines which allow geodes to work together with the windowing system to control clipping. A window's clipping region, you will recall, is that area of the window that must be redrawn. It corresponds to that area of the graphics space which is visible inside the window and not obscured by another window. The area outside the clipping region is supposed to be that area that doesn't need to be redrawn. Geodes which are certain that part of their graphics space doesn't need to be (or shouldn't be) redrawn can restrict their clipping region to exclude this portion.

Restricting the clipping region can lead to quicker redraws, since the graphics system doesn't have as much to redraw. Arcade games in which most of the action only takes place in one or two areas of the display can restrict their clipping region to speed redraws in the smaller, active areas. Also, the clip region can be used as a sort of stencil (see Figure 23-14 on page ● 850).

The clip region is defined by a path which describes the boundaries of the region. A window typically begins with a rectangular clip region. If it is partially obscured by other windows, this rectangle will have “bites” taken out of it. This is done using normal path routines used for combining paths. The clipping path of a newly obscured window is determined by computing its old clipping path with that of the obscuring window.

Since any number of paths may be combined, it is a simple matter for the graphics system to combine a window's clipping path with a geode's path chosen to restrict the clipping region. Thus the geode can define its restricted clipping area without knowing anything about the window's present status.

23.11

GrSetClipPath() sets the geode-defined path to use when restricting the window's clip path. **GrSetClipRect()** is an optimization of **GrSetClipPath()** to handle the most common case in which the new path element is a rectangle.

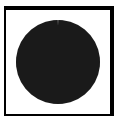
GrGetClipRegion() returns the **Region** data structure corresponding to the current clip path. **GrGetMaskBounds()** and **GrGetMaskBoundsDWord()** return the bounds of the current clipping path. To find out whether there is a clip path at all, just call **GrTestPath()** and pass GPT_CLIP.

GrTestRectInMask() determines whether a rectangular area lies fully, partially, or not at all within the clipping area. This routine is useful for optimizing redraws.

GrSetWinClipRect() and **GrSetWinClipPath()** set the clipping path associated with the window; you should never have occasion to use it. **GrGetWinBounds()** and **GrGetWinBoundsDWord()** return the bounds of the window clipping path. To find out whether there is a window clipping path at all, call **GrTestPath()** and pass GPT_WIN_CLIP.

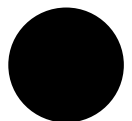
23.11.3 Signalling Updates

`GrBeginUpdate()`, `GrEndUpdate()`



Advanced Topic

The kernel provides two messages by which the geode may signal that it is updating the contents of a window. When updating a region (as when handling a MSG_META_EXPOSED), the geode should call **GrBeginUpdate()**

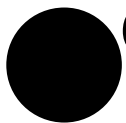


after creating the GState, and call **GrEndUpdate()** before destroying the GState.

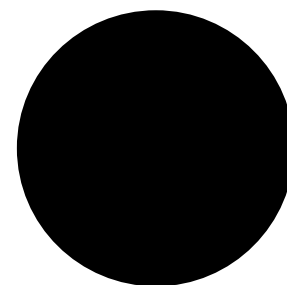
This causes the system to store the GState for future comparisons during clipping region calculation. If you don't call these functions, the clipping region is likely to be wrong for this and other updates. Default system MSG_META_EXPOSED handlers call these routines.

23.11

Note that you only need call these routines when performing a requested update; if you are drawing to a window without being asked to do so, you need not call these routines.

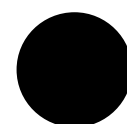


Drawing Graphics



24

24.1	Drawing Goals.....	859
24.2	Shapes	859
24.2.1	Dots	860
24.2.2	Lines.....	861
24.2.3	Rectangles.....	862
24.2.4	Ellipses.....	862
24.2.5	Elliptical Arcs	863
24.2.6	Three-Point Arcs.....	864
24.2.7	Rounded Rectangles.....	865
24.2.8	Polylines and Polygons.....	866
24.2.9	Bézier Curves and Splines.....	867
24.2.10	Drawing Bitmaps	870
24.2.11	Paths	872
24.2.12	Regions.....	873
24.2.13	Text.....	876
24.2.13.1	Displaying Text	877
24.2.13.2	Special Text Attributes	878
24.2.13.3	Accessing Available Fonts	880
24.2.13.4	Text Metrics.....	881
24.3	Shape Attributes	885
24.3.1	Color	886
24.3.1.1	Using Available Colors.....	887
24.3.1.2	When the Color Isn't in the Palette	889
24.3.1.3	Custom Palettes.....	890
24.3.2	Patterns and Hatching.....	891
24.3.3	Mix Mode.....	895
24.3.4	Masks.....	897
24.3.5	Line-Specific Attributes	899





This chapter describes the commands used to draw things on a display. "Graphics Environment," Chapter 23, explains how to set up the display necessary to use these commands.



You must have read the previous chapter; in particular, you should be familiar with GStates. To use certain commands, you should also be familiar with the GEOS fixed point number formats.

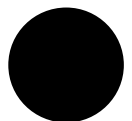
24.1

24.1 Drawing Goals

The graphics system has a set of graphic commands capable of describing anything that might ever be drawn; at the same time, the command set should not be overwhelming. Common routines must be heavily optimized: if the line or text drawing routines are slow, the system will be slowed by a similar amount. Commands for drawing the various shapes must exist for all the contexts discussed in the previous chapter. In GEOS, you use the same graphics commands whether you are drawing to a display, describing a GString (which can be sent to a printer), drawing to a bitmap, or describing a path. This simplifies graphics programming a great deal.

24.2 Shapes

Depending on how much experience you have with the GEOS graphics system, you may have some idea already about what sorts of shapes can be drawn and the ways to draw them. Most of these commands have names like **GrDrawShape()** or **GrFillShape()** (e.g. **GrDrawRect()**, **GrFillEllipse()**). Normally, a command of the form **GrDrawShape()** draws the outline of a shape, while **GrFillShape()** fills in the interior of the shape. Commands with names like **GrSetAttribute()** (e.g. **GrSetAreaColor()**) change the color, fill pattern, and other attributes of shapes to be drawn. Most of these commands are passed a GState. The drawing commands are also passed coordinates at which to draw.



For many of these commands, there are GString opcodes which represent the command in a GString. Also, the arguments used when drawing these shapes often correspond to instance data specific to the Graphic Object which draws that shape.

24.2

Most of these routines work with standard coordinates, measured in typographer's points. For geodes that need to make drawings which are precise to a fraction of a point, four routines have been set up to use WWFixed coordinates, and are thus accurate to a fraction of a point. These routines are **GrDrawRelLineTo()**, **GrRelMoveTo()**, **GrDrawRelCurveTo()**, and **GrDrawRelArc3PointTo()**. Geodes may use these routines to draw the outline of any conceivable two-dimensional shape. To create a precise, filled shape, use these routines to describe a path and then fill the path.

24.2.1 Dots

`GrDrawPoint()`, `GrDrawPointAtCP()`

A point is the smallest thing drawable; it will always appear as a single pixel. The point's position is defined by a standard coordinate pair. Points are drawn using line attributes. After drawing a point, the pen position will be at the point's coordinates.

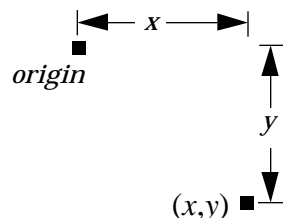


Figure 24-1 *Point*

A point is defined by its coordinates.

GrDrawPoint() draws a point at the passed coordinate pair.

GrDrawPointAtCP() draws a point at the current pen position.

24.2.2 Lines

```
GrDrawLine(), GrDrawLineTo(), GrDrawRelLineTo(),  
GrDrawHLine(), GrDrawHLineTo(), GrDrawVLine(),  
GrDrawVLineTo()
```

A line is simply a line segment that connects two points. Lines are drawn using the current line attributes stored with the GState. The new pen position becomes the last passed coordinate pair.

24.2

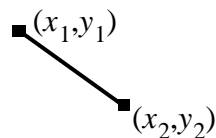


Figure 24-2 *Line*

A line is defined by its endpoints.

GrDrawLine() draws a line between points. **GrDrawLineTo()** draws a line from the current pen position to the passed point. **GrDrawHLine()** draws a horizontal line between two points. These two points share the same y coordinate (thus the line is horizontal). **GrDrawHLineTo()** draws a horizontal line from the current pen position to a passed x coordinate. **GrDrawVLine()** draws a vertical line between two points which share a common x coordinate. **GrDrawVLineTo()** draws a vertical line from the current pen position to a passed y coordinate. The **GrDrawH...()** and **GrDrawV...()** routines save space compared to **GrDrawLine()** or **GrDrawLineTo()**, since fewer coordinates are necessary to define the line.

GrDrawRelLineTo() draws a line from the current pen position to the point at the specified x and y offset from the starting position. This routine takes very precise coordinates, and is useful for describing paths.

24.2.3 Rectangles

`GrDrawRect()`, `GrDrawRectTo()`, `GrFillRect()`, `GrFillRectTo()`

Rectangles are defined by four coordinates which can be thought of as either defining the rectangle's left, top, right, and bottom bounds or as specifying two opposite corners of the rectangle.

24.2

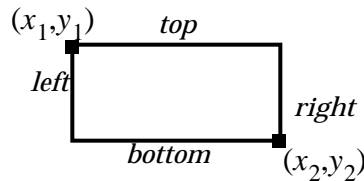


Figure 24-3 *Rectangle*

GrDrawRect() draws the outline of a rectangle using the passed coordinates. **GrDrawRectTo()** draws the outline of a rectangle using the current pen position as one of the defining points; the pen position is unchanged by this operation. These functions draw the rectangle outline using the current line attributes. **GrFillRect()** draws a filled rectangle defined by the passed coordinates. **GrFillRectTo()** fills a rectangle of which the pen position is one corner. **GrFillRect()** and **GrFillRectTo()** use the GState's area attributes. They do not draw a border around the rectangle; if you want a bordered rectangle, call **GrFillRect()** and follow it with **GrDrawRect()**. Note that if the order of these operations is reversed, the fill may obscure the draw.

24.2.4 Ellipses

`GrDrawEllipse()`, `GrFillEllipse()`

Ellipses are defined by their bounding rectangles. The pen position becomes the first coordinate pair passed. Circles are ellipses with heights equal to their widths.

GrDrawEllipse() draws the outline of an ellipse using the current line drawing attributes. **GrFillEllipse()** fills the ellipse's area using the current area attributes.

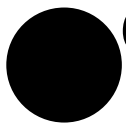




Figure 24-4 *Ellipse*
An ellipse with its bounding rectangle.

24.2

24.2.5 Elliptical Arcs

`GrDrawArc()`, `GrFillArc()`

An arc is a partial ellipse. An arc is defined in terms of its base ellipse, the angle at which to start drawing the arc, and the angle at which to stop drawing. Angles are counted in degrees counter-clockwise with 0° corresponding to the positive x axis (i.e., “3 o’clock”).

GrDrawArc() draws the outline of an elliptical arc, a curved line. It does so using the GState’s current line attributes. **GrFillArc()** fills the arc. There are two ways to fill an arc: you can fill in the wedge described by the arc, or you can fill just the region between the arc and its chord; set the style with an **ArcCloseType** value.

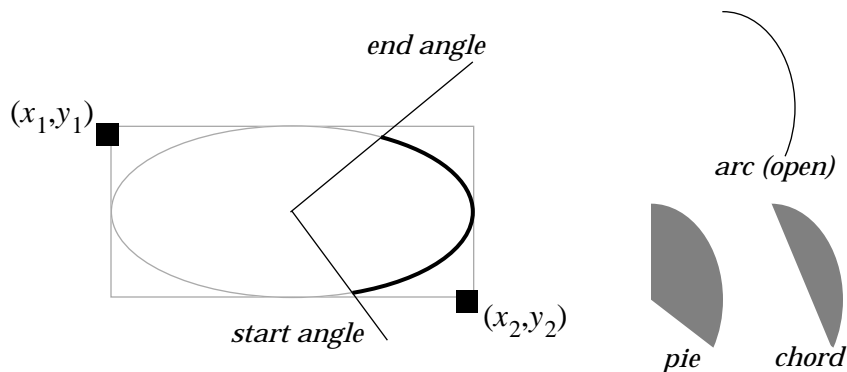
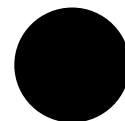


Figure 24-5 *Elliptical Arc*
On the left is an arc shown with its defining ellipse. On the right are the effects of drawing an arc, filling an arc as a pie, and filling an arc as a chord.



24.2.6 Three-Point Arcs

`GrDrawArc3Point()`, `GrDrawArc3PointTo()`, `GrFillArc3Point()`,
`GrFillArc3PointTo()`, `GrDrawRelArc3PointTo()`

The graphics system allows another way to specify arcs. Given two endpoints and one arbitrary point, there is a unique circular arc which has those endpoints and passes through that arbitrary point. The closer the arbitrary point is to the line connecting the endpoints, the shallower the described arc.

24.2

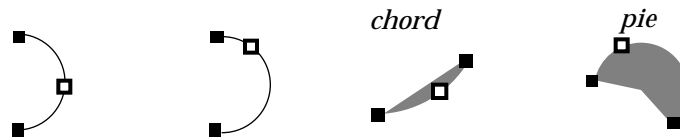


Figure 24-6 *Three-point Arcs*

GrDrawArc3Point() draws the three-point arc corresponding to the passed points. The second endpoint passed becomes the new pen position.

GrDrawArc3PointTo() draws a three-point arc using the present pen position as one of the endpoints; the other endpoint becomes the new pen position. **GrFillArc3Point()** fills a three-point arc. **GrFillArc3PointTo()** fills an arc that has the present pen position as an endpoint.

The **GrDrawRelArc3PointTo()** routine draws a three-point arc where the pen position is the first point and the other two points are specified as offsets from that position. This routine takes WWFixed coordinates for precision drawing. **GrFillRelArc3PointTo()** fills a three-point arc where the pen position is the first point and the other two points are specified as offsets from that position.

One time when programmers might especially want to use three point arcs is in the construction of paths. An important consideration when constructing paths is making sure that the various segments of the path are connected; that is, that they share endpoints. When specifying elliptical arcs, the endpoints are never explicitly defined. Thus, it is ambiguous in some cases whether an arc is supposed to be connected to something else. Because three-point arcs include their endpoints with their definition, there is no such ambiguity (see Figure 24-7).

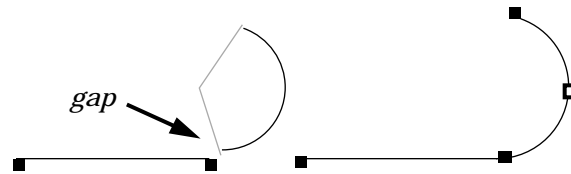


Figure 24-7 *Advantages of the Three-point Arc*

The elliptical arc doesn't start quite where the line leaves off, perhaps due to a poorly calculated angle. The three-point arc, defined in terms of its endpoints, is certain to have an endpoint in common with the line.

24.2

24.2.7 Rounded Rectangles

`GrDrawRoundRect()`, `GrDrawRoundRectTo()`, `GrFillRoundRect()`, `GrFillRoundRectTo()`

Rounded rectangles are defined in terms of their bounding rectangle and the radius of the circle used to compute the rounded corners. The smaller the corner circle is, the sharper the rectangle's corners will be. A larger corner circle results in a more rounded corner with more area taken away.

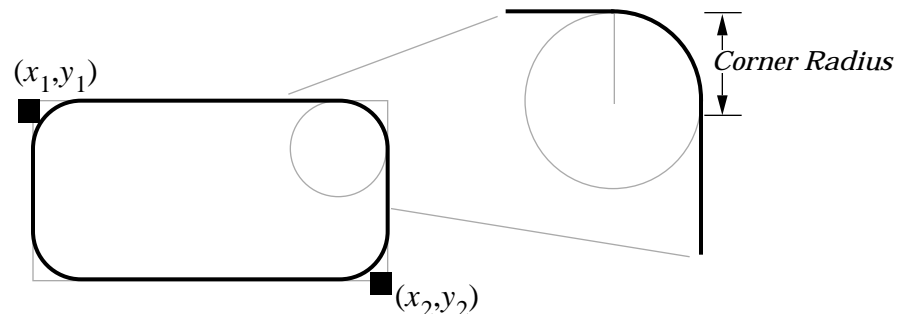


Figure 24-8 *Rounded Rectangle*

Pictured here is a rounded rectangle, with detail showing the circle whose radius defines the rounded rectangle's corner curvature.

GrDrawRoundRect() draws the outline of a rounded rectangle with the passed bounding rectangle and corner circle dimensions. The drawing position is set to the first passed coordinate pair. **GrDrawRoundRectTo()**



draws the outline of the rounded rectangle for which the current position is one corner of the bounding rectangle. The current position is unaffected by this operation. **GrDrawRoundRect()** and **GrDrawRoundRectTo()** use the current line drawing attributes. **GrFillRoundRect()** fills a rounded rectangle with passed bounding rectangle and corner radius using the current area attributes. **GrFillRoundRectTo()** fills a rounded rectangle that has the current position as one corner of the bounding rectangle.

24.2

24.2.8 Polylines and Polygons

`GrDrawPolyline()`, `GrDrawPolygon()`, `GrFillPolygon()`,
`GrBrushPolyline()`, `GrTestPointInPolygon()`

Polylines and polygons are drawings made up of chains of connected lines. They are defined as lists of points, or corners. After drawing a polyline or polygon, the pen position will be at the last point of the shape.

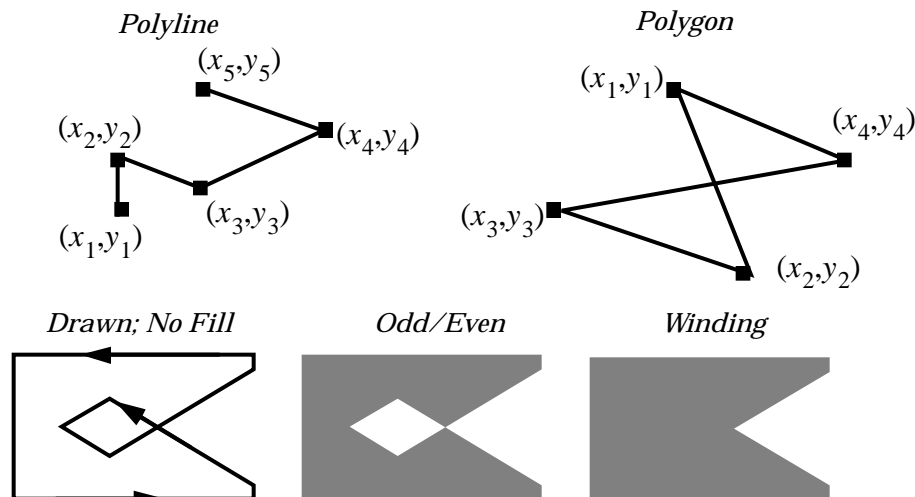


Figure 24-9 Polylines, Polygons, and Fill Rules

Polylines can't be filled. Polygons can be drawn as with polylines, or filled using one of two "fill rules." In the polygon shown, the odd/even fill results in a "cavern", or hollow area. The winding rule fills in this cavern.

GrDrawPolyline() draws a polyline using the current line attributes.
GrDrawPolygon() draws a polygon using the current line attributes.

To draw a closed figure with **GrDrawPolyline()**, the first and last point must have the same coordinates. **GrDrawPolygon()** draws figures as closed automatically; having the same beginning and ending point is unnecessary.

GrFillPolygon() fills the interior of a polygon. It does so using the current area attributes. It also uses a fill rule, describing how the polygon should be filled. The odd/even fill rule decides whether a point is within a polygon by seeing how many times a ray drawn from the point to the exterior of the polygon crosses an edge of that polygon. If the result is odd, the point is considered to be in the interior and is filled. If the result is even, the point is outside the polygon and is left alone. The winding rule works in a similar manner, but whenever the ray crosses an edge, the rule looks at the direction of that edge: if it crosses the ray left to right, an accumulator is incremented; if it crosses the ray right to left, the accumulator is decremented. Those points for which the ray's accumulator is non-zero are considered inside the region.

24.2

GrBrushPolyline() is an optimized routine. It provides a fast, simple way to draw thick polylines. Instead of specifying a line width in points, the caller passes the dimensions of a rectangle of pixels. This rectangle will be used as a sort of brush, dragged along the course of the polyline. The result is a polyline drawn very quickly. However, this function is not display-independent, and is therefore not WYSIWYG. Because the rectangle dimensions are specified in pixels instead of points, the polyline will be thicker or thinner depending on the display's resolution. In the system software, this routine provides the "ink" feedback for pen-based systems, where speed is a top priority.

To find out whether a given point falls within a polygon, call **GrTestPointInPolygon()**.

24.2.9 Bézier Curves and Splines

```
GrDrawCurve(), GrDrawCurveTo(), GrDrawSpline(),  
GrDrawSplineTo(), GrDrawRelCurveTo()
```

Bézier curves are mathematical constructs which provide a cheap and easy way to define smooth curves in a manner that computers can understand. There are other ways to define curves to computers, but the Bézier was

chosen for the GEOS kernel because it is used in many standard font description formats. Splines, as implemented in GEOS, are created by drawing curves in sequence.

24.2

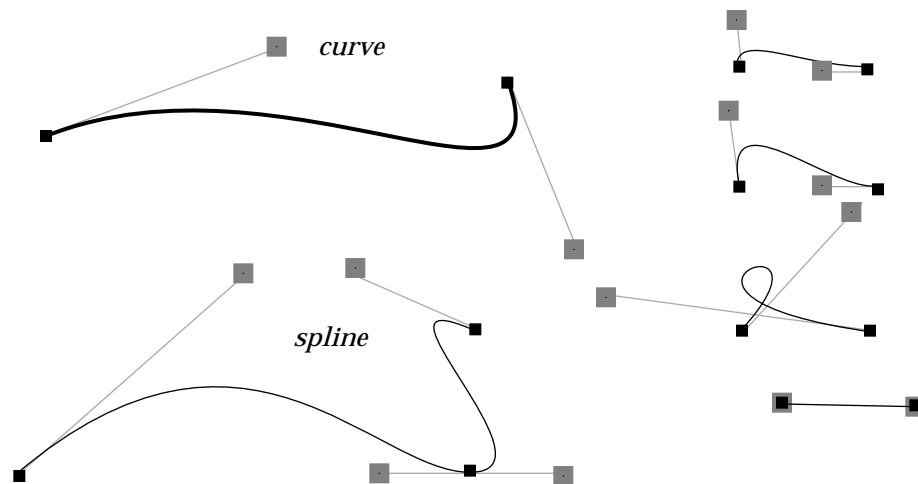


Figure 24-10 *Splines and Bézier Curves*
Curves and a spline, shown here with control points.

Bézier curves are defined in terms of four points. Two of these points are the endpoints, known as anchor points. The other two points are known as control points, one associated with each anchor point. The curve extends from anchor point to anchor point. The line between an anchor point and its control point determines the slope, or derivative, of the curve at the anchor point. The further the control point is from the anchor point, the further the curve wants to go along the straight line before curving off towards the other anchor point. A control point at zero distance from its anchor point won't affect the curve; if both control points are at zero distance from their anchors, the result will be a straight line segment.

GrDrawCurve() draws a Bézier curve. It takes four points as arguments, using the first and last as anchor points and the middle two as control points. **GrDrawCurveTo()** draws a curve but uses the current pen position as the first anchor point, setting the pen position to the second anchor point after drawing.

It would be possible to draw splines by drawing a number of curves which had common endpoints, but the graphics system provides the **GrDrawSpline()** routine by which a spline with an arbitrary number of spline segments may be drawn with one call. **GrDrawSplineTo()** draws a spline with the current position as the first anchor point. The spline drawing routines require the application to set up an array of points. When calling **GrDrawSpline()**, these points should be in the order: anchor, control, control, anchor, control, control,..., anchor. The total number of points should be equal to $3n+1$, where n is equal to the number of spline segments. Since **GrDrawSplineTo()** uses the current position as the first anchor point, for this function the array should start with the first control point, and there should be $3n$ points passed.

24.2

P_0, P_3 : Anchor Points	P_1, P_2 : Control Points
S : a point on the curve	t : $0 \leq t \leq 1$
P_x, P_y : X, Y coordinate of point P	

$$S_x(t) = P_{0x}(1-t)^3 + P_{1x}(1-t)^2t + P_{2x}(1-t)t^2 + P_{3x}t^3$$

$$S_y(t) = P_{0y}(1-t)^3 + P_{1y}(1-t)^2t + P_{2y}(1-t)t^2 + P_{3y}t^3$$

$$S_x(t) = t^3(-P_{0x} + 3P_{1x} - 3P_{2x} + P_{3x}) + t^2(3P_{0x} - 6P_{1x} + 3P_{2x}) + t(-3P_{0x} + 3P_{1x}) + P_{0x}$$

$$S_y(t) = t^3(-P_{0y} + 3P_{1y} - 3P_{2y} + P_{3y}) + t^2(3P_{0y} - 6P_{1y} + 3P_{2y}) + t(-3P_{0y} + 3P_{1y}) + P_{0y}$$

Equation 24-1 Bézier Curve Equations

Coordinates of the points of a Bézier curve can be expressed by parametric equations. The equation for both x and y is given in two forms here, one a rearrangement of the other.

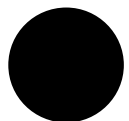


Advanced Topic

You don't need to know the math behind splines to draw them.

For most programmers, that's probably enough to know. Those programmers who want to know more and don't mind a bit of math may feel free to continue this section.

A curve is defined in terms of four points. There is a formula to determine the coordinates of all points on a spline in terms of these four points. The formula uses two parameterized cubic equations. These equations determine the x and y coordinates of a point on the curve. By finding the points corresponding



to various parameters, it is possible to approximate the spline as closely as necessary. See Equation 24-1 for the equations.

Splines may be created by drawing curves which share endpoints. Given an anchor point which two curves of a spline share, if the control point of one lies in the exact opposite direction of the other control point, the resulting spline will be smooth. If the control points are not only in the exact opposite directions but are also the same distance from the anchor point, then not only will the resulting spline be smooth, but its derivative will be smooth as well.

We call smooth splines with smooth derivatives “very smooth,” and this condition is analogous to C' continuity in functions. Smooth splines with non-smooth derivatives are called “semi smooth”, analogous to G' continuity.

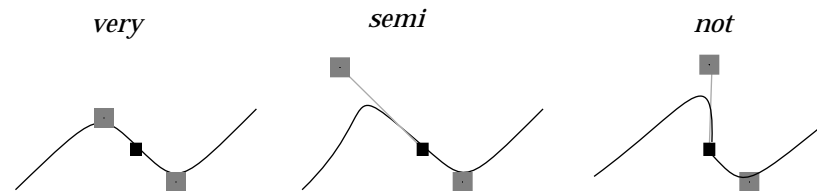


Figure 24-11 *Levels of Smoothness*

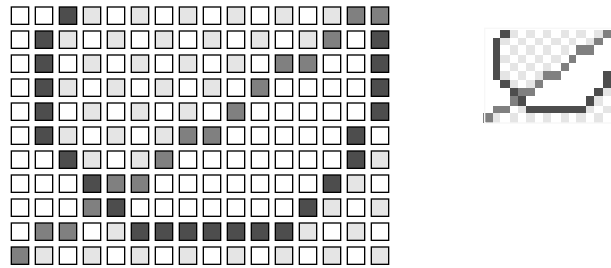
24.2.10 Drawing Bitmaps

```
GrDrawBitmap(), GrDrawBitmapAtCP, GrFillBitmap(),
GrFillBitmapAtCP(), GrDrawHugeBitmap(),
GrDrawHugeBitmapAtCP(), GrDrawImage()
```

In this section only drawing bitmaps will be discussed. For information on creating and modifying bitmaps, see section 23.7 of chapter 23.

GrDrawBitmap() draws a bitmap. This routine is very versatile. It can draw simple and complex bitmaps. It can draw compacted or uncompact bitmaps, can use a bitmap-specific palette, handles strange resolutions intelligently, and generally does the right thing. If you’re working with a large bitmap and want to manage its storage, you may provide a routine to pass in part of the bitmap at a time. If the bitmap is stored in a huge array (true of all bitmaps created with **GrCreateBitmap()**) use **GrDrawHugeBitmap()** instead of **GrDrawBitmap()**, and it will manage

memory for you. **GrDrawBitmapAtCP()** draws a bitmap at the current position.



24.2

Figure 24-12 *Bitmap*

A possible representation of a bitmap data structure and how that bitmap might be drawn.

If you just want to draw a monochrome bitmap, consider using the **GrFillBitmap()** command. This routine treats the bitmap like a mask, coloring the “on” pixels with the present area color, and leaving the “off” pixels alone so that whatever is underneath the bitmap can show through. This routine is heavily optimized and very fast. **GrFillBitmapAtCP()** works the same way, filling the bitmap at the current position.

Use **GrDrawHugeBitmap()** to draw a bitmap that has been stored in a HugeArray data structure. Remember that any bitmaps created by **GrCreateBitmap()** are stored in a **HugeArray**. **GrDrawHugeBitmap()** will automatically take care of memory management. **GrDrawHugeBitmapAtCP()** works the same way, drawing the bitmap at the current position. **GrFillHugeBitmap()** and **GrFillHugeBitmapAtCP()** fill huge bitmaps.

GrDrawImage() is less adaptable but faster than **GrDrawBitmap()**. **GrDrawImage()** has its own kind of scaling which doesn’t work in the standard GEOS fashion. This routine ignores the resolutions of both device and bitmap and displays the bitmap so that each pixel of the bitmap corresponds to one pixel of the display. If the coordinate system has been scaled or rotated, **GrDrawImage()** will ignore the scale and rotation when drawing the bitmap. The bitmap may be magnified, but this is not quite the same as normal scaling: The bitmap’s resolution is still ignored, but each pixel of the bitmap will receive a square of pixels on the display. **GrDrawHugeImage()** draws the image of a bitmap stored in a **HugeArray**.

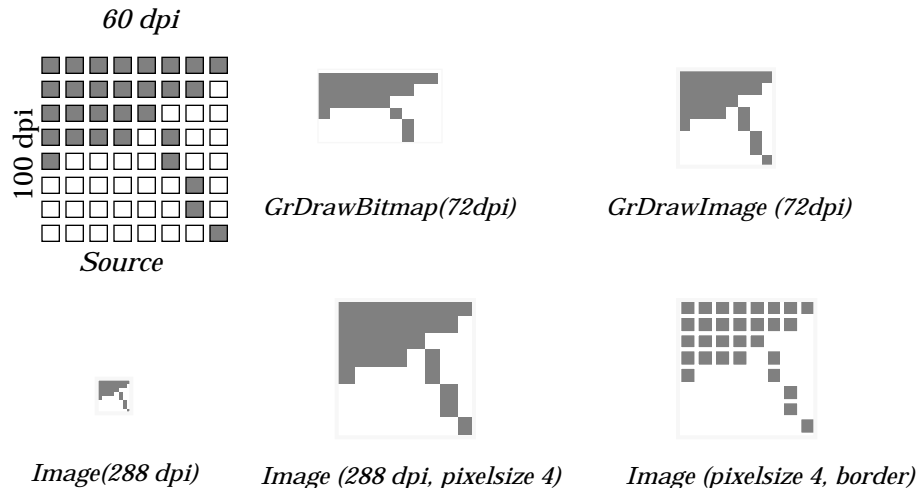


Figure 24-13 *GrDrawImage()*

GrDrawImage() is non-WYSIWYG; the output depends on the resolution.

The image-drawing routines take an **ImageFlags** structure which holds a flag to specify whether borders should be drawn between the pixels of the bitmap and a bit size field which specifies the magnification to use.

24.2.11 Paths

`GrDrawPath()`, `GrFillPath()`

This section will address only the subject of drawing paths. To find out how to create, combine, and modify paths, see section 23.9 of chapter 23.

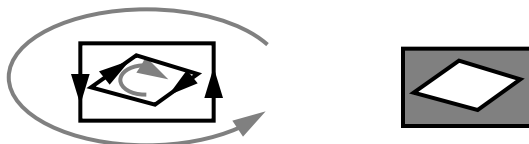


Figure 24-14 *Path Direction for Winding Fills*

The path should be counterclockwise overall, with clockwise concavities.

GrDrawPath() draws a path using the current line attributes.

GrFillPath() fills a path using the current area attributes. **GrFillPath()**

can fill the path using either an odd/even or winding fill rule. If the path is to be filled using the winding fill rule, the path must have been defined so that the segments forming the border of each region go around the region so that the interior of the region is to the left. That is, on convex parts of the border, edges should be in the counterclockwise direction. On concave parts of the border edges should go clockwise. For an illustration of a path following this rule, see Figure 24-14. The fill rule is specified by means of a **RegionFillRule** value, which may be one of ODD_EVEN or WINDING.

24.2

When you define a path by combining two other paths, the result might not be exactly what you would expect. See Figure 24-15 for an example of path intersection and union.

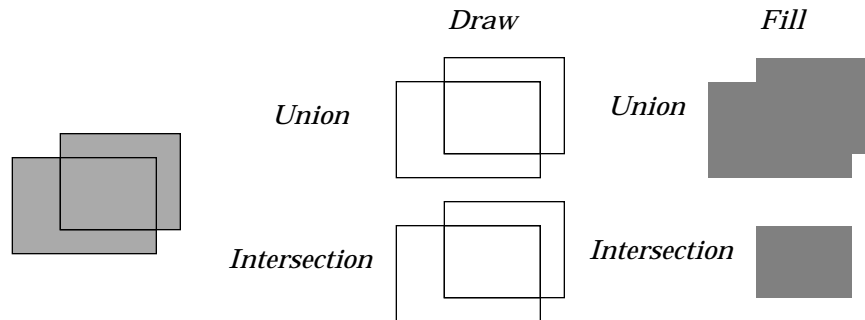


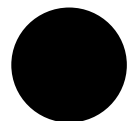
Figure 24-15 *Path Intersection and Union*

When drawing a path formed by union or intersection, both paths are drawn, perhaps not what was expected. The effects of a union or intersection aren't apparent unless the paths are filled (or used to define clip regions).

24.2.12 Regions

```
GrGetPathRegion(), GrDrawRegion(), GrDrawRegionAtCP(),
GrMoveReg(), GrGetPtrRegBounds(), GrTestPointInReg(),
GrTestRectInReg()
```

Sometimes it's useful to be able to describe an arbitrary area of a display. Regions provide a mechanism for doing so. Regions are ideal for delimiting large, blobby areas which are relatively free of detail. They are used by the system to describe many "filled shapes."



While it is possible to define a region directly, writers familiar with paths may define a path and then call the **GrGetPathRegion()** routine. To find out how to define a region directly, see below.

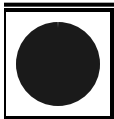
GrDrawRegion() draws a region at the passed position using the current area drawing attributes. **GrDrawRegionAtCP()** draws a region at the current pen position.

GrMoveReg() changes the coordinates stored within a region data structure by the specified *x* and *y* offsets.

GrGetPtrRegBounds() returns the coordinates of the passed region's bounding rectangle.

GrTestPointInReg() sees if the passed point falls within a region.

GrTestRectInReg() tests to see whether the passed rectangle falls entirely or partially within a region; it returns a value of type **TestRectReturnType** specifying the degree of overlap. These two functions are very useful when using regions for clipping purposes; if a given point or rectangle is discovered to be outside the clipping region, there's no need to draw the point or rectangle.



Advanced Topic

Some application writers may wish to define regions directly without describing a path. Regions are described in terms of a rectangular array (thus the similarity to bitmaps). Instead of specifying an on/off value for each pixel, however, regions assume that the region will be fairly undetailed and that the data structure can thus be treated in the manner of a sparse array. Only the cells in which the color value of a row changes are recorded. The tricky part here is keeping in mind that when figuring out whether or not a row is the same as a previous row, the system works its way up from the bottom, so that you should compare each row with the row beneath it to determine whether it needs an entry.

The easiest region to describe is the null region, which is a special case described by a single word with the value **EOREGREC** (a constant whose name stands for *End Of REGION RECORD* value). Describing a non-null region requires several numbers.

The first four numbers of the region description give the bounds of the region. Next come one or more series of numbers. Each series describes a row, specifying which pixels of that row are part of the region. The only rows which need to be described are those which are different from the row below.

The first number of each row description is the row number, its *y* coordinate. The last number of each series is a special token, EOREGREC, which lets the kernel know that the next number of the description will be the start of another row. Between the row number and EOREGREC are the column numbers where the pixels toggle on and off. The first number after the row number corresponds to the first column in which the pixel is on; the next number is the first subsequent column in which the pixel is off; and so on.

For example, Figure 24-16 shows a simple region, along with the numbers used to define it. Those rows which are the same as the rows beneath them have no entry in this structure. Notice that rows four through seven, being the same as row eight, have no entries.

24.2

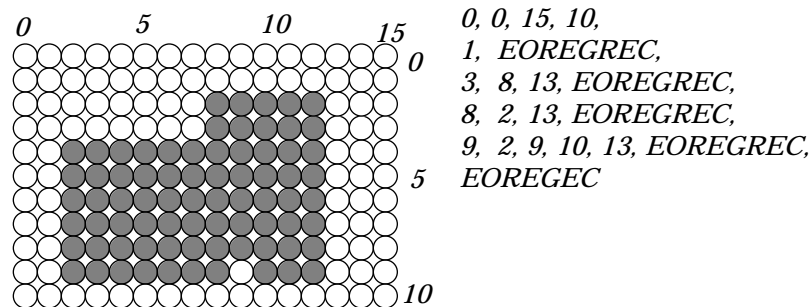
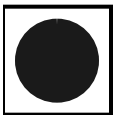


Figure 24-16 *Sample Region*



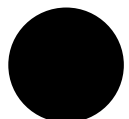
Advanced Topic

Parameters, while powerful, are not used by many programmers.

It is possible to customize a region by taking advantage of a feature of the coordinate system. Graphics routines take word-sized (16-bit) coordinate arguments. Normal coordinates only require 15 bits. When working with regions, the graphics system uses the extra bit to allow for coordinates that are described in terms of “parameters.”

When you create a region you can specify coordinates as an offset from a *parameter*. When the region is initialized, up to four parameters may be defined. Coordinates may then be specified as 13-bit offsets from any one of these four parameters. When drawing the construct, new values may be passed for the parameters. In this way, it is possible to use a single region to describe a variety of shapes, just by changing the parameters.

Coordinates with the values shown below will be interpreted as offsets.



4000h-5FFFh: Offsets from parameter zero, with 5000h corresponding to the parameter exactly.

6000h-7FFFh: Offsets from parameter one, with 7000h corresponding to the parameter exactly.

8000h-9FFFh: Offsets from parameter two, with 9000h corresponding to the parameter exactly.

A000h-BFFFh: Offsets from parameter three, with B000h corresponding to the parameter exactly.

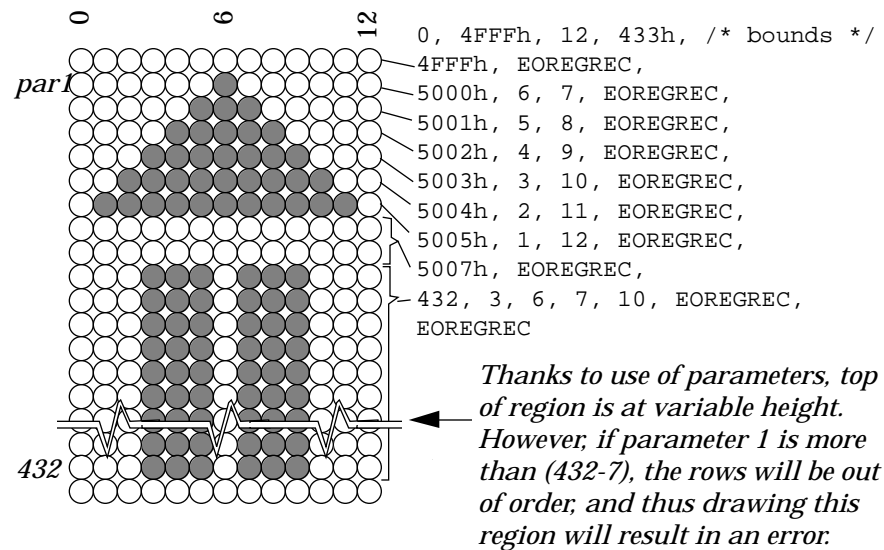
Thus, if a coordinate in a GString were 5000h, and the region were drawn with parameter zero equal to 72, then that drawing would take place at the coordinate 72. The coordinate 4FFFh would be interpreted as 71. Use the following constants to clarify parameterized coordinates:

```
/* Constants for DrawRegion */
#define PARAM_0 0x5000
#define PARAM_1 0x7000
#define PARAM_2 0x9000
#define PARAM_3 0xb000
```

Some or all coordinates of a region description may incorporate parameters. Note that the region code doesn't check regions for correctness. If the bounds of a region are set incorrectly, the rows are given out of order, or an incorrect (odd) number of on/off points is given for a row, the results are undefined. Figure 24-17 shows a complicated region which uses parametrized coordinates.

24.2.13 Text

Programs normally display text with UI gadgetry such as GenText, VisText, and GenGlyph objects. For those times when a geode will display text as part of a graphical display, sometimes it's best to display text using direct calls to the graphics system.



24.2

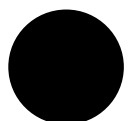
Figure 24-17 *Complicated Region Example*

This example shows how a rather unlikely region might be defined. This region is set up to draw an up-pointing stylized arrow. The base of the arrow will be six inches down from the coordinates passed to the region drawing command (432 points). The coordinate of the top of the arrow will be the first parameter. The arrow's horizontal position is determined by the second parameter. Remember that the coordinate 5000h is not absolute but corresponds to the first parameter, and all coordinates between 4000h and 5FFFh will be relative to this coordinate.

24.2.13.1 Displaying Text

```
GrDrawText(), GrDrawTextAtCP(), GrDrawChar(),
GrDrawCharAtCP(), GrDrawTextField()
```

There are several functions that display text. The **GrDrawText()** routine displays a text string. The passed *y* position, adjusted by the text mode (see below), determines the vertical position. The passed *x* position determines where the text will start printing, as normal.



GrDrawText() draws a text string at the desired position using the GState's current text attributes. This text string should contain no carriage returns, line feeds, tabs, or other non-printing characters. **GrDrawTextAtCP()** draws a text string at the current position. **GrDrawChar()** and **GrDrawCharAtCP()** draw a single character, which should not be a non-printing character. **GrDrawTextField()** draws a field of text—however, this routine is only available in Assembly language.

24.2

24.2.13.2 Special Text Attributes

```
GrGetTextStyle(), GrSetTextStyle(), GrGetTextMode(),  
GrSetTextMode(), GrGetTextSpacePad(), GrSetTextSpacePad(),  
GrGetFont(), GrSetFont(), GrGetTrackKern(),  
GrSetTrackKern(), GrGetFontWeight(), GrSetFontWeight(),  
GrGetFontWidth(), GrSetFontWidth(), GrGetSuperscriptAttr(),  
GrSetSuperscriptAttr(), GrGetSubscriptAttr(),  
GrSetSubscriptAttr()
```

Applications can display text in a number of ways. Thus the GState has many attributes it keeps track of solely for drawing text.

Text style is a collective set of attributes (bold, italic, etc.) that affects how the text is drawn by the graphics system. **GrGetTextStyle()** gets the current text style, and **GrSetTextStyle()** allows a new style to be specified. Styles are expressed as a **TextStyle** record. Note that some complicated styles which are offered by the text objects are not available here: these styles are available only from the text objects; if you wish to offer these styles without using a text object, you'll have to do the artwork yourself.

Depending on the text mode attribute, text may either be drawn from the bottom of the font box, top of the font box, baseline, or accent line.

GrGetTextMode() gets the text mode, returning information about which offset to use when drawing text. **GrSetTextMode()** allows this information to be reset. The information is stored in a **TextMode** record. Note that if you will be drawing characters of more than one size or font, and if you want those characters to line up by baseline, you should use **GrSetTextMode()** to use the **TM_DRAW_BASE** text mode.

GrSetTextSpacePad() sets the special amount used to pad space characters; **GrGetTextSpacePad()** retrieves the current space padding.



Figure 24-18 *TextMode and Drawing Position*

GrGetFont() returns the current font and type size. The font is identified by its **fontID**; the default font has the ID `DEFAULT_FONT_ID` and size `DEFAULT_FONT_SIZE`; these are the values which new GStates will start with. **GrSetFont()** sets a new font to use. The font's point size may be between `MIN_POINT_SIZE` and `MAX_POINT_SIZE`.

24.2

Track kerning adjusts the space between characters. A negative kerning value means that characters will be drawn closer together. A large negative kerning value can make characters draw overlapped. A positive kerning value causes characters to draw with more space between them.

GrGetTrackKern() returns the present track kerning value.

GrSetTrackKern() changes this value. The kerning value must be between `MIN_TRACK_KERNING` and `MAX_TRACK_KERNING`; values greater than `MAX_TRACK_KERNING` will be replaced by `MAX_TRACK_KERNING`, values less than `MIN_TRACK_KERNING` will result in `MIN_TRACK_KERNING` being used. The kerning value will be multiplied as a percentage by the font size to get a number of points to use for kerning; if this multiplied value is greater than the **BBFixed** (byte-byte fixed point) number `MAX_KERN_VALUE` or less than `MIN_KERN_VALUE` then it will be adjusted to fall at the end of this range.

A font's weight determines its boldness. For many fonts, there will be only two weights defined: plain and bold. However, some fonts allow for finer control of weight. To find out the current font weight, call

GrGetFontWeight(). To use a different font weight, call

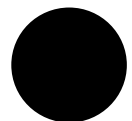
GrSetFontWeight(). Some useful weight values are stored in the

FontWeight enumerated type. The weight should definitely be between `FW_MINIMUM` and `FW_MAXIMUM`.

To make characters draw wider or narrower, adjust the font width. Some fonts come in wide or narrow versions. If the font does not support the requested width, GEOS will simulate the width as best it can. The

GrGetFontWidth() routine retrieves the current width;

GrSetFontWidth() changes it. Some helpful width values are stored in the



FontWidth enumerated type. The width should definitely be between FWI_MINIMUM and FWI_MAXIMUM.

Geodes can control how text will be drawn in superscript and subscript styles. The super- and subscript attributes determine how to scale the characters and how far they should be displaced. There are several standard super- and subscript attributes available, including values for footnote numbers, and chemical inferiors. Typesetting enthusiasts who wish to adjust the width of these characters differently than the height (as in some standard super- and sub- script layouts) should work with the font width. Use **GrGetSuperscriptAttr()** and **GrGetSubscriptAttr()** to find out what the present values are. Use **GrSetSuperScriptAttr()** and **GrSetSubscriptAttr()** to change these values. Each of these routines works with a word sized value: the top byte is a percentage of the font size to offset sub- or superscript characters; the low byte is the percentage of font size to use for the sub- or superscript character 0x0064 (decimal 100) would be full-sized with no displacement.

24.2.13.3 Accessing Available Fonts

```
GrEnumFonts(), GrCheckFontAvail(),  
GrFindNearestPointsize(), GrGetDefFontID(), GrGetFontName()
```

To find out which fonts are available in the user's environment, use the **GrEnumFonts()** command. You specify what sorts of fonts you're interested in by setting a number of flags, and the routine will fill a buffer with the available fonts with their **FontIDs** and names.

The **FontEnumFlags** record determines which fonts will be returned. At least one of the FEF_OUTLINES and FEF_BITMAPS flags must be set to determine whether outline, bitmap, or both kinds of fonts should be returned. Keep in mind that only outline fonts will result in true WYSIWYG printer output. The FEF_ALPHABETIZE flag will force the returned buffer of fonts to appear in lexical order. The FEF_DOWNCASE flag requests that the font names appear all in lowercase.

The FEF_FAMILY flag asks that the search be limited to a font family. To specify what sort of family you want, use the **FontFamily** enumerated type.

GrEnumFonts() ignores the FEF_STRING flag; other routines will use this flag to find out whether the font is specified by a **FontID** or its ASCII name.

The other flags narrow the search: if you set the `FEF_FIXED_WIDTH` flag, then only fixed-width fonts will be returned. If you set the `FEF_USEFUL` flag, only those fonts marked as “useful” will be returned.

The font information will be returned as an array of **FontEnumStruct** structures. Each of these structures will contain a **FontID** and the ASCII name of the associated font.

There may be up to `MAX_FONTS` available on the system. If you're not prepared to handle an array large enough to hold this many fonts, be sure to pass an argument specifying how large an array is being provided. 24.2

To find out if a given font is available in the user's environment, call either **GrCheckFontAvailID()** or **GrCheckFontAvailName()**, depending on whether the font is being identified by its **FontID** or its ASCII name. If programming in assembly language, use the **GrCheckFontAvail()** routine no matter how you're identifying the font. You may narrow the search by passing the appropriate **FontEnumFlags**. Make sure that the `FEF_STRING` bit is clear when searching by ID and set when searching by name.

Some fonts are available only in certain sizes and styles. Bitmap fonts are examples of this. The **GrFindNearestPointsize()** routine takes a typeface, size, and style set. It returns the closest available size and style (or returns `FID_INVALID` if the passed font isn't available).

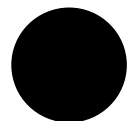
To get the font which the Generic UI will use by default, call **GrGetDefFontID()**. This also returns the font size to use, useful if you wish to respect the user's choice of type size.

To find out the ASCII name of a string for which you have the **FontID**, call **GrGetFontName()**. If the font isn't available, this function will return *false*; otherwise it will fill a passed buffer with the name of the font. The passed buffer should be `FID_NAME_LEN` bytes long.

24.2.13.4 Text Metrics

```
GrTextWidth(), GrTextWidthWBFixed(), GrCharWidth(),
GrFontMetrics(), GrCharMetrics()
```

From time to time it may become important to know something about the size of a letter to be drawn. The graphics system has several routines for



retrieving this information. **GrTextWidth()** returns the width, in points, of a string. **GrTextWidthWBFixed()** does the same thing, but returns a more accurate figure, including a fractional part. **GrCharWidth()** returns the width of a single character. Note that the width of a text string will normally be different than the sum of the widths of the component characters. **GrTextWidth()** takes track kerning and pairwise kerning into account.

GrFontMetrics() returns information pertaining to a font (see Figure 24-19). The meanings of these metrics are listed below:



Figure 24-19 *Common Font Metrics*

Height The height of the “font box.” This is how much vertical space should be allocated for a line of text. Note that when the text is drawn, some characters may go beyond these bounds. Though the height is still the vertical distance to set aside for a line of text, applications should be prepared for characters to have a vertical height equal to the maximum adjusted height, discussed below.

Maximum Adjusted Height

The maximum height required to draw the character. Some fonts contain characters that are meant to go beyond the normal vertical boundaries, perhaps extending into the space occupied by text on the next or previous line. This metric is the minimum height to guarantee being able to display such a character.

Above Box Normally the top of the font box, the uppermost limit of a font, is 3/4 of the font’s height above the baseline. For those fonts which do not follow this rule, the Above Box metric is the number of points by which the true font box extends beyond the normal font box.

Below Box Normally, the bottom of the font box, the bottommost limit of a font, is 1/4 of the font’s height below the baseline. For those

fonts which do not follow this rule, the Below Box metric is the number of points by which the true font box extends below the normal font box.

Mean The height of a typical lower case character. This metric is sometimes called the “x height,” since this will be the height of a lower case “x.”

Descent How far descending characters (like “y” and “j”) extend below the baseline.

24.2

Baseline The vertical position of the bottoms of nondescending characters. The number referred to as the “baseline” is the distance between the baseline and the top of accent marks. This corresponds to the text mode TM_DRAW_BASE.

Accent How much space to leave for accent marks. This distance is measured from the top of the accent mark area to the ascent height. This corresponds to the text mode TM_DRAW_ACCENT.

Ascent Height of capital letters (and some tall lower-case letters, such as “f”).

Underline Position

The distance from the top of the accent line to the top of the underline.

Underline Thickness

The width of the underline.

Strikethrough Position

The vertical position of the line used for “~~stri~~kethrough~~~~” text.

Average Width

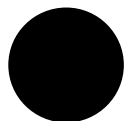
The average character width. This figure is normally computed as a weighted average width of lower-case characters, said average weighted based on the frequency distribution of the characters.

Maximum Width

Width of the font’s widest character.

Leading The height of the vertical gap between lines of text.

Kern Count The number of specially kerned pairs in the font. Many fonts allow certain pairs of characters to have special spacing, known as “pair kerning.” For example many fonts try to squish “To” so



that the “T” and “o” are closer together than normal leading would dictate. Do not confuse kerned pairs with ligatures. A kerned pair just has strange spacing (e.g. “To” vs. “To”); a ligature is a pair of characters which have been combined into one (e.g. “fi” and “æ”).

Driver The font driver associated with the font, specified by a member of the **FontMaker** enumerated type.

24.2

First Character

The **Char** value of the first defined, drawable character.

Last Character

The **Char** value of the last defined, drawable character.

Default Character

The **Char** value of the drawable character which will be drawn as a substitute for any undrawable characters.

The **GrCharMetrics()** routine returns useful measurements for a character. For each of a font's characters, you may request the following information:

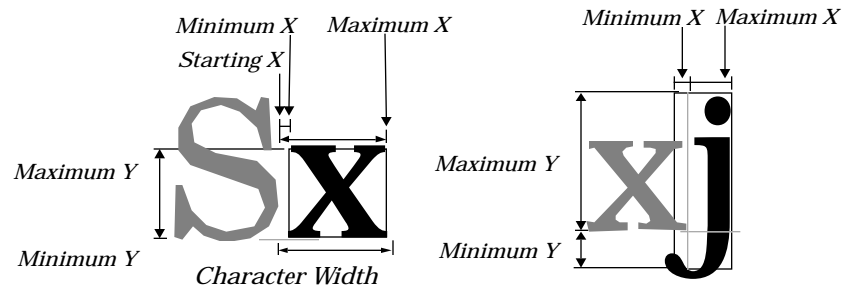


Figure 24-20 *Character Metrics*

Unlike most characters, this font's “j” has a negative “Minimum X,” meaning it extends backwards into the space allotted to the previous character.

Width

While not a metric returned by the **GrCharMetrics()** routine, the character width is very important. It determines how much space to allow for the character. The metrics that follow don't affect how much space to allow; instead, they give the bounds of the area that will be drawn to. As shown in Figure 24-19, characters are free to draw beyond the area set aside for them by the width, into the space of the previous or next character.

Minimum X

The character's leftmost bound. If this value is positive, it means that the character will have some extra space before it. If the value is negative, the character may infringe on the space of the previous character. This metric is also called the "left side bearing."

Minimum Y

The character's bottommost bound. This is the character's "descent," the amount it hangs below the baseline.

24.3

Maximum X

The character's rightmost drawing bound. This value should be greater than the Minimum X. Do not confuse this metric with the character's width.

Maximum Y

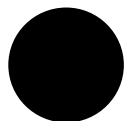
The character's topmost bound. This is the distance the character extends above the baseline. It is sometimes called the character's "ascent."

24.3 Shape Attributes

`GrSetAreaAttr()`, `GrSetLineAttr()`, `GrSetTextAttr()`

The graphics system keeps track of many attributes which affect how graphics will be drawn. By setting attributes in the graphics state, your geode can change the color, fill pattern, and other features that won't affect the shape of what you're drawing but will affect its appearance.

The following sections will explain how each of these attributes may be set individually. Note that if you wish to set all attributes to use when filling areas at once, call **GrSetAreaAttr()**. Call **GrSetLineAttr()** to set all attributes to use when drawing lines. Use **GrSetTextAttr()** to set all text-rendering attributes.



24.3.1 Color

If your geode displays graphics, you can probably make its graphical display more effective by using color. Your geode can have as much control over color as it wants, ranging from saying what colors certain objects should be to choosing which colors should be available.

24.3

Most color controllers work with the concept of a palette, a subset of the colors your video device is capable of displaying. Most controllers can only use some of these colors at a time; these colors make up the palette. You can refer to colors by their color index (i.e. their place in the palette). You can also refer to colors by their RGB values.

An RGB value specifies a color by mixing intensities of red, green, and blue. The more of each component color added, the more of that color will show up in the result. If all components are equal, the result will be a shade of gray. If all components are zero, the result will be black. If all components are at the maximum (255), the result will be white.

The data structure used to store color information is known as the **ColorQuad**, shown in Code Display 24-1.

Code Display 24-1 Color Data Structures

```
typedef struct {
    /* The ColorQuad data structure is used to represents a color. There are
    * many ways to describe a color. The CQ_info field determines how the
    * color is being specified, and the other fields will be interpreted
    * based on CQ_info's value. Colors may be referenced by palette index,
    * RGB value, or grey scale value. */
    /* CQ_redOrIndex
    * If CQ_info is CF_INDEX, then this is the palette index.
    * If CQ_info is CF_RGB, then this is the Red component.
    * If CQ_info is CF_GRAY, then this is the Gray scale */
    byte    CQ_redOrIndex;

    /* CQ_info:
    * This ColorFlag determines how the other three fields of the
    * ColorQuad will be interpreted. The ColorFlag type is shown
    * below. */
    ColorFlag CQ_info;
}
```

```

        /* CQ_green:
        * If CF_RGB, then these fields are the Green and Blue components.
        * Otherwise, these fields are ignored. */
        byte    CQ_green;
        byte    CQ_blue;
    } ColorQuad;

typedef dword ColorQuadAsDWord;

typedef ByteEnum ColorFlag;
    CF_INDEX,
        /* Color specified by palette index. The values of the first
        * 16 entries of the system palette are listed in Table 24-1 */
    CF_GRAY,
        /* Color specified by gray value; this is like CF_RGB, but the
        * value in CQ_redOrIndex will be used for the Green and Blue
        * fields as well. */
    CF_SAME,
        /* Used with hatch patterns, if this flag is set, hatches will draw
        * using the default color (the one set using GrSetXXXXColor()) */
    CF_RGB
        /* Color Set using RGB values */

/* Sample Colors:
* To use the system palette's light green:      { C_LIGHT_GREEN, CF_INDEX, 0, 0}
* To use the 40th color in the palette:         { 40, CF_INDEX, 0, 0}
* To use a custom brown:                       { 150, CF_RGB, 90, 0}
* To use a 75% Gray (75% *256 = 192):          { 192, CF_GRAY, 0, 0}
*/

```

24.3

24.3.1.1 Using Available Colors

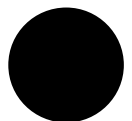
```

GrGetAreaColor(), GrSetAreaColor(), GrGetLineColor(),
GrSetLineColor(), GrGetTextColor(), GrSetTextColor()

```

If a geode is to draw something in color, it first calls a routine to set which color to use, then calls the appropriate drawing routine. The color can be specified in terms of its palette index or its RGB value. The data specifying which color to use is stored in the GState, with separate color fields to keep track of how to draw lines, areas, and text.

GrSetLineColor() changes the color used when drawing lines. The **GrSetAreaColor()** routine sets the color to be used when filling areas.



GrSetTextColor() sets the color to be used when rendering text. **GrGetLineColor()**, **GrGetAreaColor()**, and **GrGetTextColor()** return the color values and flags stored with the GState.

Table 24-1 *Convenient Color Indexes*

Constant Name	Index	RED	GREEN	BLUE
C_BLACK	0x00	0x00	0x00	0x00
C_BLUE	0x01	0x00	0x00	0xAA
C_GREEN	0x02	0x00	0xAA	0x00
C_CYAN	0x03	0x00	0xAA	0xAA
C_RED	0x04	0xAA	0x00	0x00
C_VIOLET	0x05	0xAA	0x00	0xAA
C_BROWN	0x06	0xAA	0x55	0x00
C_LIGHT_GRAY	0x07	0xAA	0xAA	0xAA
C_DARKK_GRAY	0x08	0x55	0x55	0x55
C_LIGHT_BLUE	0x09	0x55	0x55	0xFF
C_LIGHT_GREEN	0x0A	0x55	0xFF	0x55
C_LIGHT_CYAN	0x0B	0x55	0xFF	0xFF
C_LIGHT_RED	0x0C	0xFF	0x55	0x55
C_LIGHT_VIOLET	0x0D	0xFF	0x55	0xFF
C_YELLOW	0x0E	0xFF	0xFF	0x55
C_WHITE	0x0F	0xFF	0xFF	0xFF

*These are the first 16 members of the **Color** enumerated type. For a full list of available **Color** values, see the Routines reference or **color.h**.*

The default system palette includes several colors whose indexes have been set up so they may be referenced by descriptive constant names. The constants are members of the enumerated type **Color**, the most common shown in Table 24-1. Thus, instead of having to remember that the index 02 means green, the constant C_GREEN can be passed to the appropriate color setting command.

Other Color values include a 16-shade gray scale (the C_GRAY_...entries), some “unused” entries (the C_UNUSED_... entries), and a number of entries which have been set up to allow you to specify a color by its red, green, and blue components on a zero-to-five scale (the C_R..._G..._B... entries).

Programmers should use care when using these constants in conjunction with palette manipulation, as it is possible to change the RGB color value associated with a palette entry. Since the constant names are associated with palette indexes instead of RGB values, it is possible to change the RGB value

of palette entry 02 so that the `C_GREEN` constant actually refers to, for example, a shade of magenta.

24.3.1.2 When the Color Isn't in the Palette

```
GrSetLineColorMap(), GrSetAreaColorMap(),
GrSetTextColorMap(), GrGetLineColorMap(),
GrGetAreaColorMap(), GrGetTextColorMap(),
GrMapColorIndex(), GrMapColorRGB()
```

24.3

Not all users have video devices which can display 256 colors at a time. Even fewer have printers capable of doing so. Therefore, the graphics system makes allowances for drawings which use unavailable colors. Exactly what the system will do with an unavailable color depends on the color mapping mode used. By default, the system will choose the two (or more) closest available colors and dither them together in an attempt to form a mixture which will appear to be the correct color overall when seen from a distance. (See Figure 24-21.)

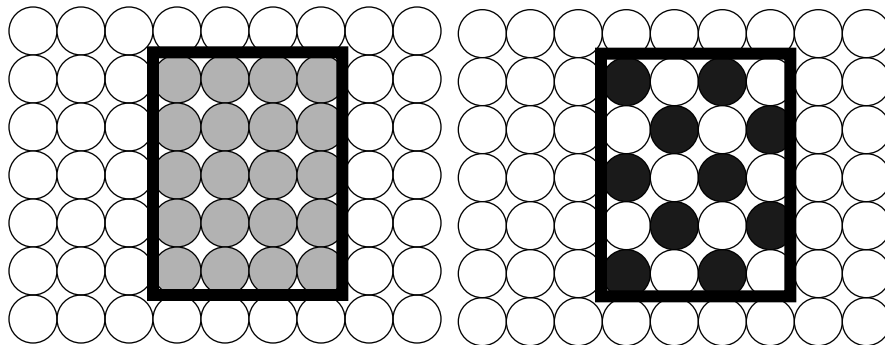
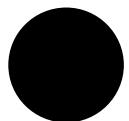


Figure 24-21 *Dithering*

Here the system wants to display the rectangle with a light gray. On a black and white system, it would simulate the gray by setting some pixels black and others white.

GrSetLineColorMap(), **GrSetAreaColorMap()**, and **GrSetTextColorMap()** set the color mapping to use when drawing with an unavailable color. **GrGetLineColorMap()**, **GrGetAreaColorMap()**, and **GrGetTextColorMap()** return the current color mapping modes. The geode



may draw either in the closest available color, or using a dithering of the closest colors. Often it's a good idea to have text map to solid colors since its detailed nature doesn't go too well with dithering.

When a geode wants to learn more about which colors are available, it can use the **GrMapColorIndex()** routine to find the RGB value associated with an index. The **GrMapColorRGB()** routine returns the index and true RGB color of the palette entry that most closely matches the values passed.

24.3

24.3.1.3 Custom Palettes

```
GrCreatePalette(), GrDestroyPalette(), GrSetPaletteEntry(),  
GrSetPalette(), GrGetPalette()
```

If the system default palette does not meet a geode's needs, the geode can change its own palette. Some bitmaps (such as those in GIF) have palettes associated with them, and a geode displaying such bitmaps might wish to display the bitmap in its true colors without settling for the closest defaults. Specialized tasks such as photo processing, anti-aliasing, and color cycle animation tend to depend on the ability to manipulate palettes. A geode can create a custom palette associated with a window, then change the color values of entries in that palette.

Since the palette is associated with a window, there can be as many custom palettes as there are windows. The video driver uses the palette associated with the active window. This means that all the inactive windows (and the background) will also be drawn with the palette of the active window; thus, if you give a window a special palette, that may make inactive windows look different. To avoid this, try to avoid changing palette entries 0-15, as the UI uses these (These are the entries representing C_RED, etc.).

GrCreatePalette() creates a custom palette and associates it with your graphic state's associated window. The custom palette starts with all entries having their original RGB values from the default palette.

GrSetPaletteEntry() takes a palette entry and sets it to a new RGB value.

GrSetPalette() allows you to specify new values for several palette entries at a time. **GrSetPalette()** can also set an entry back to its default value.

GrDestroyPalette() destroys the custom palette, freeing the memory used for it.

Given a choice of palette entries to change, you might choose one of the `C_UNUSED_...` entries.

Call **GrGetPalette()** to get a handle to a memory block containing the palette data. Depending on the **GetPalType** passed, you will either get the palette information for your current palette or the default palette. This routine returns the handle of a memory block. The memory block contains an array with 256 entries (one for each color), each entry consisting of three bytes. The first byte of each entry is the color's red component, the second is the color's green component, and the third is the color's blue component.

24.3

24.3.2 Patterns and Hatching

```
GrSetAreaPattern(), GrSetAreaPatternCustom(),
GrSetTextPattern(), GrSetTextPatternCustom(),
GrGetAreaPattern(), GrGetTextPattern()
```

Fill patterns allow the application to tile an area with a repeating pattern of bits or lines helpful for suggesting textures. The graphics system supports two types of fill patterns. Bitmap patterns, familiar to most computer users, tile the filled area with a repeated bitmap. Hatch patterns fill the area with a repeated sequence of lines. Hatch patterns are defined in terms of families of parallel lines. Patterns are referenced by a **PatternType** and an index, stored in a **GraphicPattern** structure. The pattern types are

PT_SOLID The lack of a pattern. Fills the area solid. This is the default.

PT_SYSTEM_HATCH

System-defined hatch pattern. These patterns are unchangeable and are available to all geodes.

PT_SYSTEM_BITMAP

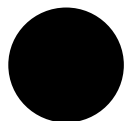
System-defined tile bitmap pattern. These patterns are unchangeable and available to all geodes.

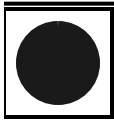
PT_USER_HATCH

User-defined hatch pattern. These patterns are available to all geodes.

PT_USER_BITMAP

User-defined tile bitmap pattern. These patterns are available to all geodes.





Advanced Topic

Most applications won't create custom patterns.

PT_CUSTOM_HATCH

Application-defined hatch pattern. These patterns are application-specific.

PT_CUSTOM_BITMAP

Application-defined tile bitmap pattern. These patterns are application-specific.

Use **GrSetAreaPattern()** and **GrSetTextPattern()** to use patterns defined outside the application (system- and user-defined patterns). To use the system's brick hatch pattern, for example, pass the **PatternType** `PT_SYSTEM_HATCH` and the **SystemHatch** `SH_BRICK`. To use a user-defined bitmap pattern, pass **PatternType** `PT_USER_BITMAP` and the number of the pattern. If you pass an invalid pattern (requesting a user hatch pattern when the user hasn't defined one, for instance), the area or text will be filled solid.

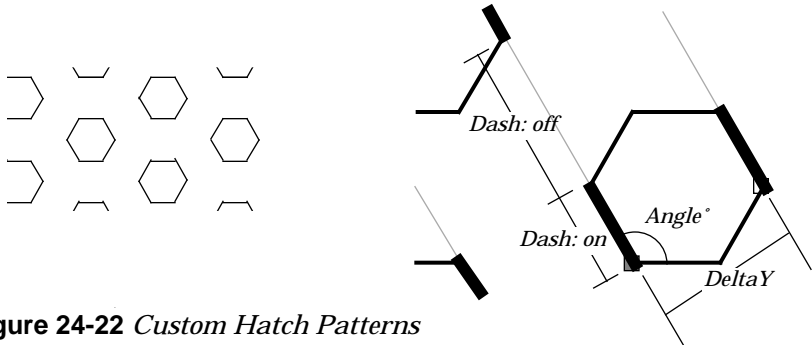
Applications may define their own patterns. Before adding custom patterns to an application, consider whether such an action is really necessary. Remember that the user may define his own patterns. The data associated with any application custom pattern may be at most 16Kbytes.

Hatch patterns are defined in terms of families of lines. The pattern designer specifies a series of families; each family consists of a set of equidistant parallel lines. Thus, by defining one family of lines, one could cover a surface with one set of parallel lines. By asking for two families, the region could be filled with a grid. For an example of a possible custom hatch pattern, see Figure 24-22.

For each line family, the application must supply certain information. Your application will work with the **HatchLine** data type to specify

◆ Origin

Since each line family is drawn as a set of equidistant parallel lines, the origin might seem useless. Who cares where the lines start, since they fill the available space? In fact, the origin will only be meaningful if your hatch pattern contains more than one line family. The origin allows you to draw one line family at an offset from another. In the case of the example presented in Figure 24-22, the third line family needed an origin offset from that of the first two families, since none of the lines of the third family pass through the point used as the origin by the first two families.



24.3

Figure 24-22 Custom Hatch Patterns
This simple hatch pattern is made up of three line families. The measurements used to compute the 120° family are illustrated to the right. Full measurements are shown below.

Origin	DeltaX, DeltaY	Angle	numDash; Dashes
(0, 0)	0, 12*sqrt(3)	0	1; 12, 24
(0, 0)	0, 12*sqrt(3)	120	1; 12, 24
(12, 0)	0, 12*sqrt(3)	60	1; 12, 24

- ◆ Delta offset between lines of the family
Each family has a horizontal and vertical offset. When drawing the pattern, the first line will be drawn starting at the origin. The second line will be drawn at a perpendicular distance equal to the passed y offset. The second line may also be drawn at a parallel offset, but this will only affect dotted lines.
- ◆ Angle at which to draw the lines
Remember that angles are measured in degrees measured counterclockwise from the positive x axis.
- ◆ Color
You may draw the lines using the default color (area color if pattern is filling an area; text if rendering text), or you may draw using a specific color.
- ◆ Dashes
You may draw the lines using a custom dash pattern. See the Line Style attribute, below, to learn how to set up a dash pattern.

Custom bitmap patterns are defined in terms of simple bitmaps. To find out the structure of a bitmap, see Bitmap on page ■ 454 of the Routine Reference.

To use a custom pattern, call **GrSetCustomAreaPattern()** or **GrSetCustomTextPattern()**. Along with the usual information, you must include a pointer to a memory location which marks the beginning of some structures holding the pattern data. The commands and structures are detailed in the reference manual. For an example of some code using a custom hatch pattern, see Code Display 24-2.

24.3

Code Display 24-2 Hatch Pattern Data

```
/* This example shows how to implement the pattern illustrated in Figure 24-22 */
/* ... */
GrSetPatternCustom(myGState, gp, hexHatchPatt);
/* ... */

GraphicPattern gp = {PT_CUSTOM_HATCH, 0};

static HatchPattern hexHatchPatt = { 3 };          /* Three HatchLine structures
                                                    * must follow */

static HatchLine line1 = { {MakeWWFixed(0) , MakeWWFixed(0)}, /* Origin */
                          MakeWWFixed(0),          /* Delta X
                                                    * dashes will be in alignment */
                          MakeWWFixed(20.7846097), /* Delta Y
                                                    * lines will be 12*sqrt(3)
                                                    * apart */
                          MakeWWFixed(0),          /* Angle */
                          (dword) (CF_SAME<<16), /* Color
                                                    * will use default color */
                          1                        /* Number of dashes
                                                    * one HatchDash pattern
                                                    * must follow */
                          };

static HatchDash dash1 = {{12, 0},                /* On for 12 points */
                          {24, 0}};               /* ...and Off for 24 points */
```

```

static HatchLine line2 = { {MakeWWFixed(0), MakeWWFixed(0)}, /* Origin */
                          MakeWWFixed(0),          /* Delta X */
                          MakeWWFixed(20.7846097), /* Delta Y */
                          {120,0},                 /* Angle */
                          (dword) (CF_SAME<<16),   /* Color */
                          1                          /* Number of dashes */
                          };

static HatchDash dash2 = {{12, 0},                 /* On for 12 points */
                          {24, 0}};                /* ...and Off for 24 points */

static HatchLine line3 = { {{12,0} , MakeWWFixed(0)}, /* Origin */
                          * this line family will be at
                          * a 12 pt. horizontal offset
                          * from the other two families.
                          MakeWWFixed(0),          /* Delta X */
                          MakeWWFixed(20.7846097), /* Delta Y */
                          {60,0},                 /* Angle */
                          (dword) (CF_SAME<<16),   /* Color */
                          1                          /* Number of dashes */
                          };

static HatchDash dash3 = {{12, 0},                 /* On for 12 points */
                          {24, 0}};                /* ...and Off for 24 points */

```

24.3

To find out the current area or text pattern, call **GrGetAreaPattern()** or **GrGetTextPattern()**.

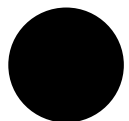
24.3.3 Mix Mode

`GrGetMixMode()`, `GrSetMixMode()`

The kernel supports several mix modes. These modes control what will happen when something is drawn on top of something else. Normally when this occurs the new drawing covers up the old one. This is known as **MM_COPY**, since the new drawing is copied directly onto the graphics space.

There are many other mix modes available (see Figure 24-23):

MM_COPY **MM_COPY** is the most common mix mode as well as the default. It draws the new drawing on top of the old as though the new



24.3

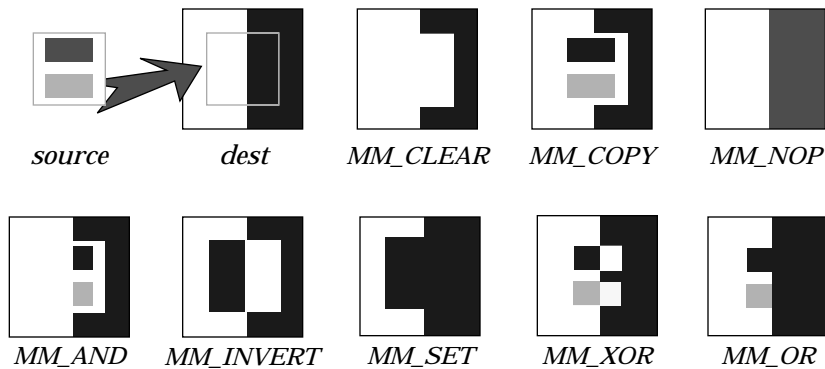


Figure 24-23 *Mix Modes*

drawing where on opaque paper—nothing of the old drawing shows through the region defined by the new drawing.

- MM_CLEAR** The region defined by the new drawing is blanked out. The color of the new drawing does not matter.
- MM_SET** The region defined by the new drawing is washed black. The color of the new drawing does not matter.
- MM_INVERT** The region defined by the new drawing is inverted. The color value displayed will be the logical inverse of the old color. The color of the new drawing does not matter.
- MM_NOP** The old drawing remains; nothing is drawn of the new drawing. Just about the only thing likely to change is that the pen position will be updated.
- MM_XOR** The color value to display is calculated by taking the bitwise XOR of the new and old colors. If painting in white, this acts as a sort of reverse paint. Note that this bitwise XOR is applied to the index numbers of the colors, not to their RGB values.
- MM_OR** The color value to display is calculated by taking the bitwise OR of the new and old colors. On a monochrome display, this corresponds to drawing on a clear plastic overlay. The OR operation is applied to the indexes of the colors as opposed to their RGB values.

MM_AND The color value to display is calculated by taking the bitwise AND of the new and old colors. Where either drawing is blank, the result will be blank. On a monochrome display, this results in a “clipping” effect. The AND operation is applied to the indexes of the colors.

GrGetMixMode() returns the current drawing mode. **GrSetMixMode()** tells the system to use a new mode.

Note that the drawing modes that use logical operators to compute the color have some nice effects when used with the default system colors. Due to the values of the indexes of the gray colors, MM_OR and MM_AND used with the gray colors can be thought of as lightening and darkening operations. For instance, AND-ing C_LIGHT_GRAY together with C_LIGHT_RED color results in the darker C_RED.

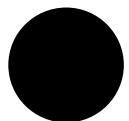
24.3

When using drawing modes, it often pays to be aware of where a drawing command is drawing something as blank as opposed to where it isn't drawing at all. Filling a rectangular area of the screen with a black and white checkerboard pattern isn't exactly like drawing only the corresponding black squares. When using MM_COPY, the white areas of the checkerboard will overwrite whatever was underneath. When drawing a checkerboard with individual black squares, the background would show through no matter what background because there's nothing drawn between the squares.

24.3.4 Masks

```
GrSetAreaMaskSys(), GrSetAreaMaskCustom(), GrGetAreaMask(),
GrSetLineMaskSys(), GrSetLineMaskCustom(), GrGetLineMask(),
GrSetTextMaskSys(), GrSetTextMaskCustom(), GrGetTextMask()
```

Masks are the pixellated equivalent of hatch patterns. Instead of specifying a set of dashed lines, the program uses an 8x8 bitmap array which will be tiled. The graphics system provides a number of standard mask patterns, including 65 percentage masks (for achieving, e.g., a 50% fill) and several shape patterns. It's simple to ask that the inverse of a system pattern be used instead of the pattern itself. The program may also specify and use custom draw masks by setting an 8 byte array.



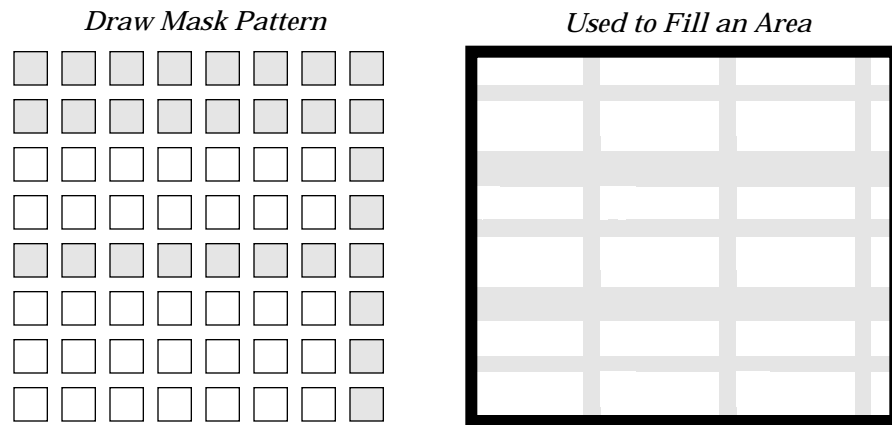


Figure 24-24 *Masks as Repeating Patterns*

A drawing mask is a monochrome bitmap. It is repeated to cover an area.

The **GrSetAreaMaskSys()** routine specifies one of the standard draw masks to use when filling areas. **GrSetAreaMaskCustom()** allows the program to specify a custom mask by passing a pointer to an 8 byte. Each byte of the buffer represents one row of the mask; the rows are ordered from top to bottom. **GrGetAreaMask()** returns the current area mask.

There are similar routines which set the drawing mask to use when drawing lines: **GrSetLineMaskSys()**, **GrSetLineMaskCustom()**, and **GrGetLineMask()**. To work with the drawing mask used when rendering text, use **GrSetTextMaskSys()**, **GrSetTextMaskCustom()**, and **GrGetTextMask()**.

For the most part, it's probably not a good idea to use masks when drawing text, since the pixelated nature of the mask is likely to make the text less legible. However, when drawing text in very large point sizes, draw masks are probably fine.

Since several of the masks are associated with a fill percentage, some programmers may confuse masks with gray-scale fills. Grays should be implemented with colors, by drawing in a color with equal red, green, and blue values.



Figure 24-25 *Drawing with Masks*

Before drawing the rectangles, the dark line was present. The line shows through both the light and dark rectangles. Each rectangle is drawn with a 50% mask.

24.3

When you are building a document image, do not use draw masks just to achieve a lighter color. That is, if you are not making use of the “mask” property of draw masks (i.e., you don’t care if you can see through it), then you should just specify a lighter color and let the kernel do the dithering. This is important especially because drawing masks in PostScript takes much longer than drawing a bitmap of the same size.

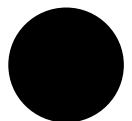
When constructing or interpreting system draw mask values, first remember that if the `SDM_INVERSE` flag is set, then the inverse of the mask will be used instead of the mask itself. The remaining bits should be a

SystemDrawMask value, a value between zero and `SDM_0`. Draw mask values between zero and 24 are various graphic draw mask patterns. Values between 25 and 89 are percentage fill masks, 89 corresponding to the empty fill and 25 to a 100% fill. Several `SDM_...` constants have been set up with the more commonly used system draw patterns. For a list of constants, see the Routines manual.

24.3.5 Line-Specific Attributes

```
GrGetLineWidth(), GrSetLineWidth(), GrGetLineJoin(),
GrSetLineJoin(), GrGetLineEnd(), GrSetLineEnd(),
GrSetMiterLimit(), GrGetLineStyle(), GrSetLineStyle(),
GrSetLineAttr()
```

Line attributes determine characteristics of lines drawn with the **GrDraw...()** routines. They do not affect anything drawn with the **GrFill...()** routines.



You may work with the width used when drawing lines by means of the **GrGetLineWidth()** and **GrSetLineWidth()** routines.

The graphics system makes it easy to draw dotted lines. The line style attribute controls the “dottedness” with which lines should be drawn; it may be changed or retrieved with the **GrSetLineStyle()** and **GrGetLineStyle()** routines. Lines can be drawn using any of a number of standard system **LineStyles** or by defining a custom dot pattern.

24.3

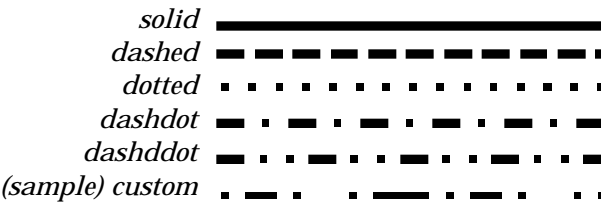


Figure 24-26 *LineStyles*

Custom dot patterns are defined in terms of arrays of pairs of bytes. The first byte of each pair gives the distance that the line should be drawn with the pen down, the second number the distance with the pen up. The array as a whole is stored in a **DashPairArray**. Dash lengths will scale with the line width.

Table 24-2 *Arrays for System Line Styles*

Name	#pairs	on/off pairs
LS_DASHED	1	4 4
LS_DOTTED	1	1 2
LS_DASHDOT	2	4 4 1 4
LS_DASHDDOT	3	4 4 1 4 1 4

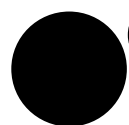
Line joins govern the behavior of angles and corners. Using the appropriate line join style, a geode can specify that angles should be blunt, pointed, or rounded. **GrGetLineJoin()** returns the current line join drawing technique, while **GrSetLineJoin()** sets a new line join to use. The join is specified by a member of the **LineJoin** enumerated type. The miter limit governs the maximum length of mitered joins; use **GrSetMiterLimit()** and **GrGetMiterLimit()** to work with this value.



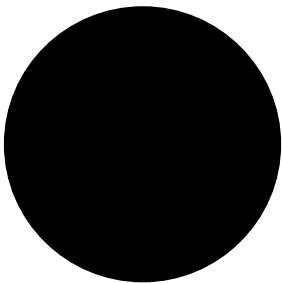
The **GrSetLineEnd()** and **GrGetLineEnd()** routines set and retrieve the style with which the graphics system will draw the ends of line segments. The ending style is one of the **LineEnd** values. It may be round, square and even with the end of the line, or square and extending past the mathematical end of the line.

To set all of the line-drawing attributes at once, call **GrSetLineAttr()**.

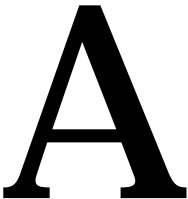
24.3



Machine Architecture



A.1	History of the 80x86.....	905
A.2	8086 Architecture Overview	906
A.2.1	Memory	906
A.2.2	Registers.....	908
A.2.3	The Prefetch Queue	909
A.2.4	Inherent Optimizations.....	910





The GEOS operating system is state-of-the art technology based on the Intel 80x86 microprocessor. This chapter discusses some of the elements of the 80x86 design and history.

Those unfamiliar with the 8086 architecture would do well not only to read this overview but also to pick up a book about the Intel chip series. There are dozens of good books about the 80x86 chips, and this section provides only a brief review.

A.1

A.1 History of the 80x86

The commercial microcomputer essentially began with Intel's introduction of the 8008 chip in 1972. This was an 8-bit machine that eventually led to the 8080 (in 1974), which was the precursor to the processors that now provide the power of today's PCs.

The 8080 was an 8-bit processor that used seven general-purpose registers and an external 8-bit bus. It addressed 64K of memory; such a small memory space necessitated optimization of each instruction in an attempt to keep as many instructions as possible under one byte in length. This optimization resulted in many instructions being forced to work on specific registers; for example, most arithmetic and logical operations used the accumulator as the destination register.

In 1978, the 8086 followed, bringing 16-bit architecture to microcomputers. The designers of this new chip, however, wanted to aid in quick software development, so they gave the 8086 an instruction set and architecture reminiscent of the 8080—this allowed software based on the earlier chip to be ported to the new hardware quickly and cheaply. The 8086 offered significant advances, most notably the ability to access up to one megabyte of memory in 64K segments.

A year later, in 1979, the 8088 was introduced. This chip had all the advanced features of the 16-bit 8086 except one: Rather than using a 16-bit external data bus, the 8088 used the same 8-bit bus used in the 8080. This step backwards made it possible for systems and peripherals designed for

use with the 8080 to be used with a faster, more powerful chip. The 8088 was built into IBM's personal computers for just this reason.

Both the 8088 and the 8086 can run the same software; they use the same instruction set. However, the power of the data processing offered by the 8086 eventually won out as systems and software became more complex.

A.2

A.2 8086 Architecture Overview

The 8086 uses a 16-bit architecture but can handle 8-bit data as well. It accesses up to one megabyte (1024K) of memory, sixteen times more than the 8-bit 8080 could. It has thirteen registers (each 16 bits) plus a status register containing nine flags.

A.2.1 Memory

The memory of the 8086 begins at hexadecimal address 0x00000 and continues (each increment representing one byte) to hexadecimal address 0xFFFFF. Any two consecutive bytes constitute a word. The address of a word is the same as the address of its low byte (see Figure A-1).

High Byte	Low Byte	
addr: 30FE	addr: 30FD	Word address: 30FD
addr: 30FC	addr: 30FB	Word address: 30FB

Figure A-1 *Structure of a Word*

A word consists of two bytes, the high byte having the higher address.

Both the 8088 and 8086 have instructions that access and manipulate words, and both have instructions that access and manipulate bytes. The 8088, however, always fetches a single byte from memory due to its 8-bit bus. The 8086 always fetches two bytes, or a word; if the instruction only operates on a byte, the excess 8 bits will be ignored. This information may be useful to assembly programmers who wish to optimize the performance of their code, but C programmers can ignore it.

As stated earlier, the 8086 can access up to one megabyte of memory. This translates to 2^{20} bytes, or 2^{20} addresses. However, because the 8086 is designed to do 16-bit arithmetic, it can not directly access all that memory. Instead, an additional mechanism, known as *segmentation*, is employed.

A *segment* is a contiguous set of bytes no larger than 64K (2^{16} bytes). It must begin on a *paragraph* boundary (a paragraph is a contiguous block of 16 bytes, the first of which has an address divisible by 16—that is, its address must have zeros for its four least significant bits). This allows the processor to use just 16 bits (leaving off the four zero bits) to access the first byte of a given segment.

A.2

To access bytes further into a segment, instructions use not only the 16-bit segment pointer but also a 16-bit offset. Combined, the segment pointer and offset can specify any byte in memory. (See Figure A-2 for an illustration of accessing a segment.)

Segments may be any size up to 64K. There is no way to specify the exact size of a segment; the processor does not police memory access. Segments may obviously overlap; it is possible, for example, to have two different segments begin 16 bytes apart. Since segments can be any size up to 64K, the two in this example may well overlap.

The addressing mode described above is used in the 8088 and 8086, a large portion of the GEOS market. However, later processors such as the 80386 also employ a *protected mode* of memory access, in which each process

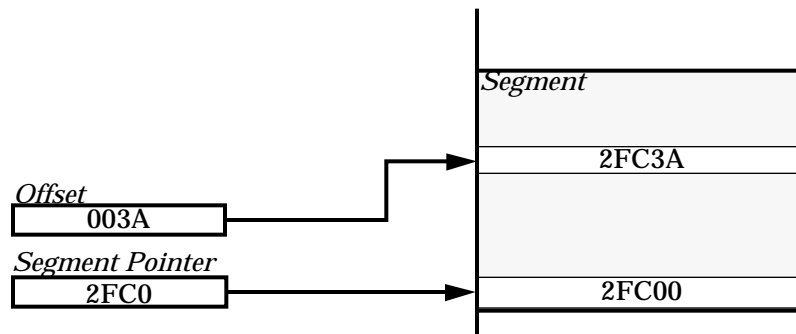
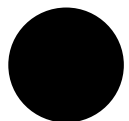


Figure A-2 Accessing a Byte in a Segment

Each byte is accessed via a segment pointer and an offset. Note that all addresses shown are in hexadecimal.



running is relegated a given portion of memory and memory access is policed by the processor. These later processors also can use the segmented mode of the 8086; however, because GEOS applications should be able to run on an 8086 or 8088 machine, they should adhere to the 8086 rules.

A.2.2 Registers

A.2

The Intel 80x86 processors have thirteen registers plus one register containing machine status flags. The thirteen registers are separated into four logical groups by their use (they are all 16 bits):

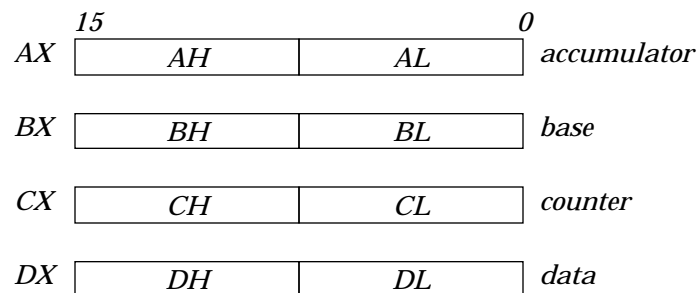
- ◆ **Instruction Pointer**
This single register maintains the address of the current instruction being executed. This is not accessed by applications.
- ◆ **Segment Registers**
These four registers contain segment pointers.
- ◆ **Index and Pointer Registers**
These four registers contain offsets into segments.
- ◆ **General registers**
These four registers can contain any general data. They may be operated on as words or bytes.

The Segment and Index registers are used in conjunction to access memory. A program may have four segments pointed to at once: A code segment (CS), a data segment (DS), a stack segment (SS), and an extra segment (ES). These segments have various uses and purposes described in most 8088/8086 books.

The four Index registers are used as offsets into the segments pointed to in the Segment registers. They are the Stack Pointer (SP), the Base Pointer (BP), the Source Index (SI), and the Destination Index (DI). Each of these index registers has special applications with certain instructions.

The General registers are four 16-bit registers that may be used for any purpose. However, due to the early restrictions of the 8080 that carried over into the later processors, some instructions place their results or take their source data from specific registers.

All four general registers may be accessed as a word or as two separate bytes. The four registers are AX (the accumulator), BX (the base register), CX (the counter), and DX (the data register). The high byte of any of these may be accessed by substituting “H” for “X,” and the low byte may be accessed by substituting “L” for “X.” (The “H” stands for “high” and the “L” for “low.”) Figure A-3 shows a diagram of these four registers.



A.2

Figure A-3 The Four General Registers

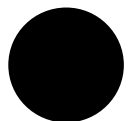
The four general registers of the 8088 and 8086 can be accessed either as entire words or as separate bytes.

A.2.3 The Prefetch Queue

Programmers who code in assembly language should be familiar with how the 8086 fetches data and instructions from memory. Good programmers can take advantage of the more efficient instructions to cut down on processor time for given operations.

The 8086 takes four clock cycles to fetch a single word from memory. To speed up instruction processing, the 8086 has a *prefetch queue*, a buffer of six bytes into which pending instructions are put. The 8086 is also broken into two separate processing units: The *Execution Unit* executes instructions while the *Bus Interface Unit* (BIU) fetches pending instructions and stuffs them into the prefetch queue.

The main goal of this separation is to make as much use of the bus as possible, even when an instruction that does not access memory is being executed. For example, if an instruction takes eight cycles to execute and



does not access memory, the BIU could meanwhile fill four instructions into the prefetch queue. Therefore, while slow instructions are still slow, the instructions after them appear to be quicker.

However, jump and branch instructions negate this prefetch effect. When a branch or jump is executed, the prefetch queue is flushed and must again be filled.

A.2

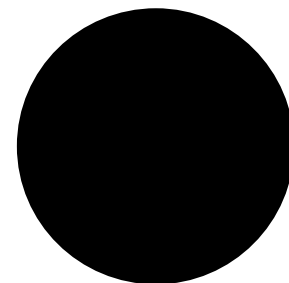
A.2.4 Inherent Optimizations

The 8088 and 8086 instruction set was designed with many instructions manifested in two different forms: one that allowed a wide range of possible arguments, and another that worked with one of the arguments prespecified. Since the second form did not have to load both arguments from memory, those instructions were shorter than their counterparts.

For example, the instruction **and** has these two forms: “And with anything,” which assembles into three bytes of machine code, and “And with AX or AL,” which assembles into only two bytes of machine code. Thus, the instruction **and al, ffh** is more efficient than **and bl, ffh**.

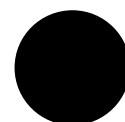
Another optimization occurs in the set of string instructions. A string is simply a set of consecutive bytes or words in memory. To repeat an operation on each element of the string could normally take a long time due to branching and checking for final conditions (branching clears out the prefetch queue). However, the string operations act as single, nonbranching instructions, so the prefetch queue is not affected by them. This speeds up string operations considerably.

Threads and Semaphores



B

B.1	Multitasking Goals.....	913
B.2	Two Models of Multitasking.....	914
B.2.1	Cooperative Multitasking	914
B.2.2	Preemptive Multitasking	915
B.3	GEOS Multitasking.....	916
B.3.1	GEOS Threads	917
B.3.1.1	Keeping Track of Threads	917
B.3.1.2	Event-Driven and Procedural Threads.....	917
B.3.2	Context Switches	918
B.3.3	Thread Scheduling.....	919
B.3.3.1	Base Priority	919
B.3.3.2	Recent CPU Usage.....	919
B.3.3.3	Current Priority	920
B.3.4	Applications and Threads	920
B.4	Using Multiple Threads	921
B.4.1	How GEOS Threads Are Created	921
B.4.1.1	The Application's Primary Thread.....	921
B.4.1.2	Event-Driven Threads	922
B.4.1.3	Threads That Run Procedural Code	922
B.4.2	Managing Priority Values	923
B.4.3	Handling Errors in a Thread	924
B.4.4	When a Thread Is Finished.....	925
B.5	Synchronizing Threads	926
B.5.1	Semaphores: The Concept.....	926
B.5.1.1	Initialize	926
B.5.1.2	Set (the "P" Operation)	926
B.5.1.3	Reset (the "V" Operation)	927
B.5.1.4	The Dreaded Deadlock Problem.....	927
B.5.2	Semaphores In GEOS.....	928



B.5.2.1	Operations on a Semaphore	929
B.5.2.2	Operations on a Thread Lock.....	930



One of the most impressive features of GEOS is its ability to perform several tasks simultaneously, even on the least powerful PCs. Of course, the PC has only one processor and can execute only one instruction at a time. GEOS keeps track of the various tasks, or *threads*, that are underway, and by switching from one thread to another many times per second creates the illusion that the PC is actually doing all the jobs at the same time.

B.1

This chapter covers

- ◆ the basic concepts of multitasking,
- ◆ the GEOS multitasking scheme,
- ◆ the steps to creating a multi-threaded application, and
- ◆ the use of semaphores to synchronize threads and avoid deadlock.

You will need to know the information in this appendix if you will be writing a multi-threaded application. The information in the appendix is not essential if your application will be single-threaded or if you will use the standard GEOS dual-thread architecture. For a dual-thread application, be careful not to send a message with **@call** from an object run by a user interface thread to an object run by any other thread of the application.

B.1 Multitasking Goals

The GEOS multitasking system is one of the most sophisticated available for PCs today. It was designed with the latest available technology and was created to serve the following two primary goals:

- ◆ **Fast Response**
The GEOS multitasking system was designed with a strong emphasis on rapid response to user input. Prompt, visible reaction to user action is the single most important factor contributing to the perception of speed. For example, if a user changes an element in a spreadsheet (requiring the whole spreadsheet to be recomputed) and then pulls

down a menu, he wants the menu to appear right away. If the menu does not appear until the computation is finished, the system will seem sluggish. If the computation takes a few seconds or longer, the user may wonder whether the system has crashed altogether.

◆ **Ease of Programming**

Making the programmer's job easy is another motive behind the design of GEOS multitasking. Ideally, the programmer should not need to be aware that his program will run in a multitasking environment. The program should proceed as though it were the only one running on the system. Besides certain "good citizen" rules, application programs are isolated from the multitasking environment. On the other hand, if a programmer wants to take advantage of the multitasking capability of GEOS (by designing a program to perform more than one task concurrently) the system is designed to make this as simple and efficient as possible.

B.2

B.2 Two Models of Multitasking

Many operating systems provide the ability to carry out multiple tasks at the same time. While each operating system has its own unique way of managing this, there are two fundamental types of multitasking: cooperative (used by Microsoft Windows), and preemptive (used by GEOS).

B.2.1 Cooperative Multitasking

A cooperative multitasking system, as the name suggests, is one in which the various programs cooperate. They agree to share the system and its resources. Each program running under a system has complete control while it is actually running. Every so often, when it reaches a convenient place, it calls a special system routine (called a context-switch routine) to see if any other program has work to do. If so, that program takes control of the system until it in turn reaches a convenient stopping point and passes control on to the next waiting program. If several programs are ready to run, they wait in a queue so that each one gets a chance to run before the first program runs again. (It is also possible to implement a cooperative multitasking system where some programs have a higher

priority than others. In this case, a more complicated algorithm might be used by the context-switch routine to determine which program gets to run next.)

Smooth operation of a cooperative multitasking system requires that all programs be written to call the context-switch routine frequently. When large calculations are being performed, programmers tend to find this requirement inconvenient. Writing well-behaved programs (i.e., programs that do not keep control of the processor for too long at a stretch) is especially difficult because most cooperative multitasking systems impose restrictions on when a context switch can take place.

B.2

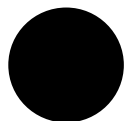
B.2.2 Preemptive Multitasking

In a preemptive multitasking system, programs do not have to relinquish control of the system voluntarily. Instead of calling a context-switch routine, the program is written as though it were going to run continuously from start to finish. The hardware generates a timer interrupt a number of times each second, and that interrupt triggers the kernel's context-switch mechanism.

The context switch can also be triggered by other interrupts. For example, if the user moves the mouse in GEOS, the mouse will generate an interrupt. GEOS responds by marking the input thread runnable; the thread will then run after the interrupt is complete. This is how GEOS achieves its extraordinary response times to user input.

With preemptive multitasking, each program can have the illusion that it is running continuously and has complete control of the system. It also enables the system to interact quickly with the user even when applications are busily computing new results.

For example, a spreadsheet program can keep running until the timer interrupt causes a context switch. Other programs, including the one responsible for drawing menus, then get their turns to run. If a user clicks on a pull-down menu, the menu will appear. When the spreadsheet program regains control of the system, it can carry on from where it was interrupted, blissfully unaware that any of this has taken place.



While preemptive multitasking makes most things simpler for the user and application programmer, there are a few important issues to consider in writing programs for a preemptive multitasking system such as GEOS. When the context switches are controlled by a timer interrupt, they can occur between any two instructions. If a program is interrupted while it is updating a data structure, that data structure may be left in an inconsistent state while another thread is running. If the data structure is not accessed by any other process running on the system, there is no problem: the update will be completed when the program resumes. However, some data structures (including system resources) may be accessed by more than one program. It is important that two updates to the same data do not happen at the same time.

This problem is analogous to one often experienced by network users. If a text file is being edited at the same time by two different users and they both save their changes to the file, whoever saves first will have his version overwritten by the other. Many systems have a means of locking a file while you are editing it; no one else can begin editing the file while you have it locked. A preemptive multitasking system must have a similar locking scheme to prevent two accesses to the same data structure from happening at the same time. The locking mechanism should be as transparent as possible to the programmer. For example, the locking and unlocking of system resources should happen automatically so that application programmers need not concern themselves with it.

This is exactly how GEOS coordinates its resources, as you shall see in the following sections.

B.3 GEOS Multitasking

GEOS implements a preemptive multitasking scheme. Application programs are required to follow certain “good citizen” rules and are otherwise given the illusion that they are alone on the system. The applications themselves are, for the most part, isolated from the multitasking environment.

B.3.1 GEOS Threads

The various units that take turns running in the system are called “threads” in GEOS. Threads can have different priorities; a thread that has a higher priority (indicated by a lower priority number) will generally get more processor time.

B.3.1.1 Keeping Track of Threads

B.3

In order to switch among threads, the system needs to keep track of certain things about each one. For each thread, the system keeps track of priority, the most recent values of the registers, and flags. The priorities are used to determine which thread is going to run next. When the thread is run, the appropriate registers are reset to the values they had the last time the thread was stopped. This allows the thread to resume execution as though it had never been interrupted.

Like other things in GEOS, each thread has a handle, a sixteen-bit value which programs use to refer to the thread. When calling thread-related routines (e.g., to set the thread’s priority), programs use the handle to specify the thread.

B.3.1.2 Event-Driven and Procedural Threads

GEOS uses two different types of threads. The two types differ only in the way they run when their turn comes and in the way they are created. The discussions about priority, context switches, and synchronization elsewhere in this chapter apply equally to both types.

The first type of GEOS thread is an “event-driven” thread. An event-driven thread normally executes code for one or more objects. Each event-driven thread has an event queue; when a message is sent to any object created by the thread, the message is placed in the thread’s event queue. The thread processes each event in the order received by executing the appropriate message handler from the object’s class definition.

Messages can also be sent to the thread itself rather than to an object created by the thread. When the thread is created, it is assigned a class—normally a subclass of **ProcessClass** (for non-application threads)

or **GenProcessClass** (for application threads)—to determine the handlers to use when messages are sent directly to the thread. In this sense, the thread can be considered an instance of the given class.

The second type of thread is procedural. Rather than running handlers for messages in an object-oriented scheme, it simply executes procedural code. The system does not provide an event queue for a procedural thread, and messages cannot be sent to such a thread.

B.3

B.3.2 Context Switches

Context switches are triggered in two ways under GEOS. The first is a timer or other hardware interrupt. The second occurs when the thread reaches a point where it cannot continue right away, such as when the thread exits or when it attempts to access a locked resource.

The PC hardware generates a timer interrupt sixty times per second. The time between timer interrupts is called a *tick*. Each thread is allowed to run for a specified number of ticks before the timer interrupt routine will transfer control to a different thread. This number of ticks is the same for all threads; it is called a *time slice*.

When a thread begins its turn, GEOS sets a counter to the number of ticks in a time slice. At each timer interrupt, GEOS decrements the counter. If the counter has not reached zero, control is immediately returned to the running thread. When the counter reaches zero, GEOS checks to see if some other thread has reached a higher priority than the current thread. If so, the current thread is placed in the system's list of runnable threads (called the *run queue*), and the highest priority thread begins running. Otherwise the current thread gets to run for another time slice.

Sometimes a thread will try to access a system resource (or shared data object) which is currently in use by another thread. When this happens, the thread must wait until the desired resource is available. The thread is placed on a queue, and a new thread is selected from the run queue. Every thread that is not currently running is either in the run queue (waiting to be executed) or in another queue waiting for a needed resource to become available. When the resource becomes available, the thread is moved to the

run queue and is ready to be run again. This process is described in greater detail in section ● B.5 on page B-926.

B.3.3 Thread Scheduling

When there is a context switch, GEOS must determine which thread will be executed next. It does this by examining the run queue and selecting the most important thread to run. A thread's importance is reflected in its priority: the lower the priority number, the more important the thread.

B.3

B.3.3.1 Base Priority

Each thread gets a base priority, a value from zero to 255. Applications generally have a priority between 128 and 191. Threads that are critical to quick system response (such as user input threads that manage such things as pull-down menus and dialog boxes) are given lower numbers. Higher numbers can be given to less time-critical threads such as those used for print spooling and other background activities. To provide faster response to the user, GEOS temporarily reduces an application's base priority by 32 (giving it a higher priority) when the user is interacting with it. When the user switches to interact with another application, the first application's base priority returns to normal, and the new one gets a reduced base priority number.

B.3.3.2 Recent CPU Usage

GEOS keeps track of a thread's recent CPU usage with a number that varies from zero to 60. Starting at zero, the number is incremented at every timer interrupt while the thread is running. Once each second, the recent CPU usage is halved, so that as a thread's CPU usage recedes into the past, its recent CPU usage number will diminish. The resulting number reflects how much time the thread has had control of the CPU and how long ago this time was.

B.3.3.3 Current Priority

Once each second, the current priority of each thread is recomputed by adding the base priority to the recent CPU usage. The resulting number is the one used in selecting a thread from the run queue.

When it is time for a context switch, GEOS selects the thread with the lowest current priority number from the run queue. If there is a tie, the selection is arbitrary. However, because recent CPU usage counts against a thread, two threads of equal priority will not stay that way. One will run, and its recent CPU usage (and thus its current priority number) will be increased. The other thread will therefore get its chance to run.

B.3

B.3.4 Applications and Threads

There are two standard architectures for GEOS applications: single-thread and dual-thread. While the single-thread option is somewhat easier to program, there are distinct advantages to the dual-thread method.

In the dual-thread architecture, one thread manages the application's user interface while the other manages the rest of the application's functionality. Since both threads are event-driven, each has an event queue. Messages that are sent to user interface objects (resulting from mouse clicks, keyboard input, etc.) can be handled without waiting for other tasks in the application to be completed. This allows the application to respond to user input (by putting up menus, moving windows, and so on) without first completing the current non-user-interface task (which may involve a lot of computation).

The dual-thread architecture, however, poses a problem of synchronization: One thread can get ahead of the other. Threads that count on each other must keep track of each other's progress in order to avoid this; when potential problems are identified, use semaphores to keep the threads in line (see section ● B.5 on page B-926).

B.4 Using Multiple Threads

It is possible for an application to create additional threads for a variety of purposes. For example, a terminal emulation program might have a thread whose sole purpose is to monitor the serial line for incoming characters. This might avert the danger of a serial input buffer or stream overflowing while the application is performing an involved task, such as loading a text file from disk, while not requiring a great deal of fixed memory for the serial driver's input buffer.

B.4

B.4.1 How GEOS Threads Are Created

GEOS threads can be created in three different ways. The first thread (or pair of threads, in the dual-thread architecture) for each application is created automatically when the application is launched. By calling the appropriate routines, the application can create additional threads to handle messages sent to certain objects or to run procedural code.

B.4.1.1 The Application's Primary Thread

The application's primary thread is created automatically by GEOS when the application is launched. (See "Applications and Geodes," Chapter 6 for information on launching applications.) For example, if a user double-clicks on your application's icon in a GeoManager window, GeoManager calls the library routine **UserLoadApplication()**, specifying the geode file and certain other parameters. This calls the **GeodeLoad()** routine in the GEOS kernel.

If the program is written using the single-thread model, GEOS creates an event-driven thread to handle messages sent to any object in the program. If the program is written using the dual-thread model, GEOS creates one event-driven thread to handle messages sent to the program's user interface objects and another to handle messages sent to other objects in the program.

If the program requires more than two threads, the extra thread(s) must be allocated manually on startup and destroyed before the application exits completely.

B.4.1.2 Event-Driven Threads

`ThreadAttachToQueue()`

B.4

To create an event-driven thread (one that handles messages sent to certain objects), send a `MSG_PROCESS_CREATE_EVENT_THREAD` to your application's primary thread, passing as arguments the object class for the new thread (a subclass of **ProcessClass**) and the stack size for the new thread (1 K bytes is usually a good value, or around 3 K bytes for threads that will handle keyboard navigation or manage a text object). This message is detailed in "System Classes," Chapter 1 of the Object Reference Book.

GEOS will create the new thread, give it an event queue, and send it a `MSG_META_ATTACH`. Initially, the thread will handle only messages sent to the thread itself. If the thread creates any new objects, however, it will handle messages sent to those objects as well. To control the behavior of the new thread, define a subclass of **ProcessClass** and a new handler for `MSG_META_ATTACH`. The new handler can create objects or perform whatever task is needed. Be sure to start your new handler with `@callsuper()` so that the predefined initializations are done as well.

If you have a thread that you want attached to a different event queue, you can use **ThreadAttachToQueue()**. This routine is not widely used except when applications are shutting down and objects need to continue handling messages while not returning anything. It's unlikely you will ever use this routine.

B.4.1.3 Threads That Run Procedural Code

`ThreadCreate()`

To create a thread to run procedural code, first load the initial function into fixed memory. Then call the system routine **ThreadCreate()**, passing the following arguments: The base priority for the new thread, an optional sixteen-bit argument to pass to the new thread, the entry point for the code,

the amount of stack space GEOS should allocate for the new thread, and the owner of the new thread.

B.4.2 Managing Priority Values

`ThreadGetInfo()`, `ThreadModify()`, `ThreadGetInfoType`

You can ascertain and modify the priority of any thread in the system, given the thread's handle. The handle is provided by the routines that create threads, and it can be provided by one thread to another in a message. The following system routines relate to the priority of a thread:

B.4

ThreadGetInfo() returns information about a thread. When calling **ThreadGetInfo()**, pass the handle of the thread in question and a value of the type **ThreadGetInfoType** (see below). If zero is passed as the thread handle, **ThreadGetInfo()** returns information on whatever thread executed the call.

ThreadGetInfoType is an enumerated type with three possible values:

TGIT_PRIORITY_AND_USAGE

This requests the base priority and recent CPU usage of a thread. (To determine the current priority, simply add the base priority to the recent CPU usage.)

TGIT_THREAD_HANDLE

This requests the handle of the thread. Use this (with a thread handle of zero) to get the caller's own thread handle.

TGIT_QUEUE_HANDLE

This requests the handle of the event queue for an event-driven thread. It returns a zero handle if the thread is not event-driven.



It is rarely a good idea to lower a thread's base priority number.

ThreadModify() changes the priority of a thread. The arguments to pass include the handle of the thread to modify (zero for the thread executing the call), a new base priority for the thread, and two flags: One that indicates whether to change the thread's base priority and one that indicates whether to reset the thread's recent CPU usage to zero. If the flag to change the thread's base priority is not set, the new base priority argument is ignored. In general, you should only lower a thread's priority (i.e., raise its base priority number). Applications that raise their own



priority damage the performance of the system as a whole. Keep in mind that GEOS already favors the thread with which the user is interacting.

There are several pre-defined priority levels you can use to set a thread's priority. You may wish to use these when debugging to raise the priority of a potentially buggy thread for efficient debugging. These are listed below, each a different constant.

B.4

- ◆ `PRIORITY_TIME_CRITICAL`
Threads should not be set time-critical unless they must own the processor exclusively for a certain amount of time. Excluding other threads can have undesirable side effects.
- ◆ `PRIORITY_HIGH`
- ◆ `PRIORITY_UI`
- ◆ `PRIORITY_FOCUS`
- ◆ `PRIORITY_STANDARD`
- ◆ `PRIORITY_LOW`
- ◆ `PRIORITY_LOWEST`

B.4.3 Handling Errors in a Thread

`ThreadHandleException()`, `ThreadException`

Some threads in GEOS will want to handle certain errors in special ways. The errors a particular thread can intercept and handle are listed in an enumerated type called **ThreadException**, the elements of which are shown below:

- ◆ `TE_DIVIDE_BY_ZERO`
- ◆ `TE_OVERFLOW`
- ◆ `TE_BOUND`
- ◆ `TE_FPU_EXCEPTION`
- ◆ `TE_SINGLE_STEP`
- ◆ `TE_BREAKPOINT`

A thread can handle a particular exception by setting up a special handler routine and calling **ThreadHandleException()** when one of these exceptions occurs. This is useful if a number of objects are run by the same thread and all should handle a particular exception in the same way; the routine can be thread-specific rather than object-specific.

ThreadHandleException() must be passed the thread's handle, the exception type, and a pointer to the handler routine's entry point.

B.4

B.4.4 When a Thread Is Finished

`ThreadDestroy()`

Whenever an application creates an additional thread with `MSG_PROCESS_CREATE_EVENT_THREAD` or **ThreadCreate()**, it must be sure that the thread exits when it is finished. Simply exiting the application may not eliminate any additional threads, and these threads can cause GEOS to hang when shutting down the system.

When a thread exits, it should first release any semaphores or thread locks it has locked and free any memory or other resources that are no longer needed. Resources in memory do not have to be freed in the same thread that allocated them, but you should be sure that they are freed before the application exits.

A procedural thread exits by calling **ThreadDestroy()** with two arguments: an error code and an `optr`. When the thread exits, it sends (as its last act) a `MSG_PROCESS_NOTIFY_THREAD_EXIT` to the application's primary thread and a `MSG_META_ACK` to the object descriptor passed. Each message has the error code as an argument. In designing a multi-threaded application, you can create methods for `MSG_PROCESS_NOTIFY_THREAD_EXIT` (in your primary thread's class) or `MSG_META_ACK` (in any class) for communication among threads, and you may use the error code for any data you choose. The convention is that an error code of zero represents successful completion of a thread's task.

An event-driven thread should not call **ThreadDestroy()** directly because its event queue must be removed from the system cleanly. Instead, send a `MSG_META_DETACH` to the thread, passing the same arguments as for **ThreadDestroy()**. The handler for `MSG_META_DETACH` in **MetaClass**

cleanly removes the event queue and terminates the thread, sending the same messages as described above. You may write a special handler for `MSG_META_DETACH` when you subclass **ProcessClass**, but be sure to end the handler with `@callsuper()` so the thread exits properly.

B.5 Synchronizing Threads

B.5

Because GEOS is a preemptive multitasking environment, it needs a way to prevent two threads from accessing system resources (or other shared data) simultaneously. Otherwise, data could be corrupted. GEOS uses semaphores to prevent threads from performing conflicting operations at the same time.

B.5.1 Semaphores: The Concept

A semaphore is a data structure on which three basic operations are performed. These operations allow threads to avoid conflicting with other threads. Think of a semaphore as a flag which programs can set to indicate that some resource is locked. Anyone else who wants to use the resource must wait in line until whoever set the flag resets it. The three basic operations on a semaphore are initialize, set, and reset.

B.5.1.1 Initialize

The “initialize” operation creates a semaphore and gives it a name. In its initial state, the semaphore is “unlocked,” meaning the first process that attempts to access it will succeed right away. A semaphore must be initialized before it can be used, although the initialization can be handled by the operating system so that it is transparent to the programmer.

B.5.1.2 Set (the “P” Operation)

The “P” operation is what a program performs in order to make sure it is allowed to proceed. For example, if the program is about to access shared

data, it performs the “P” operation on the semaphore protecting that data to make sure no other program is accessing it.

If the semaphore is unlocked and a thread performs the “P” operation on it, the thread simply marks the semaphore locked and proceeds normally. If the semaphore is locked, a thread performing the “P” operation will *block*. This means the thread will stop running and will wait in the thread queue associated with the semaphore. When its turn to perform the protected operation arrives, the thread will proceed.

B.5

B.5.1.3 Reset (the “V” Operation)

When a thread has finished a protected operation, it performs the “V” operation to unlock the semaphore. If there are other threads in the queue for this semaphore, one of them is restarted and takes over, keeping the semaphore locked. Thus only one thread at a time runs the protected operation. If a thread performs the “V” operation on a semaphore and there are no other threads waiting in the semaphore’s queue, the thread simply marks the semaphore unlocked and proceeds.

Programs must always reset a semaphore when they are done with it. If you fail to reset a semaphore, other threads may wait forever.

Because only one thread at a time is performing the protected operation and this thread is responsible for unlocking the semaphore, it is sometimes said to “have” the semaphore. The “P” and “V” operations are often referred to as “grabbing” and “releasing” a semaphore, respectively.

B.5.1.4 The Dreaded Deadlock Problem

When semaphores are not used carefully, they can cause programs to stop running entirely. Suppose Thread A tries to grab a semaphore which Thread B has locked. Thread A stops running until Thread B releases the semaphore. Thread B then tries to grab a second semaphore, which Thread A has previously locked. Thread B waits for Thread A to release the second semaphore, but Thread A is waiting for Thread B to release the first semaphore. Neither thread will ever wake up. This situation is called “deadlock.”



Avoid deadlock!

Be very careful when using multiple semaphores.

B.5

To avoid deadlock, follow these guidelines:

- ◆ When possible, avoid having a thread attempt to lock one semaphore while it already has another one locked.
- ◆ When two or more semaphores may be locked by the same thread at the same time, they should always be used in the same order. Semaphores are often arranged in a hierarchy, with the coarsest (the one controlling access to the most resources) at the top and the finest at the bottom. Any thread grabbing multiple semaphores in this hierarchy must always grab from top to bottom; that is, no thread should grab a semaphore “above” one it already has locked.
- ◆ In certain situations, a semaphore is used “between” two threads. One thread needs to wait until another performs a specific action. The first thread is said to “block” on the second. Of course, two threads must never block on each other. To ensure this situation never arises, only one of the threads should use the **@call** keyword when sending messages to the other; the other should always use **@send** and, when getting return information, have some sort of notification message sent in response.
- ◆ When using the GEOS messaging system to send a message with **@call** to an object in another thread, the sending thread automatically blocks on the receiving thread. Since a number of user interface objects must be sent messages with **@call**, the application thread sometimes blocks on the user interface thread. To avoid deadlock, code that runs in the user interface thread must never send messages with **@call** to objects in the application thread. (This is a particular example of the above rule being implemented.)

B.5.2 Semaphores In GEOS

GEOS contains several system resources that are protected by semaphores. Since application programs can access these resources only through library routines, the programmer does not need to be aware of these semaphores; the required operations are performed by the library routines. For system resources (e.g. files, memory, handles), GEOS defines semaphores and provides special routines to set and reset them. The chapter that describes each system resource explains how to use the special semaphores that protect the resource.

The routines described in this section illustrate GEOS semaphores and can be used to create semaphores to protect resources defined within a multithreaded application. There are routines for each of the operations (initialization, P, and V), and there are special routines which simplify the use of a semaphore by multiple objects within the same thread.

Note that it is possible to create a semaphore with a starting value greater than one. That is, you can create a semaphore that will allow more than one thread to grab it at once. Typically, if only one thread may grab the semaphore, the thread is called a “mutual exclusion,” or “mutex,” semaphore, because it is normally used to make sure two threads don’t mutually grab a particular resource.

B.5

Semaphores that can be grabbed by more than one thread are generally called “scheduling semaphores” because they allow easy manipulation of scheduled resources. The classic example of this is the “producer–consumer problem” wherein one thread produces buffers and another consumes them. Initially, no buffers exist, so the semaphore starts at zero. The consumer goes into a loop wherein it simply P’s the semaphore (blocking until a buffer exists), takes the first buffer in the queue, processes the buffer, destroys the buffer, and then returns to the top of the loop. The producer, meanwhile, can produce any number of buffers, queue them, and V the semaphore once for each consumable buffer. The consumer thread will continue to process until all the buffers are consumed, and then it will block and wait for more buffers.

B.5.2.1 Operations on a Semaphore

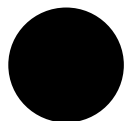
Multiple-Use Semaphores

By specifying an initial value of two or more, you can set the number of threads that can access a semaphore at the same time. This is often used to track multiple instances of a particular resource.

```
ThreadAllocSem(), ThreadPSem(), ThreadPTimedSem(),
ThreadVSem(), ThreadFreeSem()
```

To create a semaphore, simply call the routine **ThreadAllocSem()**, passing an initial value for the semaphore. This should normally be one to indicate the semaphore is unlocked. If you want the semaphore to be locked initially, pass an initial value of zero. In either case, the returned value will be the handle of the newly created semaphore. Use this handle with the routines described below.

Once a semaphore is created, a thread can lock it (i.e., perform the “P” operation) by calling the routine **ThreadPSem()**, passing the semaphore’s



handle as an argument. If the semaphore is unlocked, the thread locks it and proceeds; otherwise the thread waits in the semaphore's thread queue.

Another routine that performs the “P” operation is **ThreadPTimedSem()**. When calling this routine, pass as arguments the semaphore's handle and an integer representing a number of ticks. This integer is a time limit: If another thread has the semaphore locked and does not unlock it within the specified number of ticks, the routine will return with a flag indicating the lock was unsuccessful. Programs that use **ThreadPTimedSem()** must check this flag and must not perform the protected operation if it is set. The most common use of this routine is with a time limit of zero, meaning that the semaphore should be locked only if it is available right away—if it is not available, the thread will continue with some other action.

To release the semaphore (by performing the “V” operation), the thread calls **ThreadVSem()**, again passing the semaphore's handle. If there are other threads waiting for the semaphore, the one with the lowest current priority number takes over.

When a semaphore is no longer needed, it can be destroyed by calling **ThreadFreeSem()** with the semaphore's handle as an argument.

B.5.2.2 Operations on a Thread Lock

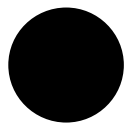
```
ThreadAllocThreadLock(), ThreadGrabThreadLock(),  
ThreadReleaseThreadLock(), ThreadFreeThreadLock()
```

At times it is convenient to have a program lock a semaphore that it has already locked. For example, one routine might lock a semaphore protecting a piece of memory and then call itself recursively. A thread lock is a semaphore that can be locked any number of times, as long as each lock is performed by the same thread. If another thread tries to grab the thread lock, it will wait until the first thread has performed the “V” operation once for each time it has run the “P” operation. It is possible to write reentrant routines using thread locks but not using regular semaphores.

A thread lock is initialized with the **ThreadAllocThreadLock()** routine, which takes no arguments. A thread lock is always created unlocked. To perform the “P” and “V” operation on a thread lock, use **ThreadGrabThreadLock()** and **ThreadReleaseThreadLock()**, respectively, and pass the semaphore's handle as an argument. These

routines are analogous to **ThreadPSem()** and **ThreadVSem()** for semaphores. When a thread lock is no longer needed, it should be freed with a call to **ThreadFreeThreadLock()**.

B.5

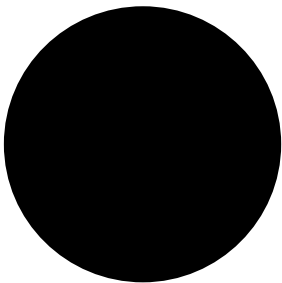


Threads and Semaphores

932

B.5

Libraries



C.1	Design Philosophy	935
C.2	Library Basics	936
C.3	The Library Entry Point	937
C.4	Exported Routines and Classes	940
C.5	Header Files	941
C.6	Compiler Directives	941





If you write more than one application, you may find yourself repeating a lot of effort. You may end up writing the same routines several times, or defining very similar classes for different applications.

For this reason, GEOS lets you define *libraries*. Libraries are much like applications. However, they don't do anything on their own; instead, they must be loaded by other geodes. They provide applications with routines and object classes. This lets applications share code easily and efficiently.

c.1



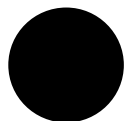
Before you read this chapter, you should be familiar with programming in GEOS; you should have successfully written a simple GEOS application. You should also be very familiar with the material in "First Steps: Hello World," Chapter 4 and "GEOS Programming," Chapter 5.

c.1 Design Philosophy

GEOS libraries are designed to make code-sharing simple and efficient. They allow several different applications to make use of the same routines and classes while using the minimum amount of space.

Conventional libraries are fully included in each application. The usual technique is to write a header file which contains the code for the library's routines. Any application which needs to use the routines can then include this library. This has one main advantage: The code can be written and tested once, and applications can then rely on it to work. However, there is a severe drawback to this approach: every application which uses the library will contain identical code. This is not a problem for non-multitasking environments; if only a single application can run at a time, then there will be only a single copy of each library. In a multithreaded environment like GEOS, however, this is a very inefficient use of resources.

GEOS libraries solve this problem. Each geode specifies which libraries it will use, either at compile-time (in the **.gp** file) or at run-time (with **GeodeUseLibrary()**). The kernel will see to it that the library is loaded



when necessary. This means that if a dozen applications are all using the same library, the code needs only be loaded once.

Conventional libraries contain only routines. GEOS libraries, on the other hand, may contain both routines and object classes. There are several advantages to defining a class in a library instead of in an application. First, there is the same code-sharing benefit that routines have. If a class is defined in a library, the heap will contain at most one copy of each of the class's methods, no matter how many applications use objects from that class. There is another advantage as well; all applications which use that class can be sure that they are using objects whose definitions are identical. this makes it possible for applications to send messages to objects owned by other geodes.

Writing a library is very much like writing an application. There are only a few differences, which are covered in this appendix. You should already be familiar with writing applications before you try to write a library.

c.2 Library Basics

A library is a geode, much like an application geode. However, its behavior is slightly different. In particular, libraries do not have any threads of their own, unless they explicitly create them.

When a geode calls a routine which is exported from a library, the routine is run by the thread which made the call, not by the library's thread. This has several implications. First, it means that a library's response time is not dependent on the number of applications which use the library. An application which uses the library a lot will do so on its own time and may have its priority reduced accordingly. Indeed, a library with many users is likely to perform better than one with few users, since its code will be less likely to be swapped out of the global heap. Similarly, library routines will use the stack of the calling thread; this means that the same routine can be called by several different threads at once, with less danger of a synchronization problem.

Another consequence is that if a library routine allocates memory, that memory will belong to the owning geode. Thus, when the application exits,

the memory will automatically be freed; on the other hand, if the library exits before the application does, the memory block will remain. If a library wants to have the block owned by the library geode, it must set the owner explicitly.

Geodes which use a library are said to be its “clients.” A client may declare that it uses a library in its **.gp** file, or it may load the library at runtime by calling **GeodeUseLibrary()**. One library may be a client of another; in this case, when the first library is loaded, the second will be as well.

C.3

A library may have a single special routine, known as its *entry point*. The kernel calls this routine to inform the library when it is launched or freed, when it acquires a new client, or when a client is unloaded. The entry point routine is described more fully in section C.3 on page 937.

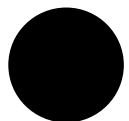
A library may export object classes or routines. If a routine is exported, it may be called by any geode which uses the library. If an object class is exported, any geode which uses the library may instantiate objects of that class, and may define a subclass of it.

Every library should have a library header file. This header file contains declarations for all exported routines and classes, as well as definitions of any appropriate macros, constants, structures, etc. Every geode which uses the library will need to include this header file. If the library exports any object classes, the header file should be a Goc header file with the suffix **.goh**; otherwise it should be a standard C header file with the suffix **.h**. The header file is described in more detail in section C.5 on page 941.

c.3 The Library Entry Point

`LibraryEntry()`, `LibraryCallType`

A library may need to do bookkeeping when it is launched, when a client is attached, or at other times. For this reason, some libraries will have an entry point routine. The entry point routine is called by the kernel; it should never be called by other geodes. Some of the calls are made in the kernel thread, while others are made by a geode’s thread. All of the calls are made automatically by the kernel.



An entry point routine must take two arguments. The format of an entry point is shown in Code Display C-1 on page ● 938.

Code Display C-1 A Library Entry Point

```
Boolean _pascal
    LibraryEntry(LibraryCallType    ty,
                 GeodeHandle        client);
```

C.3

When the kernel calls the entry point routine, it passes the following arguments:

- ◆ A member of the **LibraryCallType** enumerated type. This specifies why the kernel is calling the routine. This type is described below.
- ◆ A geode handle. This parameter is valid only if certain **LibraryCallType** values are passed.

The entry point should return *true* if an error occurs; otherwise it should return *false* (i.e. zero).

LibraryCallType contains the following members:

LCT_ATTACH

This is passed when the library has just been launched. The *client* parameter is undefined. The call is made in the kernel thread.

LCT_DETACH

This is passed when the library is about to be unloaded. The *client* parameter is undefined. The call is made in the kernel thread.

LCT_NEW_CLIENT

A thread has just called **GeodeUseLibrary()**, or a geode which depends on the library is being launched. The *client* parameter contains the **GeodeHandle** of the new client. The call is made in the kernel thread.

LCT_NEW_CLIENT_THREAD

A geode which depends on the library has just spawned a new thread. The *client* parameter contains the **GeodeHandle** of the thread's owner. The call is made in the new thread.

LCT_CLIENT_THREAD_EXIT

A thread which uses the library is being destroyed. The *client* parameter contains the **GeodeHandle** of the thread's owner. The call is made in the soon-to-be destroyed thread.

LCT_CLIENT_EXIT

A client loaded this library with **GeodeUseLibrary()** has just called **GeodeFreeLibrary()**. The *client* parameter contains the **GeodeHandle** of the former client. The call is made in the kernel thread.

C.3

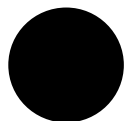
LCT_DETACH

The library is about to be unloaded. The call is made in the kernel thread.

Sometimes a single action can prompt several calls to the entry point, each with a different **LibraryCallType** value. For example, suppose FooWrite is launched. This application's **.gp** file specifies that it uses the BarObj library. At the time FooWrite is launched, BarObj has not been loaded. The kernel will automatically launch BarObj and immediately call the entry point with parameter LCT_ATTACH. The kernel will then call the entry point again with parameter LCT_NEW_CLIENT, passing FooWrite's **GeodeHandle**. It will then call the entry point once for each FooWrite thread, passing LCT_NEW_CLIENT_THREAD; it will make these calls as each thread is started.

Some libraries will not need to take any actions when the entry point is called; these libraries need not have an entry point routine. On the other hand, some libraries will need to do bookkeeping chores. This is left entirely to the library's discretion.

The entry point should take care not to perform any actions with side effects outside the library. If the entry point allocates memory, it should make sure to make the library's geode the block's owner. Similarly, the entry point should not change the working directory; instead, it should use **FilePushDir()** and **FilePopDir()** to make temporary changes to the working directory.



C.4 Exported Routines and Classes

Writing routines for a library is very much like writing them for an application. Simply export the routine in the **.gp** file and any geode which uses the library will be able to call the routine.

C.4

It is important when writing routines for export to document the routines exhaustively. Remember that the library will probably be used by other programmers; they will rely on the routines to behave exactly as specified. Exported routines should also minimize side effects; for example, it is a bad idea for a library routine to change the working directory without changing it back, unless that is the routine's main purpose.

Most libraries will have a number of routines which are not for export, but are used by routines that are exported. These are simply written normally, and are not exported in the **.gp** or declared in the header file. Remember that programmers will not see these routines; their side effects should thus be fully documented with the exported routines which call them.

Some libraries will declare classes of objects. In this case, the library should specify in the **.gp** file that it uses whichever library defines the superclass of the object. For example, if a library defines a subclass of **GenClass**, it should specify that it uses the **UI** library. It should then export the new class.

Some libraries will declare classes that are not intended to be used by clients. For example, the Impex library declares **ImportExportClass**. This class is never instantiated; it contains code and instance data that are used by its subclasses (**ImportControlClass** and **ExportControlClass**). Such "hidden" classes need not be exported. However, the classes must be fully declared in the header files so the subclasses can be defined accurately and consistently.

c.5 Header Files

Every library should have at least one header file. This file contains declarations and definitions which are needed by each of the geodes which uses the library.

If a library exports routines but does not export object classes, its header will be a standard C header file. This file should contain declarations of every exported routine. It should also contain the definitions of any macros, constants, structures, etc., which clients might use.

C.5

If a library exports object classes, its header will be a Goc header with the suffix **.goh**. In addition to routine declarations, it must contain the complete declarations of each of the exported object classes, including all the message declarations. A **.goh** file should begin with the Goc directive “**@deflib** <libname>”, and end with the directive “**@endlib**”. This ensures that the header will only be included once, even if the code tries to include it several times.

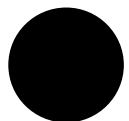
Large libraries may have several header files. For example, a library might declare several similar object classes. It is usually simpler to write a separate header for each class; a client can then include only the headers for classes which it will use. Note that the header for a class should use a Goc **@include** directive to include the header for that class's superclass.

It is of the utmost importance that the headers be kept in synchronization with the libraries they describe. As a rule, a library will include each one of those headers; that helps to keep all the files compatible. Nevertheless, you must be careful whenever changing a library.

c.6 Compiler Directives

Libraries have to be compiled slightly differently from applications. Since library routines are run under application threads, they must treat global variables differently than applications do. You must therefore add pragmas to ensure that the library is compiled correctly.

There are several steps to take:



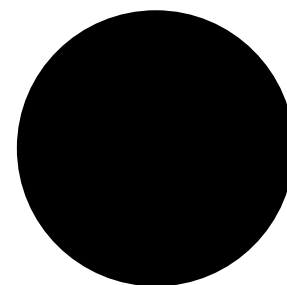
- ◆ The compiler should not expect the **ss** register to be the same as **ds**.
- ◆ The compiler must generate code to load the **dgroup** segment address into **ds** at the start of exported routines.
- ◆ The compiler must set up semaphores or other data-synchronization structures for global variables.

C.6

Most compiler manuals have a section on compiling dynamically-linked libraries (DLLs) for Microsoft Windows; this section will describe how to set up these conditions. Note that you need only do this if your library will have its own global or static variables. If the library's routines and methods use only local, automatic variables, you need not perform these actions.

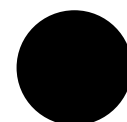
When you compile a library, you must pass the argument “-L <libname>” to Goc. Edit your **local.mk** file to make it insert this flag. For more information about the **local.mk** file, see "Using Tools," Chapter 10 of the Tools book.

The Math Library



D

D.1	Basic Math Functions	945
D.1.1	Algebraic Functions.....	947
D.1.2	Transcendental Functions	949
D.1.3	Random Number Generation.....	950
D.2	Conversions to Other Types	951
D.2.0.1	Normal Conversions	953
D.2.0.2	Date-Time Conversions	956
D.2.0.3	Using Standard Formats.....	958
D.2.0.4	Date and Time Routines.....	958
D.2.0.5	Miscellaneous Date Time Routines.....	959
D.3	Float Formats	960
D.3.1	System-defined Formats	960
D.3.2	User-defined Formats.....	963
D.4	Direct FP Operations	964
D.4.1	Floating Point Numbers.....	965
D.4.2	The Floating Point Stack	966
D.4.2.1	Initialization of the FP Stack	966
D.4.2.2	Pushing and Popping on the FP Stack	967
D.4.2.3	FP Stack Manipulation.....	968
D.4.2.4	Copying and Deleting Numbers on the FP Stack	969
D.4.2.5	Comparing Numbers on the FP Stack.....	971
D.4.2.6	Recovery of the FP Stack.....	971
D.4.3	Floating Point Math Routines	971
D.4.3.1	Constant Routines	972
D.4.3.2	Constant Operands	972
D.4.3.3	Algebraic Routines.....	972
D.4.3.4	Comparison Routines.....	974
D.4.3.5	Fractional and Integral Routines.....	974
D.4.3.6	Routines that Return Random Values.....	975
D.4.3.7	Transcendental Functions	975



D.4.3.8	Conversion Routines.....	977
---------	--------------------------	-----

The Math Library allows high precision computations not available through standard integer operations. Floating Point (FP) numbers use decimal representation to express fractional quantities. This format also allows scientific notation to represent very large and very small numbers.

To enable floating point operations, an application must include a GEOS Math Library. There are two Math Libraries: **math.h** and **math.goh**. The file **math.h** includes what you need to get floating point numbers to work within your application. The object file, **math.goh** (which includes **math.h**), includes the float-format controller, an object that allows the user to control how floating point numbers are formatted within an application.

D.1

In most cases, the mere inclusion of the Math Library will eliminate the need to directly call Math Library routines. (The C compiler will parse the mathematical expressions into appropriate floating point routines.) You may, however, wish to call some of these routines directly. Developers using Object Assembly must use this latter approach. C developers may also want to directly manipulate the FP stack for involved computations that they would rather not leave to the compiler. This approach is discussed in the latter half of this chapter.

Use of **math.goh** is optional, depending on the purpose of the application. A spreadsheet application, for example, might want to allow the user to format FP numbers. Other applications might not want to include this file.

D.1 Basic Math Functions

The Math Library includes many routines and structures that make the manipulation of FP numbers possible. Most of these are transparent to a C programmer.

C includes the following floating point types: **float**, **double**, and **long double**. A floating point number in GEOS is of type **FloatNum**, which uses the IEEE 80 bit standard (equivalent to a long double) to represent floating point values. **FloatNum** has overloaded the float, double, long double, and all basic math functions.

A **FloatNum** consists of a one bit sign, a 15 bit exponent, and a 64 bit mantissa. The maximum exponent allowed is 7FFEh. An exponent of 7FFFh (FP_NAN) signals an underflow or overflow. You may check the value of this exponent using the macro `FLOAT_EXPONENT` to extract the exponent of a **FloatNum**.

Code Display D-1 Extracting an Exponent

D.1

```
FloatNum myNum;

if (FLOAT_EXPONENT(&myNum) == FP_NAN)
{
    return(ERROR);
}
```

If you use any floats, doubles, or long doubles in your application, GEOS will convert these types into **FloatNum** values automatically and call the appropriate GEOS functions to manipulate the numbers. As this is the case, it is usually easier to declare any floating point variables in your application as **FloatNum** types (or long doubles). This cuts down on conversion time.

The Math Library provides all the familiar mathematical functions to manipulate these GEOS FP numbers (addition, multiplication, computation of logarithms, etc.). In C, most of these functions will automatically be called when their corresponding C operation involving FP numbers take place. (For example, when using the '+' binary operator to add two FP numbers, the Math Library will use the corresponding **FloatAdd()** routine.)

You may, under special circumstances, wish to use these math routines directly. If you do so, you will need to manipulate the floating point stack manually, pushing numbers on the stack and making sure numbers are in the proper stack location to perform each operation. In most cases, the FP math routines operate on numbers already in place on the FP stack; they take and return no arguments of their own.

Two functions you will need to use if you take this approach are **FloatPushNumber()** and **FloatPopNumber()**. **FloatPushNumber()** takes the address of a variable (of type **FloatNum**) to push onto the FP

stack; **FloatPopNumber()** takes the address of a buffer to place an FP number popped off the FP stack. Other routines (for example, **FloatAdd()**) can then be called to operate on the FP stack. (See Code Display D-2 for an example using this approach.)

There are many other routines that perform stack manipulation by shifting locations of FP numbers on the stack (**FloatRoll()**, **FloatDrop()**, etc.). These routines are covered in detail in the latter part of this chapter because they are seldom needed in a C applications.

D.1

D.1.1 Algebraic Functions

Algebraic routines perform algebraic operations on FP numbers. The Math Library provides all of the basic algebraic routines that operate on GEOS FP numbers (addition, subtraction, etc.). The table in Table D-1 lists the function names and the operations they perform.

If you wish to call these routines directly rather than rely on the C operations, you may manipulate the floating point stack directly. To add two numbers using **FloatAdd()** for example, you would use **FloatPushNumber()** twice to push the two values to add onto the FP stack, and then call **FloatAdd()** to operate on the FP stack. (See Code Display D-2.)

Code Display D-2 Adding Two FP Numbers

```
/*
 * The following two methods each add two FP numbers and return the result. The
 * first method is familiar C code. The second example uses the floating point
 * routines from math.h directly. Note that the C code will be assembled into code
 * that uses FloatAdd() also, but that this is transparent to the code.
 */

@interface MyProcessClass, MSG_SUM_FLOATS {
    long double      number1, number2, number3;

    number1 = 1.0;
    number2 = 2.0;
```



The Math Library

948

```
        number3 = number1 + number2;
        return(number3);
    }

    @method MyProcessClass, MSG_SUM_FLOATS_MANUALLY {
        long double          number1, number2, number3;

        number1 = 1.0;
        number2 = 2.0;

D.1    FloatPushNumber(&number1);    /* Push number1 onto the FP stack. */
        FloatPushNumber(&number2);    /* Push number2 onto the FP stack. */
        FloatAdd();                  /* Add the top two numbers on the FP stack. The
                                     * result will be placed on top of the FP stack.*/
        FloatPopNumber(&number3);    /* Pop the result into the number3 variable. */
        return(number3);
    }
```

Algebraic routines included in the Math Library are listed in Table D-1. As can be seen in the table, several functions have no equivalent C operation.

Consult “Direct FP Operations” on page 964 for more details on using these functions.

Table D-1 Basic FP Functions and corresponding C operations (if any)

Function	C Operation	Function
FloatAdd()	+	addition
FloatSub()	-	subtraction
FloatMultiply()	*	multiplication
FloatDivide()	/	division
FloatDIV()	(int) (X/Y)	division returning integer result
FloatMod()	X % Y	modulo
FloatFactorial()		factorial
FloatNegate()	-X	negation
FloatInverse()	1/X	inversion
FloatAbs()	abs(X)	absolute value
FloatMax()		maximum of two numbers
FloatMin()		minimum of two numbers
FloatLt0()	X < 0	less than 0
FloatGt0()	X > 0	greater than 0
FloatEq0()	X = 0	equal to 0
FloatFrac()		returns fractional portion of X
FloatTrunc()		truncates X to its integer value
FloatInt()	(int) X	returns integer of X, rounded down
FloatIntFrac()		separates X into integer and fraction
FloatRound()		rounds X to a given decimal places

D.1

D.1.2 Transcendental Functions

The Math Library provides an array of transcendental routines that operate on GEOS FP numbers. A transcendental function is one which cannot be derived through algebraic means. Examples of transcendental functions include the trigonometric functions (sine, cosine, etc.) and the logarithmic functions (log, natural log, etc.).

The table in Table 4-2 lists the function names and the operations they perform. Typical functions in C are listed alongside. (The basic language of C itself includes no such transcendental functions but almost all compilers

include a C math library that does.) As can be seen in the table, several functions have no equivalent C operation. Consult “Float Formats” on page 960 for more details on using these functions.

D.1.3 Random Number Generation

D.1

FloatRandomize(), FloatRandom(), FloatRandomN()

The Math Library also provides routines to create random numbers. Using any of these routines requires that you manually push and pop numbers on the FP stack.

Table 4-2 Transcendental Floating Point Functions

Function	C Equivalent	Operation
FloatSin()	sin()	sine
FloatCos()	cos()	cosine
FloatTan()	tan()	tangent
FloatArcSin()	asin()	arc-sine (inverse sine)
FloatArcCos()	acos()	arc-cosine (inverse cosine)
FloatArcTan()	atan()	arc-tangent (inverse tangent)
FloatArcTan2()		arc-tangent given two coordinates
FloatSinh()	sinh()	hyperbolic sine
FloatCosh()	cosh()	hyperbolic cosine
FloatTanh()	tanh()	hyperbolic tangent
FloatArcSinh()	asinh()	hyperbolic arc-sine
FloatArcCosh()	acosh()	hyperbolic arc-cosine
FloatArcTanh()	atanh()	hyperbolic arc-tangent
FloatExp()		e raised to the power X
FloatExponential()	exp()	X raised to the Y power
FloatLg()		logarithm to the base 2
FloatLog()	log()	logarithm to the base 10
FloatLn()		logarithm to the base e (natural log)
FloatLn1plusX()		natural log of (1 + X)
FloatSqr()		square of X
FloatSqrt()	sqrt()	square root

FloatRandomize() primes the random number generator, in preparation for a call to **FloatRandom()** or **FloatRandomN()**. If **FloatRandomize()** is passed the flag **RGIF_USE_SEED**, the routine must also pass a developer supplied seed.

FloatRandom() returns a random value between 0 (inclusive) and 1 (exclusive). The number is placed on top of the FP stack. To assign that value to a variable, use **FloatPopNumber()**.

FloatRandomN() returns a random value between 0 (inclusive) and N (exclusive), where N is an integer. The integer value must be on top of the FP stack. The returned integer is pushed onto the FP stack. To assign that value to a variable, use **FloatPopNumber()**.

D.2

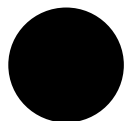
Code Display D-3 Creating a Random Number

```
/*
 * This method takes a passed seed (passedSeed) and returns a random integer
 * between 0 and 100.
 */

@method MyProcessClass, MSG_GET_RANDOM_FLOAT {
    long double    randomX;
    /* Prime the random number generator. */
    FloatRandomize(RGIF_USE_SEED, passedSeed);
    randomX = 100;
    FloatPushNumber(&randomX);
    FloatRandomN();
    FloatPopNumber(&randomX);
    return(randomX);
}
```

D.2 Conversions to Other Types

In many cases, FP numbers will need to be converted to different types for use in different parts of an application. For example, floating point numbers may be involved in a complex function that returns an integer. FP numbers may also need to appear to the user as ASCII text.



Converting Into Other Floats

There are several routines which convert GEOS FP numbers into other float formats, compatible with the C types **float** and **double**. Typically in C, this conversion is accomplished by casting the FP numbers into the other type. It is therefore done automatically for you.

If you are working in Assembly or you wish to directly pass floats or doubles to C stubs, consult “Direct FP Operations” on page 964.

Converting ASCII to FP Numbers

```
FloatAsciiToFloat()
```

FloatAsciiToFloat() converts a number represented in an ASCII text format into a FP number. The routine recognizes two flags:

- ◆ **FAF_PUSH_RESULT**
Pushes the result onto the FP stack.
- ◆ **FAF_STORE_NUMBER**
Stores the result in a specified address.

The routine must also be passed a pointer to the string to convert, the number of characters in the string to convert (starting at the address) and the buffer to store the FP number if passing **FAF_STORE_NUMBER**.

Converting FP Numbers to ASCII

```
FloatFloatToAscii(), FloatFloatToAscii_StdFormat(),  
FloatFloatIEEE64ToAscii_StdFormat()
```

FloatFloatToAscii() converts an FP number into ASCII text format. The routine must be passed a stack frame, which may be set up by declaring a local variable of type **FFA_stackFrame** and moving data into the appropriate fields. You should also pass the routine a pointer to a buffer to store the resultant string.

The **FFA_stackFrame** is a union of two structures:

FloatFloatAsciiData or **FloatFloatToDateTimeData**. You will want to use the **FloatFloatToAsciiData** structure in most cases;

FloatFloatToAsciiDateTimeData is used to format a FP number (representing a date and time) into a date-time format passed in the

structure. The routine checks a bit in the structure to see which structure is being passed.

D.2.0.1 Normal Conversions

The **FloatFloatToAsciiData** structure is used most often in formatting FP numbers into ASCII. The structure is rather large and cumbersome to set up. You may wish to use the routine **FloatFloatToAscii_StdFormat()** which sets up many of these entries automatically. (Code Display D-4 lists the entries of the **FloatFloatToAsciiData** structure.)

D.2

Code Display D-4 FloatFloatToAsciiData Structure

```
typedef struct {
/*
 * FFA_params stores the entries that the caller must set up.
 */
    FloatFloatToAsciiParams  FFA_params;

/*
 * These entries store information returned by FloatFloatToAscii() that may be
 * examined.
 */
    word                    FFA_startNumber;
    word                    FFA_decimalPoint;
    word                    FFA_endNumber;
    word                    FFA_numChars;
    word                    FFA_startExponent;

/*
 * The rest of the entries are for internal use only.
 */
    word                    FFA_bufSize;
    word                    FFA_saveDI;
    word                    FFA_numSign;
    byte                    FFA_startSigCount;
    byte                    FFA_sigCount;
    byte                    FFA_noMoreSigInfo;
    byte                    FFA_startDecCount;
    byte                    FFA_decCount;
    word                    FFA_decExponent;
    word                    FFA_curExponent;
    byte                    FFA_useCommas;
```



```
        byte          FFA_charsToComma;  
        char          FFA_commaChar;  
        char          FFA_decimalChar;  
    } FloatFloatToAsciiData;
```

FFA_params is a structure that stores the following entries of its own:

D.2

formatFlags

Flags specifying the look and feel of the ASCII text format (see below for allowed flags).

decimalOffset

Integer which specifies the number of decimal places to shift the output. For example, to display a number in terms of millions, a **decimalOffset** of -6 shifts the decimal point six places to the left; to display in terms of tenths would require a **decimalOffset** of 1.

totalDigits

Integer which specifies the maximum number of decimal places (integer and decimal portions) that the FP number may exhibit. The ASCII string is truncated if the length of the string is greater than this number.

decimalLimit

Integer which specifies the maximum number of digits to the right of the decimal point. For example a **decimalLimit** of 2 would print out 123.456789 as 123.46.

preNegative

Characters used to precede a negative number, in the format of a null terminated text string.

postNegative

Characters used to follow a negative number, in the format of a null terminated text string.

prePositive

Characters used to precede a positive number, in the format of a null terminated text string.

postPositive

Characters used to follow a positive number, in the format of a null terminated text string.

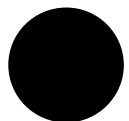
- header** Characters used to precede the number, in the format of a null terminated text string. For example, a typical header for a currency format might be “\$”.
- trailer** Characters used to follow a FP number, in the format of a null terminated text string.

The **formatFlags** record is a record of type

FloatFloatToAsciiFormatFlags and defines the format of the ASCII output. Set the appropriate flags to get the desired output.

D.2

- ◆ **FFAF_FROM_ADDR**
Set if routine should take the FP number from a specified address, rather than the FP stack. This pointer (*number) should be passed to **FloatFloatToAscii()** if this flag is set.
- ◆ **FFAF_DONT_USE_SCIENTIFIC**
Set if the result should not be expressed in scientific notation under any condition. If this is set, the number will always be formatted in fixed format.
- ◆ **FFAF_SCIENTIFIC**
Set if the result should be expressed in scientific notation even if the number can be expressed in fixed point format. For example, the number ‘2’ will be expressed as “2 x 10⁰.”
- ◆ **FFAF_PERCENT**
Set if the result should be expressed as a percentage.
- ◆ **FFAF_USE_COMMAS**
Set if the result should use commas to separate thousands.
- ◆ **FFAF_NO_TRAIL_ZEROS**
Set if extraneous zeros to the right of the decimal point should be ignored. For example, “123.67000000” will be reduced to “123.67.”
- ◆ **FFAF_NO_LEAD_ZEROS**
Set to ignore the lead zero for a number less than one. That is, “0.123” is reduced to “.123.”
- ◆ **FFAF_HEADER_PRESENT**
Set if a header is present; this speeds up conversion.
- ◆ **FFAF_TRAILER_PRESENT**
Set if a trailer is present; this speeds up conversion.



- ◆ **FFAF_SIGN_CHAR_TO_FOLLOW_HEADER**
Set if sign character(s) should follow the header.
- ◆ **FFAF_SIGN_CHAR_TO_PRECEDE_TRAILER**
Set if sign character(s) should precede the trailer.

The rest of the entries in **FloatFloatToAsciiData** store information filled in by **FloatFloatToAscii()**. These entries are described below:

D.2

FFA_startNumber stores the offset to the start of numeric characters in the ASCII buffer.

FFA_decimalPoint stores the offset to the decimal point character or zero if no decimal point is present.

FFA_endNumber stores the offset to the end of the numeric characters in the ASCII buffer.

FFA_numChars stores the total number of characters in the ASCII buffer (excluding the null terminator). This entry is set to zero if an error is encountered.

FFA_startExponent stores the offset to the “E” character in the ASCII buffer or zero if no exponent is present. Applications can check this to see if the number was expressed in scientific notation using the ‘E’ format.

D.2.0.2 Date-Time Conversions

FFA_stackFrame may contain **FloatFloatToAsciiDateTimeData** if **FloatFloatToAscii()** is being used to convert a FP number into a date-time format. In that case **FFA_stackFrame** contains the structure **FloatFloatToDateTimeData** instead of **FloatFloatToAsciiData**. (**FFA_stackFrame** is a union.)

FloatFloatToDateTimeData contains one entry, *FFA_dateTimeParams*. This structure contains several flags which specify how the date-time should be formatted and a number of entries which break down the date-time into its respective parts (year, month, day etc.) If none of these entries are filled in, the date-time is taken from the top of the FP stack.

Date-times are represented by FP numbers in GEOS. The integer portion represents dates as integers counted from Jan 1, 1900, which is designated as 1. The highest date allowed is 73050 (December 31, 2099). The fractional

portion represents a fraction of the day between midnight (0.000000) and 11:59:59 p.m. (0.999988). This fractional value is derived from the hour, minute and second of the day.

Code Display D-5 DateTime Parameters

```
typedef struct {
    FloatFloatToDateTimeFlags    FFA_dateTimeFlags;
    word                         FFA_year;
    byte                         FFA_month;
    byte                         FFA_day;
    byte                         FFA_weekday;
    byte                         FFA_hours;
    byte                         FFA_minutes;
    byte                         FFA_seconds;
} FloatFloatToDateTimeParams;

/*
 *      FloatFloatToDateTimeFlags record
 */

typedef WordFlags FloatFloatToDateTimeFlags;
#define      FFDT_DATE_TIME_OP      0x8000
#define      FFDT_FROM_ADDR      0x4000
#define      FFDT_FORMAT      0x3fff
```

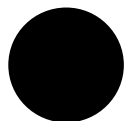
D.2

The flag **FFDT_DATE_TIME_OP** is set to notify the **FloatFloatToAscii()** routine that this operation is a date-time format, not a normal float to ASCII conversion. This flag must be set if you want to convert the FP number into a date-time format using **FloatFloatToAscii()**.

FFDT_FROM_ADDR is set if the date-time FP number should not be taken from the FP stack (or passed directly in the **FloatFloatToDateTimeParams** structure) but should instead be taken from the address passed in **FloatFloatToAscii()**.

FFDT_FORMAT stores the **DateTimeFormat** that the routine will use to format the number into a date-time string.

If the date-time is directly passed in, and not taken from an FP date-time number either at a passed address or the top of an FP stack, **FloatFloatToAscii()** looks at the other passed parameters.



FFA_year specifies the year. The value must be between 1900 and 2099. This is not a one-based year, as it is when presented as a date-time number.

FFA_month is the month of the year, a value between 1 and 12.

FFA_day is the day of a month, a value between 1 and 31.

FFA_hour specifies the hour of the day, a value between 0 and 23. Zero specifies midnight.

D.2

FFA_minutes specifies the minute of the hour, a value between 0 and 59.

FFA_seconds specifies the second of the minute, a value between 0 and 59.

D.2.0.3 Using Standard Formats

FloatFloatToAscii_StdFormat() uses a pre-set stack frame, eliminating the need to set up the variables of the **FloatFloatToAsciiData** structure manually. The only flags recognized are **FFAF_FROM_ADDR**, **FFAF_SCIENTIFIC**, **FFAF_PERCENT**, **FFAF_USE_COMMAS**, and **FFAF_NO_TRAIL_ZEROS**. The developer must pass the number of total digits and the number of decimal digits desired. If the flag **FFAF_FROM_ADDR** is used, a pointer to a FP number (not on the FP stack) must also be passed.

The standard format sets the following elements of the stack frame to zero: **decimalOffset**, **header**, **trailer**, **postNegative**, **prePositive**, and **postPositive**. The structure element **preNegative** is set to the minus sign (“-”).

FloatFloatIEEE64ToAscii_StdFormat() performs the same operation as **FloatFloatToAscii_StdFormat()** except that the FP number is passed (in 64 bit format) and is not taken from the stack. The entire FP number (not just a pointer to it) must be passed. All criteria for **FloatFloatToAscii_StdFormat()** applies to this routine, except that the flag **FFAF_FROM_ADDR** is not used.

D.2.0.4 Date and Time Routines

```
FloatGetDateNumber(), FloatDateNumberGetYear(),  
FloatDateNumberGetMonthAndDay(),
```

```
FloatDateNumberGetWeekday(), FloatGetTimeNumber(),  
FloatTimeNumberGetHour(), FloatTimeNumberGetMinutes(),  
FloatTimeNumberGetSeconds()
```

FloatGetDateNumber(), when passed the month, day, and year, converts the data into an FP “date number” representation. This format represents dates as integers counted from Jan 1, 1900, which is designated as 1. The highest date allowed is 73050 (December 31, 2099).

FloatDateNumberGetYear(),

D.2

FloatDateNumberGetMonthAndDay() and

FloatDateNumberGetWeekday() all return the appropriate data, either the year, month and day, or weekday, given an FP “date number” as defined above. All data are returned as integers, not as FP numbers, and the original FP “date number” is popped off of the stack. Years are returned as integers between 1900 and 2099. Month and Days are returned as integers between 1 and 12, and 1 and 31, respectively. Weekdays are returned as integers between 1 and 7, where 1 is Sunday, 2 is Monday, etc.

FloatGetTimeNumber() when passed hours, minutes, and seconds returns an FP decimal representation between midnight (0.000000) and 11:59:59 p.m. (0.999988).

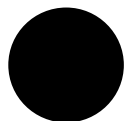
FloatTimeNumberGetHour(), **FloatTimeNumberGetMinutes()** and **FloatTimeNumberGetSeconds()** return the appropriate data given an FP “time number” as defined above. The original FP “time number” is popped off of the stack.

Note that both “date numbers” and “time numbers” can be added to specify a specific point in time. For example, 73050.999988 would specify December 31, 2099, 11:59:59. Since these formats are in FP format, they can be operated on with all standard functions in the FP library.

D.2.0.5 Miscellaneous Date Time Routines

```
FloatGetDaysInMonth(), FloatGetNumDigitsInIntegerPart(),  
FloatFormatNumber()
```

FloatGetDaysInMonth() returns the total number of days in a specific month, for a specific year. The routine must be passed the appropriate month and year.



FloatGetNumDigitsInIntegerPart() returns the number of digits in the integer portion of an FP number. Numbers between zero and one will return one as the number of digits.

`FloatStringGetDateNumber()`, `FloatStringGetTimeNumber()`

D.3

D.3 Float Formats

FP numbers can be displayed in many ways. For example, as we have seen, an FP number may actually represent a date-time. When we display the FP number 366.0000, we may want to how it (in text) as “Jan 1, 1901.” The Math Library has a number of system-defined formats for your use. You may also allow users to define their own formats with the Float Format controller.

The underlying structures and routines to use and create float format options are explained in the next section. In most cases, however, the simple inclusion of a Float Format controller (of **FloatFormatClass**) provides all of the UI and functionality to create and apply formats to FP numbers within text objects.

D.3.1 System-defined Formats

A system-defined FP format is stored within a **FormatParams** structure. This structure defines whether the FP number is a number to be converted into numerical text or a date-time. These format parameters are stored within arrays managed by the format control code.

Code Display D-6 System-defined Float Formats

```
/*
 * System-defined float formats are stored in an array that is maintained and
 * accessed by the float controller code. Each element is made up of a
 * FormatParams structure.
 */
```

```
typedef struct {
    /*
     * The FloatFloatToAsciiParams_Union stores either a
     * FloatFloatToAsciiParams structure if the number is a 'pure' FP number,
     * or a FloatFloatToDateTimeParams structure if the number is a date-time.
     * In this way, it is essentially the same as the FFA_stackFrame discussed
     * earlier.
     */
    FloatFloatToAsciiParams_Union    FP_params;

    /*
     * FP_formatName stores the name of this formatting option that will be
     * displayed in the float controller's dynamic list. This name is loaded
     * from the optr given in FP_nameHan and FP_nameOff. (The table where these
     * strings are kept is within a localizable resource and therefore will
     * have different text under different country setups.)
     */
    char                            FP_formatName[FORMAT_NAME_LENGTH+1];
    word                            FP_nameHan;      /* MemHandle */
    word                            FP_nameOff;      /* ChunkHandle */

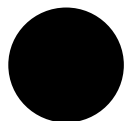
    /*
     * FP_listEntryNum stores the zero-based position of this FormatParams
     * entry within the table.
     */
    word                            FP_listEntryNum;

    /*
     * FP_signature is an internal field used for error-checking.
     */
    word                            FP_signature;
} FormatParams;
```

D.3

An application will never need to access this table of formats directly. GEOS contains several routines (in **math.goh**) that can access this table and add, delete and modify table entries. Usually, it is easiest to include a Float Format controller in your application if you intend to allow the user to change float formats with these routines.

There are many system-defined float formatting options. These formats are identified by **FormatIdType** enumerations. Each type corresponds to a **FormatParams** structure.



Each **FormatIdType** enumeration is a direct offset into the float format lookup table. To distinguish between system-defined and user-defined formats, the high bit of a **FormatIdType** is set to indicate that the format is system-defined. Thus, 8000h refers to the first system-defined format, 8000h + (size(**FormatParams**)) refers to the second system-defined format, etc.

D.3

The format strings themselves are stored within a localizable resource, so that they may appear in a manner relevant to the particular country involved. For example, an FP number of 12.0 using the **FormatIdType** **FORMAT_ID_CURRENCY** might appear in the U.S. as \$12.00, but will appear as £12.00 in Great Britain.

Code Display D-7 Float Format IDs

```
typedef enum {
    FORMAT_ID_GENERAL                = 0x8000,
    FORMAT_ID_FIXED                  = 0x8061,
    FORMAT_ID_FIXED_WITH_COMMAS     = 0x80c2,
    FORMAT_ID_FIXED_INTEGER          = 0x8123,
    FORMAT_ID_CURRENCY               = 0x8184,
    FORMAT_ID_CURRENCY_WITH_COMMAS  = 0x81e5,
    FORMAT_ID_CURRENCY_INTEGER       = 0x8246,
    FORMAT_ID_PERCENTAGE             = 0x82a7,
    FORMAT_ID_PERCENTAGE_INTEGER     = 0x8308,
    FORMAT_ID_THOUSANDS              = 0x8369,
    FORMAT_ID_MILLIONS               = 0x83ca,
    FORMAT_ID_SCIENTIFIC             = 0x842b,

    FORMAT_ID_DATE_LONG              = 0x848c,
    FORMAT_ID_DATE_LONG_CONDENSED   = 0x84ed,
    FORMAT_ID_DATE_LONG_NO_WKDAY     = 0x854e,
    FORMAT_ID_DATE_LONG_NO_WKDAY_CONDENSED = 0x85af,
    FORMAT_ID_DATE_SHORT             = 0x8610,
    FORMAT_ID_DATE_SHORT_ZERO_PADDED = 0x8671,
    FORMAT_ID_DATE_LONG_MD           = 0x86d2,
    FORMAT_ID_DATE_LONG_MD_NO_WKDAY  = 0x8733,
    FORMAT_ID_DATE_SHORT_MD          = 0x8794,
    FORMAT_ID_DATE_LONG_MY           = 0x87f5,
    FORMAT_ID_DATE_SHORT_MY          = 0x8856,
    FORMAT_ID_DATE_YEAR              = 0x88b7,
    FORMAT_ID_DATE_MONTH             = 0x8918,
    FORMAT_ID_DATE_DAY               = 0x8979,
```

```

FORMAT_ID_DATE_WEEKDAY          = 0x89da,
FORMAT_ID_TIME_HMS              = 0x8a3b,
FORMAT_ID_TIME_HM               = 0x8a9c,
FORMAT_ID_TIME_H                = 0x8afd,
FORMAT_ID_TIME_MS               = 0x8b5e,
FORMAT_ID_TIME_HMS_24HR         = 0x8bbf,
FORMAT_ID_TIME_HM_24HR          = 0x8c20,

FORMAT_ID_INDETERMINATE         = 0xffff
} FormatIdType;

```

D.3

D.3.2 User-defined Formats

Users may also design their own float formatting options. These user-defined formats are stored in **FormatEntry** structures. The array of these user-defined formats is kept separate from the system-defined formats. (user-defined formats are stored within their own VM block).

User-defined **FormatIdType** enums do not have the high bit (8000h) set in order to distinguish them from system-defined **FormatIdType** enums.

Code Display D-8 User-defined Formats

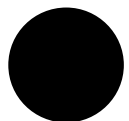
```

/*
 * User-defined formats are stored in a VM block that is created and maintained by
 * the float controller code. As each new format is added, space is made for a
 * FormatEntry structure. As formats are deleted, their entry's FE_used field is
 * set to zero to indicate that the entry is free for new formats to overwrite.
 */

typedef struct {
    /*
     * Each FormatEntry contains a corresponding FormatParams structure.
     */
    FormatParams    FE_params;

    /*
     * FE_listEntryNumber is the zero-based position of the format counting
     * both previous user-defined formats and system-defined formats. For
     * example, if there are 10 system-defined formats, the first user-defined

```



D.4

```
    * format will have a FE_listEntryNumber of 10 (because positions are
    * zero-based).
    */
word                FE_listEntryNumber;

/*
 * FE_used, if non-zero, indicates that this entry within the user-defined
 * list is currently in use. If FE_used is zero, then this entry position
 * may be used to add a new user-defined entry without increasing the size
 * of the VM block.
 */
byte                FE_used;

/*
 * FE_sig is an error-checking field used internally.
 */
word                FE_sig;
} FormatEntry;
```

D.4 Direct FP Operations

The Math Library allows your application to use floating point (FP) numbers. C Developers will find little reason to make direct calls to functions in the Math Library as most of this functionality is taken care of for you in the C environment. (An exception is conversions from FP numbers to ASCII text characters and use of special date-time routines.) Assembly developers, however, will find direct use of the functions and structures in the Math Library essential.

The latter half of this chapter (from this point on) is provided for Assembly developers who need more complete information about the intricacies of the Math Library. Much of this information may also be useful for C developers who wish to make direct calls to the functions in this library for optimization purposes.

D.4.1 Floating Point Numbers

The Math Library defines floating point numbers by using binary point representation. In this format the bit in the zeros place is multiplied by 2^0 , the bit in the minus-ones place is multiplied by 2^{-1} , etc. The exponent is in base 2. For example, the Binary Point representation for 5 is:

$$\begin{aligned} &1.01^{10} \\ &= (1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}) \times (2^2) \\ &= (1.25) \times 4 \\ &= 5 \end{aligned}$$

D.4

Figure D-1 *Translating FP to decimal*

Binary Point Representation for the number five, translated

FP Format

$+3.2 \times 10^4$ is a floating point representation of 32,000. In the above example, the mantissa is 3.2 and the exponent is 4.

Floating Point Numbers in GEOS follow the IEEE 754 standard used by Intel. In this format a floating point number is represented in an 80 bit (5 word) format. (In C, this type is known as a **long double**.) This format specifies that the 80 bits contain:

- ◆ a 1 bit sign (the most significant bit).
- ◆ a 15 bit exponent (0000h to 7FFFh).
- ◆ a 64 bit mantissa.

The 15 bit exponent is biased by 3FFFh, so that an exponent of 1 would be represented by 4000h, and an exponent of -1 would be represented by 3FFEh. This produces a hexadecimal range for the exponent of $\pm 4000h$, or a decimal range of ± 4932 .

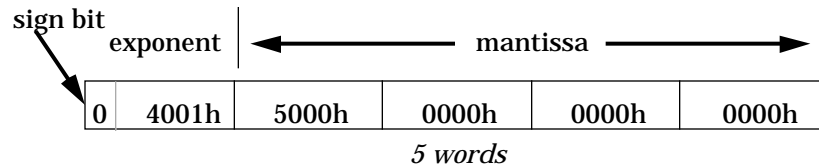
Infinity is represented by 7FFFh in the exponent and 1 followed by 63 zeros in the mantissa. Therefore, all exponent bits are set to 1. (Negative infinity is the same except that the sign bit is also set to 1).

Zero is a special case in which all 80 bits— sign, exponent, and mantissa—are set to zero.

The mantissa is normalized so that the most significant bit is always 1 (except when the FP number is zero). The binary point follows this 1 bit. This produces a precision of approximately 19 decimal places, which is



adequate for most needs. This most significant bit is not “assumed away”; it is always present in the mantissa.



D.4

Figure D-2 Diagram of an 80-bit Floating Point Number

Here we see $5(1.25 \times 2^2)$ represented as a floating point number.

Note that in most uses, the use of binary point representation is transparent to the application.

D.4.2 The Floating Point Stack

Floating Point (FP) numbers are placed and manipulated on an FP stack. Numbers can be rearranged, operated on, and removed from this stack through pushes, pops, and the use of special routines.

D.4.2.1 Initialization of the FP Stack

`FloatInit()`, `FloatExit()`

Before performing any floating point operations, a thread needs to call **FloatInit()** to create and initialize an FP stack. Each thread using floats must have its own unique FP stack. This call to **FloatInit()** is automatically performed by any application that includes the Math Library.

FloatInit() creates a swappable block of memory for the thread, initializes various stack pointers, and stores the handle for the block in the thread's data structure. **FloatInit()** must be passed the size of the stack to create (in bytes) and the type of stack (**FloatStackType**) to create.

The default FP stack holds 25 FP elements (250 bytes). An FP stack must be able to hold at least 5 FP elements.

The default floating point stack is `FLOAT_STACK_GROW` which instructs the system to increase the size of the stack whenever its bounds are reached. This is done automatically.

Other **FloatStackType** types are `FLOAT_STACK_WRAP`, which drops the FP numbers at the low end of the stack (effectively wrapping over that end) and `FLOAT_STACK_ERROR` which signals an error when the stack limit has been reached.

FloatExit() detaches the floating point stack for the current thread and frees its memory. **FloatExit()** only frees the FP stack associated with the current thread; other FP stacks in other threads remain unaffected. As is the case with **FloatInit()**, the call to **FloatExit()** is automatically performed by any application that includes the Math Library.

D.4

If **FloatInit()** is called twice before calling **FloatExit()**, the data on the original floating point stack will be lost.

D.4.2.2 Pushing and Popping on the FP Stack

`FloatPushNumber()`, `FloatPopNumber()`, `FloatDepth()`

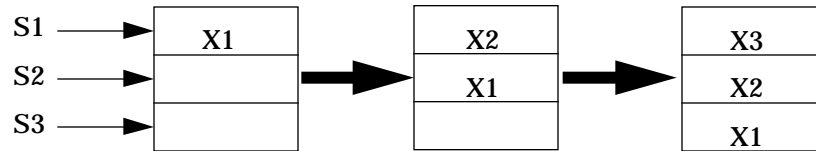
FloatPushNumber() pushes an FP number onto the top of the FP stack for the current thread from a passed buffer. The number must be already set up in 80 bit, FP format. The routine must be passed the pointer to the buffer storing the number.

Similarly, **FloatPopNumber()** pops an FP number from the top of the FP stack for the current thread into a passed buffer.

FloatDepth() returns the number of FP numbers currently in place on the stack.

Note: For clarity in diagrams within this chapter, stack locations will be numbered in order from the top position of the stack, S1 being first, S2 being second, etc. Variables will be numbered in the order they are pushed

onto the stack, so that if X1, X2, and X3 are pushed onto the stack, the format in Figure D-3 will result.



D.4

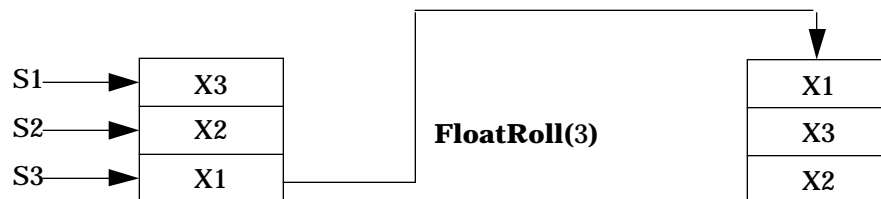
Figure D-3 *The Floating Point Stack*

D.4.2.3 FP Stack Manipulation

`FloatRoll()`, `FloatRollDown()`, `FloatRot()`, `FloatSwap()`

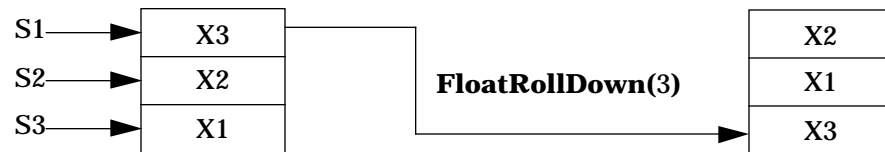
Besides basic pushing and popping, the Float Library also includes many other routines which manage FP numbers on the stack. Unless otherwise specified, an operation that pushes, pops, or extracts an FP number on the stack affects all other FP numbers below the position of the operation by shifting their location in the stack either up or down, in standard stack fashion.

FloatRoll() pushes a selected FP number (SX) onto the top of the stack (S1), removing it from location SX in the process. **FloatRoll()** passed with a value of 3 would move the FP number in S3 onto the top of the stack, pushing the stack in the process. All FP numbers below the extracted number remain unaffected by this routine.



FloatRollDown() performs the inverse operation of **FloatRoll()**, popping the top stack value (S1) into the specified location on the stack (SX).

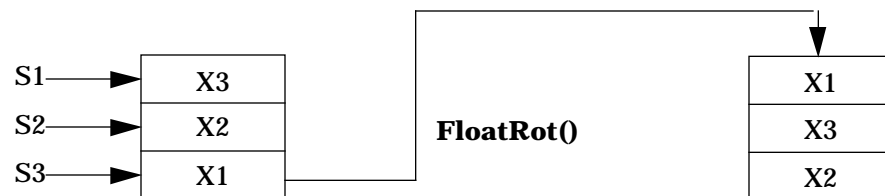
FloatRollDown() passed with a value of 3 would move the FP number in S1 into location S3, shifting the stack in the process.



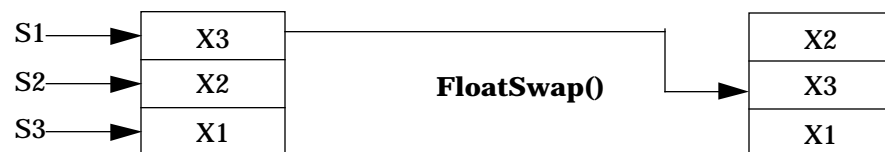
Both of these routines must be passed a stack location to move to or from.

D.4

FloatRot() rotates the top three numbers on the stack, placing S3 onto the top of the stack. This is equivalent to a **FloatRoll()** passed with a value of 3.



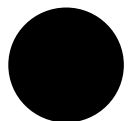
FloatSwap() exchanges S1 and S2.



Repetitious applications of these routines will return the stack to its former state.

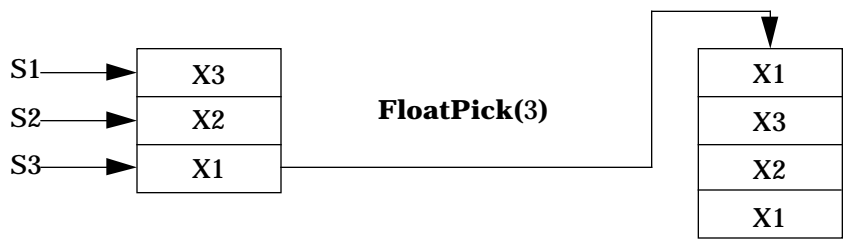
D.4.2.4 Copying and Deleting Numbers on the FP Stack

`FloatPick()`, `FloatOver()`, `FloatDrop()`, `FloatDup()`



D.4

FloatPick() copies the contents of *SX* and pushes that value onto the FP stack. The entire stack is pushed in the process. **FloatPick()** passed with a value of 3 would copy the contents of S3 onto the FP stack.



FloatOver() copies S2 to the top of the stack, equivalent to a **FloatPick()** passed with a value of 2.

FloatDrop() drops (pops) the top number (S1) off the FP stack. This routine is different than **FloatPopNumber()** because the routine does not place the popped number into a memory address, and is therefore much faster.



FloatDup() duplicates the value at S1, pushing it onto the top of the stack. The stack is pushed in the process.



D.4.2.5 Comparing Numbers on the FP Stack

`FloatComp()`, `FloatCompESDI()`, `FloatCompAndDrop()`

These routines essentially perform the same operation as the Assembly command **cmp**. **FloatComp()** performs a compare of the top two FP numbers (S1 and S2) and sets the appropriate flags in the flags register. The two FP numbers remain on the stack. **FloatCompESDI()** compares the contents of **es:[di]** with the value in S1 (and the FP number in S1 remains on the stack). **FloatCompAndDrop()** performs a compare of S1 and S2 and drops both from the FP stack.

D.4

D.4.2.6 Recovery of the FP Stack

`FloatGetStackPointer()`, `FloatSetStackPointer()`

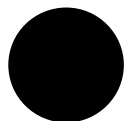
FloatGetStackPointer() returns the current stack pointer value of the FP stack. **FloatSetStackPointer()** sets the stack pointer to a previous position. This routine must be passed a value that is greater than or equal to the current value of the stack pointer. (I.e. you must be throwing something, or nothing, away.)

These routines may be useful before the execution of involved routines that may push many numbers onto the stack. If an unrecoverable error is encountered, the programmer need not pop the intermediate values off the stack to return to the previous stack configuration. Only the stack pointer is saved; the state of the stack is not. If any numbers below the stack pointer are popped or altered, **FloatSetStackPointer()** will not recover the previous state of the stack.

D.4.3 Floating Point Math Routines

The Float Library provides a number of routines to modify data on the FP stack. These routines can be categorized in several major groups:

- ◆ Constant Routines
- ◆ Math Routines
- ◆ Transcendental Routines



- ◆ Conversion Routines
- ◆ Date and Time Routines

D.4.3.1 Constant Routines

D.4

`Float0()`, `FloatPoint5()`, `Float1()`, `FloatMinusPoint5()`,
`FloatMinus1()`, `Float2()`, `Float5()`, `Float10()`, `Float3600()`,
`Float16384()`, `Float86400()`

`FloatPi()`, `FloatPiDiv2()`, `FloatLg10()`, `FloatLn2()`,
`FloatLn10()`, `FloatSqrt2()`

The Constant Routines provide a means of quickly obtaining an often used number or an often used operation with a constant operand. Each of these functions pushes the constant FP value onto the top of the FP stack. (For example, **FloatMinusPoint50** pushes -.5 onto the FP stack.)

FloatPi0, **FloatPiDiv20**, **FloatLg100**, **FloatLn20**, **FloatLn100**, and **FloatSqrt20** each push the specified transcendental constant onto the FP stack: π , $\pi/2$, the log of 10, the natural log of 2, the natural log of 10, and the square root of 2, respectively. (See also Transcendental Routines.)

D.4.3.2 Constant Operands

`FloatMultiply2()`, `FloatMultiply10()`, `FloatDivide2()`,
`FloatDivide10()`, `Float10ToTheX()`

FloatMultiply20, **FloatMultiply100**, **FloatDivide20**,
FloatDivide100 perform the specified operations on the contents of S1, either multiplying or dividing the contents of S1 by 2 or 10, and push the result onto the FP stack. The original value in S1 is popped off of the stack.

Float10ToTheX0 pushes 10 to a passed exponent onto the FP stack.

D.4.3.3 Algebraic Routines

`FloatAbs()`, `FloatAdd()`, `FloatSub()`, `FloatDivide()`,
`FloatMultiply()`, `FloatDIV()`, `FloatMod()`, `FloatFactorial()`,
`FloatNegate()`, `FloatInverse()`

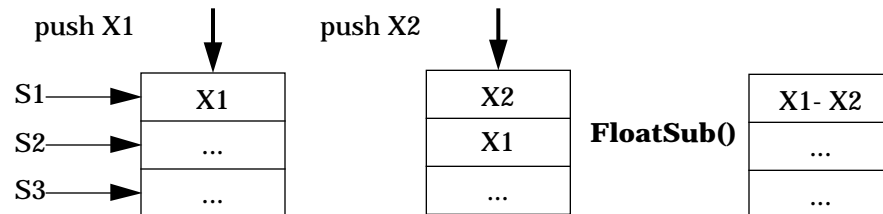
The Math Routines perform various operations on numbers already placed on the FP stack. All of the following routines pop the operated numbers off of the stack and push the result onto S1.

FloatAdd() adds the contents of S1 and S2, and pushes the result onto the top of the FP stack. The original values are popped off the stack.

FloatSub() subtracts the contents of S1 from the contents of S2, and pushes the result onto the top of the FP stack. The original values are popped off the stack.

D.4

Note that, in general, numbers will be pushed onto the stack in the order they would normally be operated on. For example, to obtain $X1 - X2$, the programmer would first push X1, then X2, and call **FloatSub**, as in the following diagram:



since X1 would now be in location S2, while X2 is in location S1.

FloatMultiply() multiplies the contents of S1 and S2 and pushes the result onto the top of the FP stack. The original values are popped off the stack.

FloatDivide() divides the contents of S2 by the contents of S1 and pushes the result onto the top of the FP stack. The original values are popped off the stack.

FloatDIV() performs a **FloatDivide()**, truncating the fractional portion of the number and returning only the integer result. The original values are popped off the stack.

FloatMod() pushes $S2 \bmod S1$ (the remainder of **FloatDivide()**) onto the top of the FP stack. The original values are popped off the stack.



FloatAbs() pushes the absolute value of S1 onto the top of the FP stack. The original value is popped off the stack.

FloatFactorial() pushes the factorial (x!) of the value in S1 onto the top of the FP stack. The original value is popped off the stack.

FloatNegate() negates the value in S1 and pushes the result onto the top of the FP stack. The original value is popped off the stack.

D.4

FloatInverse() pushes the inverse of the value in S1 (-S1) onto the top of the FP stack. The original value is popped off of the stack.

D.4.3.4 Comparison Routines

`FloatMax()`, `FloatMin()`, `FloatLt0()`, `FloatEq0()`, `FloatGt0()`

FloatMax() performs a compare of the FP numbers in S1 and S2 and, if necessary, swaps the greater number into S1.

FloatMin() performs a compare of the FP numbers in S1 and S2 and, if necessary, swaps the lesser number into S1.

FloatLt0, **FloatEq0**, **FloatGt0** check whether the FP number in S1 is less than zero, equal to zero, or greater than zero. The carry bit is set to TRUE if the relationship is true. The original value is popped off of the stack.

D.4.3.5 Fractional and Integral Routines

`FloatFrac()`, `FloatTrunc()`, `FloatInt()`, `FloatIntFrac()`,
`FloatRound()`

FloatFrac() pushes the fractional portion of S1 onto the FP stack. The original value is popped off of the stack.

FloatTrunc() pushes the integral portion of the contents of S1 onto the FP stack. This amounts to a rounding of the FP number toward zero, so that **FloatTrunc()** performed on -7.8 would return -7. The original value is popped off the stack.

FloatInt() rounds S1 down to its integral component, so that **FloatInt()** performed on -7.8 would return -8. Note that for negative numbers, this is different from **FloatTrunc()**. The original value is popped off the stack.

FloatIntFrac() splits a number into its fractional and integral parts, with the fractional part in S1 and the integral part in S2. The original value is popped off the stack.

FloatRound() rounds S1 to a given number of decimal places.

FloatRound() passed with zero as an argument rounds S1 to the nearest integer, rounding up if greater than or equal to .5, rounding down if less than .5

D.4

D.4.3.6 Routines that Return Random Values

`FloatRandom()`, `FloatRandomN()`, `FloatRandomize()`

FloatRandom() pushes a random number between 0 (inclusive) and 1 (exclusive) onto the stack.

FloatRandomN() pushes a random integer between 0 (inclusive) and N (exclusive) onto the stack.

FloatRandomize() primes the random number generator. This routine expects a seed and some **RandomGenInitFlags**. If the flag `RGIF_USE_SEED` is passed, then a developer-supplied seed will be used. Otherwise, a seed based on the timer clock will be used.

FloatRandomize() should always be called before any of the **FloatRandom()** routines to ensure a high degree of randomness.

The random number generation method uses the linear congruential method, an algorithm which ensures a high degree of randomness in the computation method.

D.4.3.7 Transcendental Functions

Transcendental functions are functions that cannot be constructed using normal arithmetic routines. Care must be taken with the following routines to ensure that they are operating on a valid range of values. Otherwise, the routines will return an error.



Trigonometric Routines

```
FloatSin(), FloatCos(), FloatTan(), FloatArcSin(),  
FloatArcCos(), FloatArcTan(), FloatArcTan2(),  
FloatSinh(), FloatCosh(), FloatTanh(), FloatArcSinh(),  
FloatArcCosh(), FloatArcTanh()
```

D.4

FloatSin(), **FloatCos()**, and **FloatTan()** perform the given operation on the contents of S1, pushing the result onto the FP stack. S1 must be expressed in radians. The original value is popped off of the stack.

FloatArcSin(), **FloatArcCos()**, and **FloatArcTan()** perform the given inverse operation on S1, pushing the result onto the FP stack. The result is given in radians. The original value is popped off of the stack.

FloatArcTan2() calculates the arc tangent given cartesian coordinates x and y . The arctangent is calculated from the x -axis through the origin to the given point. The value returned is expressed in radians, between $-\pi$ (exclusive) and $+\pi$ (inclusive). The original values are popped off of the stack.

FloatSinh(), **FloatCosh()**, and **FloatTanh()** perform the given hyperbolic operations on S1, pushing the result onto the FP stack. The original value is popped off of the stack.

FloatArcSinh(), **FloatArcCosh()**, and **FloatArcTanh()** perform the given inverse hyperbolic operations on S1, pushing the result onto the FP stack. The original value is popped off of the stack.

Exponential Routines

```
FloatExp(), FloatExponential(), FloatLg(), FloatLog(),  
FloatLn(), FloatLn1plusX(), FloatSqr(), FloatSqrt()
```

FloatExp() performs the exponentiation of e to the power of S1, pushing the result onto the FP stack. The original value is popped off of the stack.

FloatExponential() performs the exponentiation of S2 to the power of S1, pushing the result onto the FP stack. The original values are popped off of the stack.

FloatLg() performs the logarithm to the base 2 on S1, pushing the result onto the FP stack. The original value is popped off of the stack.

FloatLog() performs the logarithm to the base 10 on S1. The original value is popped off the stack.

FloatLn() performs the natural logarithm (log base e) on S1. The original value is popped off the stack.

FloatLn1plusX() performs the natural log of $(1 + S1)$. The original value is popped off the stack.

FloatSqr() performs the square of S1, pushing the result onto the FP stack. The original value is popped off the stack.

D.4

FloatSqrt() performs the square root of S1, pushing the result onto the FP stack. The original value is popped off the stack.

D.4.3.8 Conversion Routines

At some point 80-bit FP numbers may need to be converted into different formats, or you may need to convert these differently formatted numbers into a floating point representation. The Math Library provides for this contingency.

Conversions Between Integers and FP Numbers

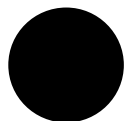
```
FloatDwordToFloat(), FloatWordToFloat(),  
FloatFloatToDword()
```

FloatDwordToFloat() converts a passed double-word signed integer into a floating point number, and pushes that number onto the FP stack.

FloatWordToFloat() converts a passed word signed integer into a floating point number, and pushes that number onto the FP stack.

FloatFloatToDword() converts the FP number in S1 into a double-word signed integer. The FP number is converted into an integer by rounding the FP number to zero decimal places.

You can convert a FP number into a word value by using **FloatFloatToDword()** and just using the low word.



Conversions Between 80 bit FP Numbers and Other Floats

```
FloatGeos80ToIEEE64(), FloatGeos80ToIEEE32(),  
FloatIEEE64ToGeos80(), FloatIEEE32ToGeos80()
```

FloatGeos80ToIEEE64() converts a GEOS FP number (80 bits) into a 64 bit floating point number (in C, a type of **double**).

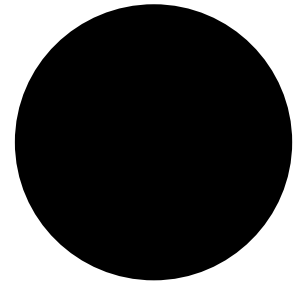
D.4

FloatGeos80ToIEEE32() converts a GEOS FP number (80 bits) into a 32 bit floating point number (in C, a type of **float**).

FloatIEEE64ToGeos80() converts a 64 bit FP number into a GEOS 80 bit FP number (in C, a type of **long double**).

FloatIEEE32ToGeos80() converts a 32 bit FP number into a GEOS 80 bit FP number (in C, a type of **long double**).

Credits



Technical Writers

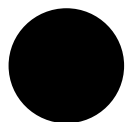
Peter Dudley
Larry H.
Tom Manshreck
Peter J. Secor
Andrew M. Solovay

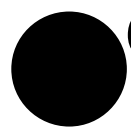
Engineers

ardeb
Jim DeFrisco
Paul DuBois
Douglas A. Fults
Jennifer K. Greenwood
Jimmy Lefkowitz
Thomas Lester
John D. Mitchell
Josh Putnam
Tony Requist
Joon Song
Jon Witort
Steve Yegge

Other

Ed Ballot
Paul Chen
Clayton Jung
Mark Troy
Michael Zern





Index

"
 localizing CLocal : 343
& CCoding : 142
.goc files CGetSta : 107
 see also Source files
.goh files CGetSta : 108
 see also Header files
.gp file CGetSta : 114
.gp files CGetSta : 107
 see also Parameter files
.h files CGetSta : 107
 see also Header files
.ini file: see Initialization file
@ (object pointer shortcut) CCoding : 226
@alias CCoding : 190
@call CCoding : 157, CCoding : 220
@callsuper CCoding : 222
@chunk CCoding : 210
@chunkArray CCoding : 211
@class CCoding : 184
@classdecl CCoding : 185
@composite CCoding : 193
@default CCoding : 218
@define CCoding : 152
@deflib CCoding : 209
@dispatch CCoding : 223
@dispatchcall CCoding : 223
@elementArray CCoding : 211
@end CCoding : 210
@endc CCoding : 184
@endif CCoding : 151
@endlib CCoding : 209
@exportMessages CCoding : 190
@extern CCoding : 211
 with methods CCoding : 205
@genChildren (messaging shortcut) CCoding : 226
@genParent (messaging shortcut) CCoding : 226
@header CCoding : 210
@if CCoding : 151
@ifdef CCoding : 151
@ifndef CCoding : 151
@importMessage CCoding : 190
@include CCoding : 150
@instance CCoding : 192
@kbdAccelerator CCoding : 217
@link CCoding : 193
@message CCoding : 187

@method CCoding : 204
@noreloc Goc keyword CCoding : 203
@object CCoding : 212–CCoding : 213
@prototype CCoding : 190
@record CCoding : 223
@reloc CCoding : 202
@reserveMessages CCoding : 190
@send CCoding : 219
@specificUI CCoding : 217
@stack CCoding : 188
@start CCoding : 210
@vardata CCoding : 197
@vardataAlias CCoding : 197
@visChildren (messaging shortcut) CCoding : 226
@visMoniker CCoding : 215
@visParent (messaging shortcut) CCoding : 226
| CCoding : 142
 localizing CLocal : 343

A

ABS CParse : 762
ACOS CParse : 762
ACOSH CParse : 762
alt (keyboard accelerator modifier) CCoding : 217
AND CParse : 762
application (messaging shortcut) CCoding : 225
Application object
 accessing CAppl : 259
Applications
 launching CAppl : 247–CAppl : 249
 multithreaded CMultit : 916–CMultit : 931
 samples CGetSta : 107–CGetSta : 137,
 CUIOver : 391–CUIOver : 415
 saving to state file CAppl : 250–CAppl : 253
 shutting down CAppl : 249–CAppl : 250
 structure CGetSta : 107–CGetSta : 108
 see also Geodes
ArcCloseType CShapes : 863
Arcs CShapes : 863
ArrayQuickSort() CLMem : 592
ASCII text CLocal : 350
 number strings CLocal : 341
ASIN CParse : 762

ASINH CParse : 762
ATAN CParse : 762
ATAN2 CParse : 762
ATANH CParse : 762
AVG CParse : 762

B

BBFixed CCoding : 147
BBFixedAsWord CCoding : 147
Beeps CSound : 506
Bézier curves CShapes : 867
BI_... CInput : 429
BitmapMode CGraph : 830
Bitmaps CGraph : 829–CGraph : 831,
CShapes : 870–CShapes : 872
mouse pointer image CInput : 434
BLTM_... CGraph : 853
BLTMode CGraph : 853
BMD_... CGraph : 830
BMDestroy CGraph : 830
Boolean CCoding : 141, CCoding : 142
ButtonInfo CInput : 429
byte CCoding : 141
ByteEnum CCoding : 142
ByteFlags CCoding : 142

C

CALLBACK_... CFile : 629–CFile : 630
calloc() CMemory : 565
canDiscardIfDesperate (message flag)
CCoding : 221
cdrom library CArch : 101
cell library CDB : 731–CDB : 740
CellDeref() CDB : 737
CellDirty() CDB : 737
CellFunctionParameters CDB : 735
CellGetDBItem() CDB : 737
CellGetExtent() CDB : 738
CellLock() CDB : 737
CellLockGetRef() CDB : 737
CellReference CParse : 748
CellReplace() CDB : 736
CF_... CInput : 440–CInput : 441
CFF_... CLocal : 343
CHAR CParse : 762
CharFlags CInput : 440
checkDuplicate (message flag) CCoding : 221
checkLastOnly (message flag) CCoding : 221
CHOOSE CParse : 762
Chunk arrays CLMem : 584–CLMem : 595
Chunk handles CCoding : 146, CLMem : 572

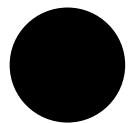
ChunkArrayAppend() CLMem : 588
ChunkArrayAppendHandles() CLMem : 588
ChunkArrayCreate() CLMem : 586
ChunkArrayCreateAt() CLMem : 587
ChunkArrayCreateAtHandles() CLMem : 587
ChunkArrayDelete() CLMem : 588
ChunkArrayDeleteHandle() CLMem : 589
ChunkArrayDeleteRange() CLMem : 589
ChunkArrayDeleteRangeHandles() CLMem : 589
ChunkArrayElementResize() CLMem : 589
ChunkArrayElementResizeHandles()
CLMem : 589
ChunkArrayElementToPtr() CLMem : 589
ChunkArrayElementToPtrHandles() CLMem : 589
ChunkArrayEnum() CLMem : 590
ChunkArrayEnumHandles() CLMem : 591
ChunkArrayEnumRange() CLMem : 591
ChunkArrayGetCount() CLMem : 590
ChunkArrayGetCountHandles() CLMem : 590
ChunkArrayGetElement() CLMem : 590
ChunkArrayGetElementHandles() CLMem : 590
ChunkArrayInsertAt() CLMem : 588
ChunkArrayInsertAtHandles() CLMem : 588
ChunkArrayPtrToElement() CLMem : 590
ChunkArrayPtrToElementHandle() CLMem : 590
ChunkArraySort() CLMem : 591
ChunkArraySortHandles() CLMem : 592
ChunkArrayZero() CLMem : 590
ChunkArrayZeroHandles() CLMem : 590
Chunks CLMem : 570–CLMem : 583
declaring with @chunk CCoding : 210
declaring with @chunkArray CCoding : 211
declaring with @elementArray CCoding : 211
declaring with @object CCoding : 212
declaring with @visMoniker CCoding : 193
instance chunk structure CCoding : 159
relation to db item blocks CDB : 720
CIF_... CClipb : 308
CIH_... CClipb : 303–CClipb : 304
Class_... CCoding : 164–CCoding : 171
Classed events
with input heirarchies CInput : 452
Classes ??–CCoding : 207
class tables
illustrated CCoding : 161
naming conventions CGetSta : 113
CLASSF_... CCoding : 169–CCoding : 170
ClassStruct CCoding : 165
CLEAN CParse : 762
Clipboard CClipb : 299–CClipb : 331
ClipboardAbortQuickTransfer() CClipb : 331
ClipboardAddToNotificationList() CClipb : 311
ClipboardClearQuickTransferNotification()
CClipb : 331
ClipboardDoneWithItem() CClipb : 310

ClipboardEndQuickTransfer() CClipb : 330
ClipboardEnumItemFormats() CClipb : 322
ClipboardGetClipboardFile() CClipb : 323
ClipboardGetItemInfo() CClipb : 323
ClipboardGetNormalItemInfo() CClipb : 323
ClipboardGetQuickItemInfo() CClipb : 330
ClipboardGetQuickTransferStatus() CClipb : 329
ClipboardGetUndoItemInfo() CClipb : 323
ClipboardItemFormat CClipb : 308
ClipboardItemFormatInfo
 CClipb : 305–CClipb : 306
ClipboardItemHeader CClipb : 303
ClipboardQueryArgs CClipb : 307
ClipboardQueryItem() CClipb : 310
ClipboardQuickTransferFeedback CClipb : 329
ClipboardRegisterItem() CClipb : 310
ClipboardRemoveItemFromNotificationList()
 CClipb : 322
ClipboardRequestArgs CClipb : 307
ClipboardSetQuickTransferFeedback()
 CClipb : 329
ClipboardStartQuickTransfer() CClipb : 327
ClipboardTestItemFormat() CClipb : 322
ClipboardUnregisterItem() CClipb : 323
Clipping CGraph : 805, CGraph : 854
cmps
 Compared to LocalCmpStrings() CLocal : 348
CODE CParse : 763
code
 conditional CCoding : 151
Color CShapes : 886
ColorQuad CShapes : 886
COLS CParse : 763
Composite objects CCoding : 156
Constants
 Floating-Point CMath : 972
 naming conventions CGetSta : 113
 Operands CMath : 972
 Routines CMath : 972
ConstructOptr() macro CCoding : 146
control (keyboard accelerator modifier)
 CCoding : 217
Coordinates CGraph : 810–CGraph : 824
 parameterized CShapes : 875
Copy: see Clipboard
Core block CAppl : 243
COS CParse : 763
COSH CParse : 763
CQTF_CLEAR CClipb : 329
CTERM CParse : 763
CTIEnvelopeFormat CSound : 522
ctrl (keyboard accelerator modifier) CCoding : 217
Currency
 formatting CLocal : 343
CURRENCY_SYMBOL_LENGTH CLocal : 343
CurrencyFormatFlags structure CLocal : 343

Current position (graphics) CGraph : 824
Cut: see Clipboard

D

DA_... CAppl : 255
DashPairArray CShapes : 900
data (@start flag) CCoding : 210
Data types CCoding : 141
Date
 formatting CLocal : 344
 Math Conversions CMath : 956
DATE_TIME_BUFFER_SIZE CLocal : 346
DATE_TIME_FORMAT_SIZE CLocal : 346
DateTimeFormat CLocal : 345
DB groups CDB : 721
DB items CDB : 720
db library CDB : 719–CDB : 731
DBAlloc() CDB : 725
DBAllocUngrouped() CDB : 729
DBCombineGroupAndItem() macro CDB : 731
DBCopDBItem() CDB : 730
DBCopDBItemUngrouped() CDB : 730
DBDeleteAt() CDB : 728
DBDeleteAtUngrouped() CDB : 729
DBDeref() CDB : 727
DBDirty() CDB : 727
DBFree() CDB : 726
DBFreeUngrouped() CDB : 729
DBGGetMap() CDB : 729
DBGGroupAlloc() CDB : 725
DBGGroupAndItem CDB : 723
DBGGroupFree() CDB : 725
DBGGroupFromGroupAndItem() macro CDB : 731
DBInsertAt() CDB : 728
DBInsertAtUngrouped() CDB : 729
DBItemFromGroupAndItem() macro CDB : 731
DBLock() CDB : 726
DBLockGetRef() CDB : 726
DBLockMap() CDB : 729
DBLockUngrouped() CDB : 729
DBLockUngroupedGetRef() CDB : 729
DBReAlloc() CDB : 728
DBReAllocUngrouped() CDB : 729
DBSetMap() CDB : 728
DBSetMapUngrouped() CDB : 729
DBUnlock() CDB : 727
DDB CParse : 763
Deadlock CCoding : 182, CMultit : 927
 avoidance CMemory : 558
Debugging
 multithreaded applications CMultit : 924
 with error checking code CAppl : 280
 see also swat



DEFAULT_FONT_ID CShapes : 879
 DEFAULT_FONT_SIZE CShapes : 879
 DES_... CFile : 618
 Device drivers: see Drivers
 DFF_... CFile : 627
 DFR_... CFile : 622–CFile : 623
 Dialog boxes CGetSta : 111
 Directories (file system) CFile : 614
 Directory stack CFile : 638
 DIS_... CAppl : 254–CAppl : 256
 Discardable memory CMemory : 547
 discardOnSave (class flag) CCoding : 185
 DiskCheckInUse() CFile : 623
 DiskCheckUnnamed() CFile : 622
 DiskCheckWritable() CFile : 623
 DiskCopy() CFile : 629
 DiskCopyCallback CFile : 629
 DiskCopyError CFile : 630
 DiskFind() CFile : 622
 DiskFindResult CFile : 622
 DiskForEach() CFile : 623
 DiskFormat() CFile : 627
 DiskFormatFlags CFile : 627
 DiskGetDrive() CFile : 622
 DiskGetVolumeFreeSpace() CFile : 622
 DiskGetVolumeInfo() CFile : 621
 DiskHandle CCoding : 145, CFile : 614
 DiskInfoStruct CFile : 621
 DiskRegisterDisk() CFile : 620
 DiskRegisterDiskSilently() CFile : 620
 DiskRestore() CFile : 624
 DiskRestoreError CFile : 625
 Disks
 handles CCoding : 144
 DiskSave() CFile : 624
 DiskSetVolumeName() CFile : 626
 DiskVolumeName() CFile : 622
 Display control objects: see GenDisplayGroup,
 GenDisplayControl 21
 Distances
 formatting CLocal : 341
 DistanceUnit CLocal : 342
 Dithering
 clustered vs. dispersed CGraph : 830
 Document control objects: see GenDocumentGroup,
 GenDocumentControl
 Documents
 management CArch : 88
 DosCodePage CLocal : 351
 DosExec() CAppl : 279
 DOTTED_... (Music Note Durations) CSound : 508
 DOUBLE_DOT_... (Music Note Durations
 CSound : 508
 DPI_... CFile : 660
 DR_SERIAL_GET_FORMAT CStream : 786
 DR_SERIAL_SET_FORMAT CStream : 784

DR_STREAM_CLOSE
 with serial port CStream : 788
 DR_STREAM_CREATE CStream : 775
 DR_STREAM_OPEN
 with serial port CStream : 783, CStream : 789
 DRE_... CFile : 625–CFile : 626
 Drive number CFile : 614
 DRIVE_... CFile : 617–CFile : 618
 DriveGetDefaultMedia() CFile : 618
 DriveGetExtStatus() CFile : 618
 DriveGetName() CFile : 619
 DriveGetStatus() CFile : 617
 DRIVER_TYPE_... CAppl : 255–CAppl : 256
 DriverAttrs CAppl : 254
 DriverInfoStruct CAppl : 254
 Drivers CAppl : 254–CAppl : 257
 geode attribute CAppl : 246
 standard paths CFile : 634–CFile : 635
 see also Geodes
 DriveStatus CFile : 617
 DriveTestMediaSupport() CFile : 619
 DriveType CFile : 617
 DS_... CFile : 617
 DTF_... CLocal : 345
 DU_... (DistanceUnit) CLocal : 343
 DWFixed CCoding : 148
 dword CCoding : 141
 DWordFlags CCoding : 142
 DYNAMIC_... CSound : 509

E

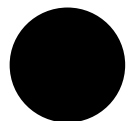
EC macro CAppl : 281
 EC: see Error checking
 EC_BOUNDS macro CAppl : 281
 EC_ERROR macro CAppl : 281
 EC_ERROR_IF macro CAppl : 281
 EIGHTH CSound : 508
 Element arrays CLMem : 595–CLMem : 602
 ElementArrayAddElement() CLMem : 597
 ElementArrayAddElementHandles() CLMem : 598
 ElementArrayAddReference() CLMem : 598
 ElementArrayAddReferenceHandles()
 CLMem : 598
 ElementArrayCreate() CLMem : 596
 ElementArrayCreateAt() CLMem : 597
 ElementArrayDelete() CLMem : 600
 ElementArrayDeleteHandles() CLMem : 600
 ElementArrayElementChanged() CLMem : 599
 ElementArrayGetUsedCount() CLMem : 601
 ElementArrayGetUsedCountHandles()
 CLMem : 601
 ElementArrayHeader CLMem : 596
 ElementArrayRemoveReference() CLMem : 600

ElementArrayRemoveReferenceHandles()
 CLMem : 600
 ElementArrayTokenToUsedIndex() CLMem : 602
 ElementArrayTokenToUsedIndexHandles()
 CLMem : 602
 ElementArrayUsedIndexToToken() CLMem : 601
 ElementArrayUsedIndexToTokenHandles()
 CLMem : 602
 Ellipses CShapes : 862
 EMS CMemory : 544
 Enumerated types CCoding : 142
 EOEGREC CShapes : 874
 ERR CParse : 763
 Error checking CAppl : 280–CAppl : 283
 ErrorCheckingFlags CAppl : 282
 Evaluator (parse library) CParse : 759
 Event queues CAppl : 260–CAppl : 261,
 CMultit : 917–CMultit : 918, CMultit : 922
 @call and @send flags CCoding : 221
 accessing CMultit : 923
 handles CCoding : 145
 event-driven CGetSta : 128
 EXACT CParse : 763
 EXP CParse : 763
 Expanded memory CMemory : 544
 Exporting files: see impex library
 Extended memory CMemory : 544

F

FA_... CFile : 648–CFile : 649
 FACT CParse : 763
 FAF_PUSH_RESULT CMath : 952
 FAF_STORE_NUMBER CMath : 952
 FALSE CCoding : 142
 parse library internal function CParse : 763
 FCF_... CFile : 664
 FDAT_... CFile : 647–CFile : 648
 FDATExtract...() macros CFile : 648
 FEA_CREATION CFile : 651
 FEA_CREATOR CFile : 650
 FEA_CUSTOM CFile : 647
 FEA_DRIVE_STATUS CFile : 652
 FEA_FILE_ATTR CFile : 648
 FEA_FILE_TYPE CFile : 649
 FEA_FLAGS CFile : 649
 FEA_MODIFICATION CFile : 647
 FEA_MULTIPLE CFile : 646
 FEA_NAME CFile : 651
 FEA_NOTICE CFile : 651
 FEA_PASSWORD CFile : 651
 FEA_PROTOCOL CAppl : 262, CFile : 650
 FEA_RELEASE CAppl : 261, CFile : 650
 FEA_SIZE CFile : 649

FEA_TOKEN CFile : 650
 FEA_USER_NOTES CFile : 651
 FEAD_... CFile : 646–CFile : 647
 FEDosInfo CFile : 660
 FEF_... CShapes : 880
 FEP_... CFile : 656–CFile : 658
 FESF_... CFile : 658–CFile : 659
 FESRT_... CFile : 659–CFile : 660
 File position CFile : 666
 File system CFile : 611–CFile : 669
 driver standard path CFile : 635
 File system drivers CArch : 102
 File transfer
 PCCom library 795
 FILE_ACCESS_... CFile : 662
 FILE_DENY_... CFile : 662–CFile : 663
 FILE_POS_... CFile : 667–CFile : 668
 FileAccessFlags CFile : 662
 FileAttrs CFile : 648
 FileClose() CFile : 665
 FileCommit() CFile : 667
 FileContstructFullPath() CFile : 637
 FileCopy() CFile : 653
 FileCreate() CFile : 664
 FileCreateDir() CFile : 639
 FileCreateFlags CFile : 664
 FileCreateTempFile() CFile : 665
 FileDateAndTime CFile : 647
 Printing CLocal : 346
 FileDelete() CFile : 653
 FileDeleteDir() CFile : 640
 FileDuplicateHandle() CFile : 643
 FileEnum() CFile : 655
 FileEnumParams CFile : 656
 FileEnumSearchFlags CFile : 658
 FileEnumStandardReturnType CFile : 659
 FileEnumWildcard() CFile : 661
 FileExtAttrDesc CFile : 646
 FileGetAttributes() CFile : 669
 FileGetCurrentPath() CFile : 636
 FileGetDateAndTime() CFile : 668
 FileGetDiskHandle() CFile : 654
 FileGetHandleExtAttributes() CFile : 644
 FileGetPathExtAttributes() CFile : 645
 FileHandle CCoding : 145
 FileLockRecord() CFile : 669
 FileMove() CFile : 654
 FileOpen() CFile : 662
 FileParseStandardPath() CFile : 637
 FilePopDir() CFile : 638
 FilePos() CFile : 667
 FilePosMode CFile : 667
 FilePushDir() CFile : 638
 FileRead() CFile : 666
 FileRename() CFile : 653
 FileResolveStandardPath() CFile : 638



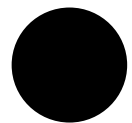
FileResolveStandardPathFlags CFile : 638
 Files CFile : 611–CFile : 669,
 CVM : 673–CVM : 715
 Formatting Date Strings CLocal : 346
 handles CCoding : 144
 FileSetAttributes() CFile : 669
 FileSetCurrentPath() CFile : 636
 FileSetDateAndTime() CFile : 668
 FileSetHandleExtAttributes() CFile : 645
 FileSetPathExtAttributes() CFile : 645
 FileUnlockRecord() CFile : 669
 FileWrite() CFile : 666
 FIND CParse : 763
 Fixed memory CMemory : 547
 Fixed point numbers CCoding : 147
 see also WWFixed, DWFixed, BBFixed,
 WBFixed
 Flag records CCoding : 142
 Float CMath : 960
 FLOAT_EXPONENT CMath : 946
 FloatAdd() CMath : 946
 FloatAsciiToFloat() CMath : 952
 FloatComp() CMath : 971
 FloatCompAndDrop() CMath : 971
 FloatCompESD1() CMath : 971
 FloatDateNumberGetMonthAndDay()
 CMath : 958
 FloatDateNumberGetWeekday() CMath : 959
 FloatDateNumberGetYear() CMath : 958
 FloatDepth() CMath : 967
 FloatDrop() CMath : 969
 FloatDup() CMath : 969
 FloatExit() CMath : 966
 FloatFloatIEEE64ToAscii_StdFormat()
 CMath : 952
 FloatFloatToAscii() CMath : 952
 FloatFloatToAscii_StdFormat() CMath : 952
 FloatFloatToAsciiFormatFlags CMath : 955
 FloatFloatToDateTimeData CMath : 956
 FloatFormatNumber() CMath : 959
 FloatGetDateNumber() CMath : 958
 FloatGetDaysInMonth() CMath : 959
 FloatGetNumDigitsInIntegerPart() CMath : 959
 FloatGetStackPointer() CMath : 971
 FloatGetTimeNumber() CMath : 959
 Floating-Point Formats CMath : 960
 FloatInit() CMath : 966
 FloatNum CMath : 946
 FloatOver() CMath : 969
 FloatPick() CMath : 969
 FloatPopNumber() CMath : 946, CMath : 967
 FloatPushNumber() CMath : 946, CMath : 967
 FloatRandom() CMath : 950
 FloatRandomize() CMath : 950
 FloatRandomN() CMath : 950
 FloatRoll() CMath : 968

FloatRollDown() CMath : 968
 FloatRot() CMath : 968
 FloatSetStackPointer() CMath : 971
 FloatSwap() CMath : 968
 FloatTimeNumberGetHour() CMath : 959
 FloatTimeNumberGetMinutes() CMath : 959
 FloatTimeNumberGetSeconds() CMath : 959
 FMT_... CFile : 628
 Focus CInput : 423, CInput : 453–CInput : 456
 Font drivers CArch : 103
 FontEnumFlags CShapes : 880
 Fonts
 driver standard path CFile : 635
 kernel graphics routines CShapes : 879
 metrics CShapes : 882
 FontWeight CShapes : 879
 FontWidth CShapes : 880
 forceQueue (message flag) CCoding : 221
 Format
 Floating-Point CMath : 960
 FormatArray CClipb : 304
 FormatEntry CMath : 963
 FormatError CFile : 628
 FormatExpression() CParse : 770
 FormatIdType CMath : 961
 FormatParams CMath : 960
 fptr CCoding : 154
 FRSPF_... CFile : 638
 FV CParse : 763

G

GA_... CAppl : 246–CAppl : 247
 GA_TARGETABLE CInput : 457
 GAGCNLT_SELF_LOAD_OPTIONS CAppl : 269
 GAGCNLT_STARTUP_LOAD_OPTIONS
 CAppl : 269
 GCN: see General change notification 21
 gcnList() keyword CCoding : 216
 GCNListAdd() CGCN : 357
 GCNListRemove() CGCN : 364
 GCNListSend() CGCN : 361
 GCNSLT_... CGCN : 358–CGCN : 360
 GCNStandardListType CGCN : 358
 GDDT_... CAppl : 256–CAppl : 257
 GE_... CSound : 515
 GenApplication CUIOver : 381
 GenBoolean CUIOver : 382
 GenBooleanGroup CUIOver : 382
 GenClass CUIOver : 381
 variant behavior CCoding : 171
 GenContent CUIOver : 385
 GenControl CUIOver : 384
 GenDisplay CUIOver : 384

GenDisplayControl CUIOver : 384
 GenDisplayGroup CUIOver : 384
 GenDocument CUIOver : 383
 GenDocumentControl CUIOver : 383
 GenDocumentGroup CUIOver : 383
 GenDynamicList CUIOver : 382
 GenEditControl CClipb : 314–CClipb : 316
 General change notification
 CGCN : 355–CGCN : 372
 goc syntax CCoding : 216–CCoding : 217
 Generic objects CUIOver : 378–CUIOver : 385
 GenFileSelector CUIOver : 385
 GenGlyph CUIOver : 385
 GenInteraction CUIOver : 382
 variant behavior CCoding : 173
 GenItem CUIOver : 382
 GenItemGroup CUIOver : 382
 GenPrimary CUIOver : 381
 GenProcess CGetSta : 109
 GenText CUIOver : 383
 GenToolControl CUIOver : 384
 GenTrigger CUIOver : 382
 instance data
 illustrated CCoding : 176
 variant behavior CCoding : 175–CCoding : 180
 GenValue CUIOver : 383
 GenView CGetSta : 111, CUIOver : 382
 GenViewInkType CInput : 444
 GeodeAllocQueue() CAppl : 261
 GeodeAttrs CAppl : 245
 GeodeDefaultDriverType CAppl : 256
 GeodeFind() CAppl : 259
 GeodeFindResource() CAppl : 259
 GeodeFlushQueue CAppl : 261
 GeodeFreeDriver() CAppl : 254
 GeodeFreeLibrary() CAppl : 253
 GeodeFreeQueue CAppl : 261
 GeodeGetAppObject() CAppl : 259
 GeodeGetCodeProcessHandle() CAppl : 260
 GeodeGetInfo() CAppl : 259
 GeodeGetInfoType CAppl : 259
 GeodeGetProcessHandle() CAppl : 260
 GeodeHandle CCoding : 145
 GeodeInfoDriver() CAppl : 254
 GeodeInfoQueue CAppl : 261
 GeodeLoad() CAppl : 247
 GeodePrivAlloc() CAppl : 264
 GeodePrivFree() CAppl : 265
 GeodePrivRead() CAppl : 264
 GeodePrivWrite() CAppl : 264
 Geodes CAppl : 242–CAppl : 265
 handles CCoding : 145, CAppl : 243
 GeodeSetDefaultDriver() CAppl : 257
 GeodeToken CAppl : 266
 GeodeUseDriver() CAppl : 254
 GeodeUseLibrary() CAppl : 253
 Geometry manager CGeom : 467–CGeom : 501
 GEOS.INI: see Initialization file
 GeosFileHeaderFlags CFile : 649
 GeosFileType CFile : 649
 GetPalType CShapes : 891
 GetPathType CGraph : 851
 GFHF_... CFile : 649–CFile : 650
 GFT_... CFile : 649
 GGIT_... CAppl : 260
 GIGT_OPTIONS_MENU CAppl : 269
 Glue
 externally declared chunks CCoding : 211
 see also Parameters files
 GPT_CLIP CGraph : 855
 GPT_CURRENT CGraph : 851
 GPT_WIN_CLIP CGraph : 855
 GraphicPattern CShapes : 891
 Graphics CGraph : 801–CGraph : 855,
 CShapes : 859–CShapes : 901
 Graphics strings: see GStrings
 GrApplyDWordTranslation() CGraph : 823
 GrApplyRotation() CGraph : 815
 GrApplyScale() CGraph : 815
 GrApplyTransform() CGraph : 817
 GrApplyTranslation() CGraph : 815
 GrBeginPath() CGraph : 851
 GrBeginUpdate() CGraph : 855
 GrBitBlit() CGraph : 853
 GrBrushPolyline() CShapes : 867
 GrCharMetrics() CShapes : 884
 GrCharWidth() CShapes : 882
 GrCheckFontAvail() CShapes : 881
 GrCheckFontAvailID() CShapes : 881
 GrCheckFontAvailName() CShapes : 881
 GrClearBitmap() CGraph : 830
 GrCloseSubPath() CGraph : 851
 GrComment() CGraph : 836
 GrCompactBitmap() CGraph : 831
 GrCopyGString() CGraph : 834
 GrCreateBitmap() CGraph : 829
 GrCreatePalette() CShapes : 890
 GrCreateState() CGraph : 827
 GrDeleteGStringElement() CGraph : 847
 GrDestroyBitmap() CGraph : 830
 GrDestroyGString() CGraph : 834
 GrDestroyPalette() CShapes : 890
 GrDestroyState() CGraph : 828
 GrDrawArc() CShapes : 863
 GrDrawArc3Point() CShapes : 864
 GrDrawArc3PointTo() CShapes : 864
 GrDrawBitmap() CShapes : 870
 GrDrawBitmapAtCP() CShapes : 871
 GrDrawChar() CShapes : 878
 GrDrawCharAtCP() CShapes : 878
 GrDrawCurve() CShapes : 868



GrDrawCurveTo() CShapes : 868
 GrDrawEllipse() CShapes : 862
 GrDrawGString() CGraph : 844
 GrDrawGStringAtCP() CGraph : 845
 GrDrawHLine() CShapes : 861
 GrDrawHLineTo() CShapes : 861
 GrDrawHugeBitmap() CShapes : 871
 GrDrawHugeBitmapAtCP() CShapes : 871
 GrDrawHugeImage() CShapes : 871
 GrDrawImage() CShapes : 871
 GrDrawLine() CShapes : 861
 GrDrawLineTo() CShapes : 861
 GrDrawPath() CShapes : 872
 GrDrawPoint() CShapes : 860
 GrDrawPointAtCP() CShapes : 860
 GrDrawPolygon() CShapes : 866
 GrDrawPolyline() CShapes : 866
 GrDrawRect() CShapes : 862
 GrDrawRectTo() CShapes : 862
 GrDrawRegion() CShapes : 874
 GrDrawRegionAtCP() CShapes : 874
 GrDrawRelArc3PointTo() CShapes : 864
 GrDrawRelLineTo() CShapes : 861
 GrDrawRoundRect() CShapes : 865
 GrDrawRoundRectTo() CShapes : 865
 GrDrawSpline() CShapes : 869
 GrDrawSplineTo() CShapes : 869
 GrDrawText() CShapes : 877
 GrDrawTextAtCP() CShapes : 878
 GrDrawTextField() CShapes : 878
 GrDrawVLine() CShapes : 861
 GrDrawVLineTo() CShapes : 861
 GrEditBitmap() CGraph : 830
 GrEditGString() CGraph : 834
 GrEndGString() CGraph : 835
 GrEndPath() CGraph : 851
 GrEndUpdate() CGraph : 856
 GrEnumFonts() CShapes : 880
 GrEscape() CGraph : 836
 GrFillArc() CShapes : 863
 GrFillArc3Point() CShapes : 864
 GrFillArc3PointTo() CShapes : 864
 GrFillBitmap() CShapes : 871
 GrFillBitmapAtCP() CShapes : 871
 GrFillEllipse() CShapes : 862
 GrFillHugeBitmap() CShapes : 871
 GrFillHugeBitmapAtCP() CShapes : 871
 GrFillPath() CShapes : 872
 GrFillPolygon() CShapes : 867
 GrFillRect() CShapes : 862
 GrFillRectTo() CShapes : 862
 GrFillRelArc3PointTo() CShapes : 864
 GrFillRoundRect() CShapes : 866
 GrFillRoundRectTo() CShapes : 866
 GrFindNearestPointsize() CShapes : 881
 GrFontMetrics() CShapes : 882

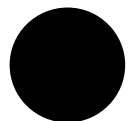
GrGetAreaColor() CShapes : 888
 GrGetAreaColorMap() CShapes : 889
 GrGetAreaMask() CShapes : 898
 GrGetAreaPattern() CShapes : 895
 GrGetBitmap() CGraph : 853
 GrGetBitmapMode() CGraph : 831
 GrGetBitmapRes() CGraph : 831
 GrGetBitmapSize() CGraph : 831
 GrGetClipRegion() CGraph : 855
 GrGetCurPos() CGraph : 825
 GrGetCurPosWWFixed() CGraph : 825
 GrGetDefFontID() CShapes : 881
 GrGetExclusive() CGraph : 852
 GrGetFont() CShapes : 879
 GrGetFontName() CShapes : 881
 GrGetFontWeight() CShapes : 879
 GrGetFontWidth() CShapes : 879
 GrGetGStringBounds() CGraph : 846
 GrGetGStringBoundsDWord() CGraph : 846
 GrGetGStringElement() CGraph : 847
 GrGetGStringHandle() CGraph : 834
 GrGetHugeBitmapSize() CGraph : 831
 GrGetLineColor() CShapes : 888
 GrGetLineColorMap() CShapes : 889
 GrGetLineEnd() CShapes : 901
 GrGetLineJoin() CShapes : 900
 GrGetLineMask() CShapes : 898
 GrGetLineStyle() CShapes : 900
 GrGetLineWidth() CShapes : 900
 GrGetMaskBounds() CGraph : 855
 GrGetMaskBoundsDWord() CGraph : 855
 GrGetMiterLimit() CShapes : 900
 GrGetMixMode() CShapes : 897
 GrGetPalette() CShapes : 891
 GrGetPath() CGraph : 852
 GrGetPathBounds() CGraph : 851
 GrGetPathBoundsDWord() CGraph : 851
 GrGetPathPoints() CGraph : 851
 GrGetPathRegion() CGraph : 851, CShapes : 874
 GrGetPoint() CGraph : 830, CGraph : 853
 GrGetPtrRegBounds() CShapes : 874
 GrGetSubscriptAttr() CShapes : 880
 GrGetSuperscriptAttr() CShapes : 880
 GrGetTextColor() CShapes : 888
 GrGetTextColorMap() CShapes : 889
 GrGetTextMask() CShapes : 898
 GrGetTextMode() CShapes : 878
 GrGetTextPattern() CShapes : 895
 GrGetTextSpacePad() CShapes : 878
 GrGetTextStyle() CShapes : 878
 GrGetTrackKern() CShapes : 879
 GrGetTransform() CGraph : 817
 GrGetWinBounds() CGraph : 855
 GrGetWinBoundsDWord() CGraph : 855
 GrGetWinHandle() CGraph : 853
 GrGrabExclusive() CGraph : 852

GrInitDefaultTransform() CGraph : 816
 GrInvalRect() CGraph : 852
 GrInvalRectDWord() CGraph : 852
 GrLabel() CGraph : 835
 GrLoadGString() CGraph : 834, CGraph : 843
 GrMapColorIndex() CShapes : 890
 GrMapColorRGB() CShapes : 890
 GrMoveReg() CShapes : 874
 GrMoveTo() CGraph : 825
 GrNewPage() CGraph : 835
 GrNullOp() CGraph : 836
 GrParseGString() CGraph : 848
 GrReleaseExclusive() CGraph : 852
 GrRelMoveTo() CGraph : 825
 GrRestoreState() CGraph : 828
 GrRestoreTransform() CGraph : 828
 GrSaveState() CGraph : 828
 GrSaveTransform() CGraph : 828
 GrSetAreaAttr() CShapes : 885
 GrSetAreaColor() CShapes : 887
 GrSetAreaColorMap() CShapes : 889
 GrSetAreaMaskCustom() CShapes : 898
 GrSetAreaMaskSys() CShapes : 898
 GrSetAreaPattern() CShapes : 892
 GrSetBitmapMode() CGraph : 830
 GrSetBitmapRes() CGraph : 831
 GrSetClipPath() CGraph : 855
 GrSetClipRect() CGraph : 855
 GrSetCustomAreaPattern() CShapes : 894
 GrSetCustomTextPattern() CShapes : 894
 GrSetDefaultState() CGraph : 828
 GrSetDefaultTransform() CGraph : 816
 GrSetFont() CShapes : 879
 GrSetFontWeight() CShapes : 879
 GrSetFontWidth() CShapes : 879
 GrSetGStringBounds() CGraph : 836
 GrSetGStringPos() CGraph : 845
 GrSetLineAttr() CShapes : 885, CShapes : 901
 GrSetLineColor() CShapes : 887
 GrSetLineColorMap() CShapes : 889
 GrSetLineEnd() CShapes : 901
 GrSetLineJoin() CShapes : 900
 GrSetLineMaskCustom() CShapes : 898
 GrSetLineMaskSys() CShapes : 898
 GrSetLineStyle() CShapes : 900
 GrSetLineWidth() CShapes : 900
 GrSetMiterLimit() CShapes : 900
 GrSetMixMode() CShapes : 897
 GrSetNullTransform() CGraph : 816
 GrSetPalette() CShapes : 890
 GrSetPaletteEntry() CShapes : 890
 GrSetStrokePath() CGraph : 851
 GrSetSuperScriptAttr() CShapes : 880
 GrSetTextAttr() CShapes : 885
 GrSetTextColor() CShapes : 888
 GrSetTextColorMap() CShapes : 889

GrSetTextMaskCustom() CShapes : 898
 GrSetTextMaskSys() CShapes : 898
 GrSetTextMode() CShapes : 878
 GrSetTextPattern() CShapes : 892
 GrSetTextSpacePad() CShapes : 878
 GrSetTextStyle() CShapes : 878
 GrSetTrackKern() CShapes : 879
 GrSetTransform() CGraph : 817
 GrSetVMFile()
 with bitmaps CGraph : 830
 with GStrings CGraph : 834
 GrSetWinClipPath() CGraph : 855
 GrSetWinClipRect() CGraph : 855
 GrTestPath() CGraph : 851
 clipping paths CGraph : 855
 GrTestPointInPath() CGraph : 851
 GrTestPointInPolygon() CShapes : 867
 GrTestPointInReg() CShapes : 874
 GrTestRectInMask() CGraph : 855
 GrTestRectInReg() CShapes : 874
 GrTextWidth() CShapes : 882
 GrTextWidthWBFixed() CShapes : 882
 GrTransform() CGraph : 822
 GrTransformByMatrix() CGraph : 818
 GrTransformDWFixed() CGraph : 824
 GrTransformDWord() CGraph : 824
 GrTransformWWFixed() CGraph : 822
 GrUncompactBitmap() CGraph : 831
 GrUntransform() CGraph : 822
 GrUntransformByMatrix() CGraph : 818
 GrUntransformDWFixed() CGraph : 824
 GrUntransformDWord() CGraph : 824
 GrUntransformWWFixed() CGraph : 822
 GSKT_... CGraph : 834
 GSRetType CGraph : 845
 GSRT_... CGraph : 845
 GSSPT_... CGraph : 845
 GST_... CGraph : 832
 GStateHandle CCoding : 145
 GStates CGraph : 825–CGraph : 828
 handles CCoding : 145
 GStringKillType CGraph : 834
 GStrings CGraph : 832–??
 GStringSetPosType CGraph : 845
 GStringType CGraph : 832
 GVIT_... CInput : 444–CInput : 445
 GWNT_PCCOM_DISPLAY_CHAR 796
 GWNT_PCCOM_DISPLAY_STRING 797
 GWNT_PCCOM_EXIT_PCCOM 797

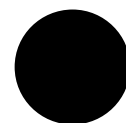
H

HAF_... CMemory : 550–CMemory : 551
 HALF CSound : 508



Handle CCoding : 145
 HandleModifyOwner() CMemory : 562
 HandleP() CMemory : 561
 Handles CCoding : 144–CCoding : 145,
 CHandle : 537–CHandle : 540
 types CCoding : 145
 VM block handles CVM : 676
 HandleV() CMemory : 561
 Handwriting Recognition
 library standard path CFile : 635
 HatchLine CShapes : 892
 Header files CGetSta : 107, CGetSta : 108
 goc library files CCoding : 209
 Heap CMemory : 545–CMemory : 552
 HeapAllocFlags CMemory : 550
 HeapFlags CMemory : 551
 Hello World sample application CGetSta : 108–??
 HF... CMemory : 551–CMemory : 552
 HIGH... CSound : 507
 HIGH... (Music Note Frequencies) CSound : 507
 HINT_ALIGN_BOTTOM_EDGE_WITH_OBJECT
 CGeom : 479
 HINT_ALIGN_LEFT_EDGE_WITH_OBJECT
 CGeom : 479
 HINT_ALIGN_LEFT_MONIKER_EDGE_WITH_C
 HILD CGeom : 489
 HINT_ALIGN_RIGHT_EDGE_WITH_OBJECT
 CGeom : 479
 HINT_ALIGN_TOP_EDGE_WITH_OBJECT
 CGeom : 479
 HINT_ALLOW_CHILDREN_TO_WRAP
 CGeom : 492
 HINT_AVOID_MENU_BAR CGeom : 496
 HINT_BOTTOM_JUSTIFY_CHILDREN
 CGeom : 478
 HINT_CENTER_CHILDREN_HORIZONTALLY
 CGeom : 479
 HINT_CENTER_CHILDREN_ON_MONIKERS
 CGeom : 489
 HINT_CENTER_CHILDREN_VERTICALLY
 CGeom : 479
 HINT_CENTER_MONIKER CGeom : 488
 HINT_CENTER_WINDOW CGeom : 499
 HINT_CUSTOM_CHILD_SPACING CGeom : 490
 HINT_CUSTOM_CHILD_SPACING_IF_LIMITED
 _SPACE CGeom : 491
 HINT_DEFAULT_FOCUS CInput : 454
 HINT_DEFAULT_MODEL CInput : 461
 HINT_DEFAULT_TARGET CInput : 457
 HINT_DIVIDE_HEIGHT_EQUALLY
 CGeom : 484
 HINT_DIVIDE_WIDTH_EQUALLY CGeom : 484
 HINT_DO_NOT_USE_MONIKER CGeom : 488
 HINT_DONT_ALLOW_CHILDREN_TO_WRAP
 CGeom : 492
 HINT_DONT_FULL_JUSTIFY_CHILDREN
 CGeom : 480
 HINT_DONT_INCLUDE_ENDS_IN_CHILD_SPAC
 ING CGeom : 480
 HINT_DONT_KEEP_INITALLY_ONSCREEN
 CGeom : 501
 HINT_DONT_KEEP_PARTIALLY_ONSCREEN
 CGeom : 501
 HINT_DRAW_IN_BOX CGeom : 487
 HINT_EXPAND_HEIGHT_TO_FIT_PARENT
 CGeom : 481
 HINT_EXPAND_WIDTH_TO_FIT_PARENT
 CGeom : 481
 HINT_EXTEND_WINDOW_NEAR_BOTTOM_RIG
 HT CGeom : 500
 HINT_EXTEND_WINDOW_TO_BOTTOM_RIGHT
 CGeom : 500
 HINT_FIXED_SIZE CGeom : 485
 HINT_FULL_JUSTIFY_CHILDREN_HORIZONTA
 LLY CGeom : 480
 HINT_FULL_JUSTIFY_CHILDREN_VERTICALL
 Y CGeom : 480
 HINT_INCLUDE_ENDS_IN_CHILD_SPACING
 CGeom : 480
 HINT_INITIAL_SIZE CGeom : 485
 HINT_KEEP_ENTIRELY_ONSCREEN
 CGeom : 502
 HINT_KEEP_ENTIRELY_ONSCREEN_WITH_MA
 RGIN CGeom : 502
 HINT_KEEP_INITIALLY_ONSCREEN
 CGeom : 501
 HINT_KEEP_PARTIALLY_ONSCREEN
 CGeom : 501
 HINT_LEFT_JUSTIFY_CHILDREN CGeom : 478
 HINT_LEFT_JUSTIFY_MONIKERS CGeom : 489
 HINT_MAKE_REPLY_BAR CGeom : 493
 HINT_MAXIMUM_SIZE CGeom : 485
 HINT_MINIMIZE_CHILD_SPACING
 CGeom : 492
 HINT_MINIMUM_SIZE CGeom : 486
 HINT_NO_BORDERS_ON_MONIKERS
 CGeom : 489
 HINT_NO_TALLER_THAN_CHILDREN_REQUIR
 E CGeom : 481
 HINT_NO_WIDER_THAN_CHILDREN_REQUIRE
 CGeom : 481
 HINT_ORIENT_CHILDREN_ALONG_LARGER_D
 IMENSION CGeom : 476
 HINT_ORIENT_CHILDREN_HORIZONTALLY
 CGeom : 474
 HINT_ORIENT_CHILDREN_VERTICALLY
 CGeom : 474
 HINT_PLACE_MONIKER_ABOVE CGeom : 488
 HINT_PLACE_MONIKER_ALONG_LARGER_DI
 MENSION CGeom : 488
 HINT_PLACE_MONIKER_BELOW CGeom : 488

HINT_PLACE_MONIKER_TO_LEFT
 CGeom : 488
 HINT_PLACE_MONIKER_TO_RIGHT
 CGeom : 488
 HINT_POPS_UP_BELOW CGeom : 497
 HINT_POPS_UP_TO_RIGHT CGeom : 497
 HINT_POSITION_WINDOW_AT_MOUSE
 CGeom : 499
 HINT_POSITION_WINDOW_AT_RATIO_OF_PARENT CGeom : 499
 HINT_RIGHT_JUSTIFY_CHILDREN
 CGeom : 478
 HINT_SAME_ORIENTATION_AS_PARENT
 CGeom : 475
 HINT_SEEK_BOTTOM_OF_VIEW CGeom : 496
 HINT_SEEK_LEFT_OF_VIEW CGeom : 496
 HINT_SEEK_MENU_BAR CGeom : 496
 HINT_SEEK_REPLY_BAR CGeom : 495
 HINT_SEEK_RIGHT_OF_VIEW CGeom : 496
 HINT_SEEK_TITLE_BAR_LEFT CGeom : 496
 HINT_SEEK_TITLE_BAR_RIGHT CGeom : 496
 HINT_SEEK_TOP_OF_VIEW CGeom : 496
 HINT_SEEK_X_SCROLLER_AREA CGeom : 496
 HINT_SEEK_Y_SCROLLER_AREA CGeom : 496
 HINT_SIZE_WINDOW_AS_DESIRED
 CGeom : 500
 HINT_SIZE_WINDOW_AS_RATIO_OF_FIELD
 CGeom : 501
 HINT_SIZE_WINDOW_AS_RATIO_OF_PARENT
 CGeom : 500
 HINT_STAGGER_WINDOW CGeom : 499
 HINT_TILE_WINDOW CGeom : 499
 HINT_TOP_JUSTIFY_CHILDREN CGeom : 478
 HINT_WINDOW_NO_CONSTRAINTS
 CGeom : 500
 HINT_WINDOW_NO_SYS_MENU CGeom : 502
 HINT_WINDOW_NO_TITLE_BAR CGeom : 502
 HINT_WRAP_AFTER_CHILD_COUNT
 CGeom : 492
 HINT_WRAP_AFTER_CHILD_COUNT_IF_VERTICAL_SCREEN CGeom : 492
 HLOOKUP CParse : 763
 Hot keys: see Keyboard accelerators
 Hot spot (mouse pointer) CInput : 434
 HugeArray CVM : 705–CVM : 715
 HugeArrayAppend() CVM : 711
 HugeArrayContract() CVM : 714
 HugeArrayCreate() CVM : 709
 HugeArrayDelete() CVM : 712
 HugeArrayDestroy() CVM : 709
 HugeArrayDirectory CVM : 706
 HugeArrayDirEntry CVM : 706
 HugeArrayDirty() CVM : 710
 HugeArrayEnum() CVM : 714
 HugeArrayExpand() CVM : 713
 HugeArrayGetCount() CVM : 712
 HugeArrayInsert() CVM : 711
 HugeArrayLock() CVM : 710
 HugeArrayNext() CVM : 713
 HugeArrayPrev() CVM : 713
 HugeArrayReplace() CVM : 712
 HugeArrayUnlock() CVM : 710
 IACP CAppl : 283
 IF CParse : 763
 IF... CShapes : 872
 ignoreDirty
 object flag CCoding : 213
 ImageFlags CShapes : 872
 impex library CArch : 101
 drivers standard path CFile : 635
 Importing files: see impex library
 INDEX CParse : 763
 INI file: see Initialization file
 InitFileCommit() CAppl : 274
 InitFileDeleteCategory() CAppl : 276
 InitFileDeleteEntry() CAppl : 276
 InitFileDeleteStringSection() CAppl : 276
 InitFileEnumStringSection() CAppl : 276
 InitFileGetTimeLastModified() CAppl : 274
 InitFileReadBoolean() CAppl : 275
 InitFileReadDataBlock() CAppl : 275
 InitFileReadDataBuffer() CAppl : 275
 InitFileReadInteger() CAppl : 275
 InitFileReadSectionBuffer() CAppl : 276
 InitFileReadStringBlock() CAppl : 276
 InitFileReadStringBuffer() CAppl : 276
 InitFileReadStringSectionBlock() CAppl : 276
 InitFileRevert() CAppl : 273
 InitFileSave() CAppl : 273
 InitFileWriteBoolean() CAppl : 275
 InitFileWriteData() CAppl : 274
 InitFileWriteInteger() CAppl : 275
 InitFileWriteString() CAppl : 274
 InitFileWriteStringSection() CAppl : 274
 Initialization file (GEOS.INI)
 CAppl : 271–CAppl : 277
 Ink input CInput : 442–CInput : 448
 InkDestinationInfo CInput : 446
 InkHeader CInput : 443
 Input manager CInput : 422–CInput : 463
 insertAtFront (message flag) CCoding : 221
 Instance data
 discardable CCoding : 169
 Goc syntax CCoding : 192–CCoding : 194
 master offsets CCoding : 167
 illustrated CCoding : 168
 naming conventions CGetSta : 113



- structure CCoding : 159–CCoding : 162,
 - CCoding : 168
 - illustrated CCoding : 176
- INT CParse : 763
- Inter-Application Communication (IACP)
 - CAppl : 283
- Internationalization: see Localization
- Inverse drawing CShapes : 896
 - mouse pointer image CInput : 435
- IRR CParse : 763
- ISERR CParse : 763
- ISNUMBER CParse : 763
- ISSTRING CParse : 763

K

- Keyboard accelerators
 - goc syntax CCoding : 217
- Keyboard drivers CArch : 102
 - standard path CFile : 635
- Keyboard input CInput : 437–CInput : 442

L

- Large document model
 - coordinates CGraph : 823
 - mouse events CInput : 433
- LARGEST_VISIBLE_ROW CDB : 732
- LE_... CShapes : 901
- LEFT CParse : 763
- LENGTH CParse : 763
- Libraries CArch : 101
 - @deflib in library header files CCoding : 209
 - geode attribute CAppl : 246
 - loading dynamically CAppl : 253
 - protocol numbers CAppl : 262
 - writing libraries CAppl : 257–CAppl : 258
 - see also Geodes
- LineEnd CShapes : 901
- LineJoin CShapes : 900
- Lines (graphics) CShapes : 861
- LineStyles CShapes : 900
- Linking files: see Glue
- LJ_... CShapes : 900
- LMEM_TYPE_... CLMem : 573–CLMem : 575
- LMemAlloc() CLMem : 579
- LMemContract() CLMem : 581
- LMemDeleteAt() CLMem : 580
- LMemDeleteAtHandles() CLMem : 581
- LMemDeref() CLMem : 579
- LMemFree() CLMem : 580
- LMemGetChunkSize() CLMem : 580
- LMemGetChunkSizeHandles() CLMem : 580

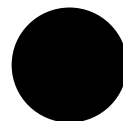
- LMemInitHeap() CLMem : 577
- LMemInsertAt() CLMem : 580
- LMemInsertAtHandles() CLMem : 580
- LMemReAlloc() CLMem : 580
- LMemReAllocHandles() CLMem : 580
- LMF_... CLMem : 575–CLMem : 576
- LMFreeHandles() CLMem : 580
- LN CParse : 763
- Local memory CLMem : 569–CLMem : 607
 - storing instance data CCoding : 159
 - see also Resources
- LOCAL_DISTANCE_BUFFER_SIZE CLocal : 342
- LocalAsciiToFixed() CLocal : 342
- LocalCalcDaysInMonth() CLocal : 346
- LocalCmpChars() CLocal : 348
- LocalCmpCharsNoCase() CLocal : 348
- LocalCmpStrings() CLocal : 348
- LocalCmpStringsDosToGeos() CLocal : 351
- LocalCmpStringsDosToGeosFlags CLocal : 351
- LocalCmpStringsNoCase() CLocal : 348
- LocalCmpStringsNoSpace() CLocal : 348
- LocalCmpStringsNoSpaceCase() CLocal : 348
- LocalCodePageToGeos() CLocal : 351
- LocalCodePageToGeosChar() CLocal : 351
- LocalCurrencyFormat CLocal : 343
- LocalCustomFormatDateTime() CLocal : 346
- LocalCustomParseDateTime() CLocal : 346
- LocalDistanceFromAscii() CLocal : 342
- LocalDistanceToAscii() CLocal : 342
- LocalDosToGeos() CLocal : 351
- LocalDosToGeosChar() CLocal : 351
- LocalDowncaseChar() CLocal : 349
- LocalDowncaseString() CLocal : 349
- LocalFixedToAscii() CLocal : 342
- LocalFormatDateTime() CLocal : 344
- LocalFormatFileDateTime() CLocal : 346
- LocalGeosToCodePage() CLocal : 351
- LocalGeosToCodePageChar() CLocal : 351
- LocalGeosToDos() CLocal : 351
- LocalGeosToDosChar() CLocal : 351
- LocalGetCodePage() CLocal : 351
- LocalGetCurrencyFormat() CLocal : 343
- LocalGetDateTimeFormat() CLocal : 346
- LocalGetLanguage() CLocal : 348
- LocalGetMeasurementType() CLocal : 342
- LocalGetNumericFormat() CLocal : 341,
 - CLocal : 343
- LocalGetQuotes() CLocal : 344
- LocalIsAlpha() CLocal : 349
- LocalIsAlphaNumeric() CLocal : 349
- LocalIsCodePageSupported() CLocal : 351
- LocalIsControl() CLocal : 349
- LocalIsDateChar() CLocal : 347
- LocalIsDigit() CLocal : 349
- LocalIsDosChar() CLocal : 351
- LocalIsGraphic() CLocal : 349

LocalIsHexDigit() CLocal : 349
 LocalIsLower() CLocal : 349
 LocalIsNumChar() CLocal : 347
 LocalIsPrintable() CLocal : 349
 LocalIsPunctuation() CLocal : 349
 LocalIsSpace() CLocal : 349
 LocalIsSymbol() CLocal : 349
 LocalIsTimeChar() CLocal : 347
 LocalIsUpper() CLocal : 349
 Localization CLocal : 335
 LocalLexicalValue() CLocal : 350
 LocalLexicalValueNoCase() CLocal : 350
 LocalMemoryFlags CMem : 575
 LocalNumericFormat CLocal : 342
 LocalParseDate() CLocal : 344
 LocalQuotes CLocal : 344
 LocalSetCurrencyFormat() CLocal : 343
 LocalSetDateTimeFormat() CLocal : 346
 LocalSetMeasurementType() CLocal : 342
 LocalSetNumericFormat() CLocal : 341
 LocalSetQuotes() CLocal : 344
 LocalStringLength() CLocal : 349
 LocalStringSize() CLocal : 349
 LocalUppcaseChar() CLocal : 349
 LocalUppcaseString() CLocal : 349
 LOG CParse : 764
 LOW_... CSound : 507
 LOW_... (Music Note Frequencies) CSound : 507
 LOWER CParse : 764
 LS_... CShapes : 900

M

macros CCoding : 151
 MakeWWFixed() macro CCoding : 148
 malloc() CMemory : 565
 Manufacturer ID
 in file CFile : 650
 master (class flag) CCoding : 185
 Master classes CCoding : 163–CCoding : 164
 class flag CCoding : 169, CCoding : 185
 in example CCoding : 175–CCoding : 180
 instance data groups
 illustrated CCoding : 161
 offset CCoding : 167
 Master groups CCoding : 160–CCoding : 162
 Math CArch : 100
 Algebraic Routines CMath : 972
 Comparison Routines CMath : 974
 Conversion Routines CMath : 977
 Exponential Routines CMath : 976
 Fractions and Integers CMath : 974
 Functions, Algebraic CMath : 947
 Functions, Transcendental CMath : 949

Random Numbers CMath : 975
 Trigonometric Routines CMath : 976
 Math Library CMath : 945
 MAX CParse : 764
 MAX_DAY_LENGTH CLocal : 346
 MAX_FONTS CShapes : 881
 MAX_KERN_VALUE CShapes : 879
 MAX_MONTH_LENGTH CLocal : 346
 MAX_POINT_SIZE CShapes : 879
 MAX_SEPARATOR_LENGTH CLocal : 346
 MAX_TRACK_KERNING CShapes : 879
 MAX_WEEKDAY_LENGTH CLocal : 346
 MAX_YEAR_LENGTH CLocal : 346
 MEASURE_METRIC CLocal : 342
 MEASURE_US CLocal : 342
 Measurements
 formatting CLocal : 341
 MeasurementType CLocal : 342
 MEDIA_... CFile : 618–CFile : 619
 MediaType CFile : 618
 MemAlloc() CMemory : 554
 MemAllocLMem() CMem : 578
 MemAllocSetOwner() CMemory : 554
 MemDecRefCount() CMemory : 564
 MemDeref() CMemory : 561
 MemDowngradeExclLock() CMemory : 560
 MemFree() CMemory : 555
 MemGetInfo() CMemory : 562
 MemHandle CCoding : 145
 MemIncRefCount() CMemory : 564
 MemInitRefCount() CMemory : 564
 MemLock() CMemory : 556
 MemLockExcl() CMemory : 559
 MemLockShared() CMemory : 559
 MemModifyFlags() CMemory : 562
 MemModifyOtherInfo() CMemory : 563
 Memory CMemory : 543–??
 80x86 addressing CHardw : 906
 handles CCoding : 144
 see also Local memory, Virtual memory, Heap,
 Handles, Database
 MemPLock() CMemory : 561
 MemReAlloc() CMemory : 554
 MemThreadGrab() CMemory : 558
 MemThreadGrabNB() CMemory : 559
 MemThreadRelease() CMemory : 559
 MemUnlock() CMemory : 556
 MemUnlockExcl() CMemory : 560
 MemUnlockShared() CMemory : 559
 MemUnlockV() CMemory : 561
 MemUpgradeSharedLock() CMemory : 560
 Menus CGetSta : 111
 message (method parameter) CCoding : 206
 Messages CCoding : 180–CCoding : 182
 calling CCoding : 157, CCoding : 180
 dispatcher CCoding : 157



exporting CCoding : 189–CCoding : 190
 handles CCoding : 145
 handling with variant classes CCoding : 179
 importing CCoding : 189–CCoding : 190
 naming conventions CGetSta : 113
 ranges CCoding : 187
 reserving ranges CCoding : 190
 sending CCoding : 181
 Metafile
 graphics CGraph : 804
 METAL_NOISE CSound : 522
 Method table CCoding : 179
 Methods
 goc syntax CCoding : 204–CCoding : 207
 storage CCoding : 170
 MGIT_... CMemory : 562
 MID CParse : 764
 MIDDLE_... CSound : 507
 MIDDLE_... (Music Note Frequencies)
 CSound : 507
 MIN CParse : 764
 MIN_KERN_VALUE CShapes : 879
 MIN_MAP_CHAR CLocal : 351
 MIN_POINT_SIZE CShapes : 879
 MIN_TRACK_KERNING CShapes : 879
 MixMode CShapes : 895
 MM_... CShapes : 895–CShapes : 897
 MOD CParse : 764
 Model CInput : 424, CInput : 460–CInput : 462
 Money
 currency formats CLocal : 343
 Monikers
 localizing CLocal : 338
 Mouse CInput : 424–CInput : 437
 driver standard path CFile : 634
 drivers CArch : 102
 grabbing CInput : 431–CInput : 433
 pointer image CInput : 434–CInput : 437
 MouseReturnFlags CInput : 430
 MouseReturnParams CInput : 430
 Moveable memory CMemory : 547
 MRF_... CInput : 430–CInput : 431
 MSG_GEN_APPLICATION_INK_QUERY_REPLY
 CInput : 445
 MSG_GEN_COPY_TREE CCoding : 230
 MSG_GEN_PROCESS_OPEN_APPLICATION
 CAppl : 248
 MSG_META_ACK CCoding : 236
 MSG_META_ADD_VAR_DATA CCoding : 200
 MSG_META_CLIPBOARD_COPY
 CClipb : 316–CClipb : 319
 MSG_META_CLIPBOARD_CUT
 CClipb : 316–CClipb : 319
 MSG_META_CLIPBOARD_NOTIFY_NORMAL_T
 TRANSFER_ITEM_CHANGED
 CClipb : 313
 MSG_META_CLIPBOARD_NOTIFY_QUICK_TRA
 NSFER_CONCLUDED CClipb : 330
 MSG_META_CLIPBOARD_NOTIFY_QUICK_TRA
 NSFER_FEEDBACK CClipb : 328
 MSG_META_CLIPBOARD_PASTE
 CClipb : 319–CClipb : 321
 MSG_META_CLIPBOARD_UNDO CClipb : 322
 MSG_META_DELETE_VAR_DATA
 CCoding : 200
 MSG_META_DETACH CCoding : 235
 MSG_META_DETACH_COMPLETE
 CCoding : 236
 MSG_META_DRAG_... CInput : 426–CInput : 428
 MSG_META_END_... CInput : 425–CInput : 427
 MSG_META_END_MOVE_COPY
 signalling quick-transfer CClipb : 330
 MSG_META_FINAL_OBJ_FREE CCoding : 237
 MSG_META_GAINED_FOCUS_EXCL
 CInput : 455
 MSG_META_GAINED_MODEL_EXCL
 CInput : 462
 MSG_META_GAINED_TARGET_EXCL
 CInput : 458
 MSG_META_GCN_LIST_ADD CGCN : 366
 MSG_META_GCN_LIST_REMOVE CGCN : 371
 MSG_META_GET_FOCUS_EXCL CInput : 455
 MSG_META_GET_MODEL_EXCL CInput : 461
 MSG_META_GET_TARGET_EXCL CInput : 458
 MSG_META_GET_VAR_DATA CCoding : 200
 MSG_META_GRAB_FOCUS_EXCL CInput : 454
 MSG_META_GRAB_MODEL_EXCL CInput : 461
 MSG_META_GRAB_TARGET_EXCL
 CInput : 458
 MSG_META_INITIALIZE_VAR_DATA
 CCoding : 200
 MSG_META_KBD_CHAR
 CInput : 439–CInput : 442
 MSG_META_LARGE_...
 CInput : 426–CInput : 428
 MSG_META_LOST_FOCUS_EXCL CInput : 455
 MSG_META_LOST_MODEL_EXCL CInput : 462
 MSG_META_LOST_TARGET_EXCL CInput : 458
 MSG_META_NOTIFY CGCN : 368
 With PCom library 796
 MSG_META_NOTIFY_APP_EXITED CGCN : 363
 MSG_META_NOTIFY_APP_STARTED
 CGCN : 363
 MSG_META_NOTIFY_DRIVE_CHANGE
 CGCN : 363
 MSG_META_NOTIFY_FILE_CHANGE
 CGCN : 361
 MSG_META_NOTIFY_USER_DICT_CHANGE
 CGCN : 363
 MSG_META_NOTIFY_WITH_DATA_BLOCK
 CGCN : 368
 With PCom library 797

MSG_META_OBJ_FLUSH_INPUT_QUEUE
 CCoding : 237
 MSG_META_OBJ_FREE CCoding : 236
 MSG_META_POST_PASSIVE_... (mouse events)
 CInput : 432-CInput : 433
 MSG_META_POST_PASSIVE_KBD_CHAR
 CInput : 439
 MSG_META_PRE_PASSIVE_... (mouse events)
 CInput : 432-CInput : 433
 MSG_META_PRE_PASSIVE_KBD_CHAR
 CInput : 439
 MSG_META_PTR CInput : 427
 MSG_META_QUERY_IF_PRESS_IS_INK
 CInput : 444
 MSG_META_RELEASE_FOCUS_EXCL
 CInput : 454
 MSG_META_RELEASE_MODEL_EXCL
 CInput : 461
 MSG_META_RELEASE_TARGET_EXCL
 CInput : 458
 MSG_META_START_...
 CInput : 425-CInput : 427
 MSG_META_START_MOVE_COPY
 signalling quick-transfer CClpb : 327
 MSG_VIS_GRAB_MOUSE CInput : 432
 MSG_VIS_NOTIFY_GEOMETRY_VALID
 CGeom : 471
 MSG_VIS_QUERY_IF_OBJECT_HANDLES_INK
 CInput : 447
 MSG_VIS_RECALC_SIZE CGeom : 470
 MSG_VIS_RELEASE_MOUSE CInput : 432
 MSG_VIS_UPDATE_GEOMETRY CGeom : 470
 Multithreading CMultit : 913-CMultit : 931
 Musical notes CSound : 507

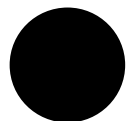
N

N CParse : 764
 NA CParse : 764
 Name arrays CLMem : 602-CLMem : 607
 NameArrayAdd() CLMem : 604
 NameArrayAddHandles() CLMem : 605
 NameArrayChangeName() CLMem : 606
 NameArrayChangeNameHandles() CLMem : 606
 NameArrayCreate() CLMem : 603
 NameArrayCreateAt() CLMem : 604
 NameArrayCreateAtHandles() CLMem : 604
 NameArrayFind() CLMem : 606
 NameArrayFindHandles() CLMem : 606
 NameArrayHeader CLMem : 603
 Naming conventions CGetSta : 112
 NEC macro CAppl : 281
 neverSaved (class flag) CCoding : 185
 NFF_LEADING_ZERO CLocal : 341

NO_NOISE CSound : 522
 @noreloc Goc keyword CCoding : 203
 notDetachable (@start flag) CCoding : 210
 NPV CParse : 764
 NULL CCoding : 145
 NullHandle CCoding : 145
 NumberFormatFlags CLocal : 341
 Numbers
 Floating-Point CMath : 965
 formatting CLocal : 341

O

ObjBlockGetOutput() CCoding : 231
 ObjBlockSetOutput() CCoding : 231
 ObjCompAddChild() CCoding : 233
 ObjCompFindChildByNumber() CCoding : 233
 ObjCompFindChildByOpnr() CCoding : 233
 ObjCompMoveChild() CCoding : 233
 ObjCompProcessChildren() CCoding : 233
 ObjCompRemoveChild() CCoding : 233
 ObjDecInUseCount() CCoding : 231
 ObjDispatchMessage() CAppl : 261
 ObjDoRelocation() CCoding : 232
 ObjDoUnRelocation() CCoding : 232
 ObjDuplicateResource() CCoding : 228
 Object blocks CCoding : 228-CCoding : 231
 Object oriented programming
 CArch : 68-CArch : 75
 Object pointer: see optr
 Objects CCoding : 158-CCoding : 182
 child objects CCoding : 156
 creating CCoding : 227-CCoding : 231
 destruction CCoding : 236-CCoding : 237
 detaching CCoding : 234-CCoding : 237
 goc syntax CCoding : 209-CCoding : 219,
 CCoding : 226-CCoding : 238
 naming conventions CGetSta : 113
 relocating CCoding : 170
 ObjEnableDetach() CCoding : 236
 ObjFreeDuplicate() CCoding : 231
 ObjFreeObjBlock() CCoding : 231
 ObjGetFlags() CCoding : 232
 ObjIncDetach() CCoding : 236
 ObjIncInUseCount() CCoding : 231
 ObjInitDetach() CCoding : 235
 ObjInitializeMaster() CCoding : 232
 ObjInitializePart() CCoding : 232
 ObjInstantiate() CCoding : 229
 ObjIsObjectInClass() CCoding : 232
 ObjLinkFindParent() CCoding : 233
 ObjLockObjBlock() CCoding : 231
 ObjResizeMaster() CCoding : 232
 ObjSetFlags() CCoding : 232



ObjTestIfObjBlockRunByCurThread()
 CCoding : 231
 ObjVarAddData() CCoding : 199
 ObjVarCopyDataRange() CCoding : 200
 ObjVarDeleteData() CCoding : 199
 ObjVarDeleteDataAt() CCoding : 199
 ObjVarDeleteDataRange() CCoding : 200
 ObjVarDerefData() CCoding : 199
 ObjVarFindData() CCoding : 199
 ObjVarScanData() CCoding : 199
 ODD_EVEN CShapes : 873
 OLButtonClass CCoding : 177
 ONE_HUNDRED_TWENTY_EIGHTH
 CSound : 508
 OOP: see Object oriented programming
 OP_... CParse : 749–CParse : 752
 Operands
 Floating-Point Routines CMath : 972
 OperatorType CParse : 748
 optr CCoding : 146, CCoding : 159
 OptrToChunk() macro CCoding : 146
 OptrToHandle() macro CCoding : 146
 OR CParse : 764
 oself CCoding : 206

P

PageEndCommand CGraph : 835
 Parallel port
 see also Streams
 Parallel ports CStream : 789
 Parameters files CGetSta : 107, CGetSta : 114,
 CAppl : 253–CAppl : 254
 parse library CParse : 743–CParse : 770
 PARSE_TOKEN_... CParse : 754–CParse : 756
 ParserEvalExpression() CParse : 767
 ParserParseString() CParse : 766
 ParserReturnStruct CParse : 766
 Passive grab
 keyboard events CInput : 439
 mouse events CInput : 432
 Paste: see Clipboard
 Paths (graphics) CGraph : 849–CGraph : 852,
 CShapes : 872
 PatternType CShapes : 891, CShapes : 892
 PCom library 795
 PCCOMABORT() 796
 PCCOMEXIT() 796
 PCCOMINIT() 795
 PEC_... CGraph : 835
 Pen input CInput : 442–CInput : 448
 Pen Input Drivers CArch : 102
 PI CParse : 764
 PMT CParse : 764

PointerDef CInput : 434
 Pointers CCoding : 144
 far pointers CCoding : 154
 Polygons CShapes : 866
 Polylines CShapes : 866
 Ports: see Streams
 Postpassive grab CInput : 432
 keyboard events CInput : 439
 Power management
 drivers standard path CFile : 635
 Power management drivers CArch : 102
 Prepassive grab CInput : 432
 keyboard events CInput : 439
 Printer drivers CArch : 102
 Printing CArch : 99
 drivers standard path CFile : 635
 PRIORITY_... CMultit : 924
 Private data CAppl : 264
 process (messaging shortcut) CCoding : 225
 Process object CGetSta : 109
 ProcessClass
 oself and pself CCoding : 206
 PRODUCT CParse : 764
 PROPER CParse : 764
 ProtocolNumber CAppl : 262
 Protocols
 geode CAppl : 261–CAppl : 264
 pself CCoding : 206
 PT_... CShapes : 891–CShapes : 892
 PT_SYSTEM_HATCH CShapes : 892
 PV CParse : 764

Q

QUARTER CSound : 508
 QueueGetMessage() CAppl : 261
 QueueHandle CCoding : 145
 Queues
 handles CCoding : 145
 Quick-transfer CClipb : 323–CClipb : 331
 Quotation marks
 localizing CLocal : 343

R

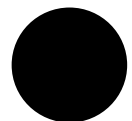
RANDOM CParse : 764
 RANDOMN CParse : 764
 RangeEnum() CDB : 740
 RangeExists() CDB : 739
 RangeInsert() CDB : 739
 RangeInsertParams CDB : 739
 RangeSort() CDB : 740
 RATE CParse : 764

realloc() CMemory : 565
 Rectangles CShapes : 862
 Region CShapes : 873
 RegionFillRule CShapes : 873
 Registers CHardw : 908
 Release number CAppl : 261–CAppl : 262
 in file CFile : 650
 ReleaseNumber CAppl : 261
 Relocatable data
 goc syntax CCoding : 202
 information in ClassStruct CCoding : 168
 kernel routines CCoding : 232
 VM operations CVM : 700
 REPEAT CParse : 764
 REPLACE CParse : 764
 replace (message flag) CCoding : 221
 Resources
 goc syntax CCoding : 210–CCoding : 212
 REST (sound library) CSound : 507
 RIGHT CParse : 764
 RIP... CDB : 740
 Roadmap to documentation CIntro : 38
 ROUND CParse : 764
 Routines
 Floating-Point Algebra CMath : 972
 Floating-Point Comparison CMath : 974
 Floating-Point Conversion CMath : 977
 Floating-Point Exponents CMath : 976
 Floating-Point Fractions and Integers
 CMath : 974
 Floating-Point Routines CMath : 972
 Floating-Point Trigonometry CMath : 976
 naming conventions CGetSta : 113
 Random Numbers CMath : 975
 ROWS CParse : 764
 ruler library CArch : 101

S

SBIEF... CSound : 528
 SBIEnvelopeFormat CSound : 528
 sbyte CCoding : 141
 SDM... CShapes : 899
 sdword CCoding : 141
 self (messaging shortcut) CCoding : 225
 Semaphores
 handles CCoding : 145
 threads CMultit : 926
 Serial port CStream : 782
 see also Streams
 SH... CShapes : 892
 Shared files
 VM files CVM : 697
 shift (keyboard accelerator modifier) CCoding : 217

ShiftState CInput : 439
 SIN CParse : 765
 SINH CParse : 765
 SIXTEENTH CSound : 508
 SIXTYFOURTH CSound : 508
 SLN CParse : 765
 SM... CStream : 784, CStream : 786,
 CStream : 787
 sound library CSound : 505
 parameters file CAppl : 258
 SoundAllocMusic() CSound : 516
 SoundAllocMusicNote() CSound : 510
 SoundAllocMusicStream() CSound : 518
 SoundAllocSampleStream() CSound : 519
 SoundDisableSampleStream() CSound : 519
 SoundEnableSampleStream() CSound : 519
 SoundFreeMusic() CSound : 517
 SoundFreeMusicNote() CSound : 510
 SoundFreeMusicStream() CSound : 518
 SoundFreeSampleStream() CSound : 520
 SoundGetExclusive() CSound : 520
 SoundGetExclusiveNB() CSound : 520
 SoundInitMusic() CSound : 516
 SoundPlayMusic() CSound : 516
 SoundPlayMusicNote() CSound : 510
 SoundPlayToMusicStream() CSound : 518
 SoundPlayToSampleStream() CSound : 519
 SoundReallocMusic() CSound : 517
 SoundReleaseExclusive() CSound : 521
 SoundStopMusic() CSound : 517
 SoundStopMusicNote() CSound : 510
 SoundStopMusicStream() CSound : 518
 SoundStreamDeltaTimeType CSound : 507
 SoundSynthDriverInfo() CSound : 521
 Source files CGetSta : 107
 SP... CSound : 511, CFile : 633–CFile : 636
 Specific user interface
 variant classes CCoding : 171–CCoding : 180
 SpecSizeSpec CGeom : 490
 SpecSizeTypes CGeom : 490
 Splines CShapes : 867
 Spooler CArch : 99
 Spreadsheets CArch : 101
 SQRT CParse : 765
 SS... CInput : 439–CInput : 440
 SSDTT... CSound : 507
 SSE... CSound : 513
 ssheet library CArch : 101
 SST... CGeom : 490–CGeom : 491
 Stack
 Floating-Point CMath : 966
 see also @stack
 Standard paths CFile : 632
 StandardSoundTypes CSound : 506
 State files CCoding : 157, CAppl : 250–CAppl : 253
 see also Relocatable data,



CLASSF_NEVER_SAVED
 STD CParse : 765
 STDP CParse : 765
 Streams CStream : 773, CStream : 773-??
 STRING CParse : 765
 Strings
 case CLocal : 349
 code pages CLocal : 350
 comparing CLocal : 348, CLocal : 350
 size CLocal : 349
 SUM CParse : 765
 Superclass
 sending messages to CCoding : 222
 Swap drivers
 standard path CFile : 635
 Swapable memory CMemory : 547
 sword CCoding : 141
 SYD CParse : 765
 SysDrawMask CShapes : 899
 SysGetConfig() CAppI : 279
 SysGetDosEnvironment() CAppI : 279
 SysGetECLevel() CAppI : 282
 SysGetInfo() CAppI : 279
 SysGetPenMode() CAppI : 279
 SysNotify() CAppI : 283
 SysSetECLevel() CAppI : 282
 SysShutdown() CAppI : 279
 SysStatistics() CAppI : 278
 SysStats CAppI : 278
 System architecture CArch : 64
 SystemDrawMask CShapes : 899
 SystemHatch CShapes : 892

T

TAN CParse : 765
 TANH CParse : 765
 Target CInput : 423, CInput : 456-CInput : 460
 Task switch drivers CArch : 102
 standard path CFile : 635
 TE... CMultit : 924
 TERM CParse : 765
 TestRectReturnType CShapes : 874
 Text
 objects CArch : 98-CArch : 99
 rendering with graphics CShapes : 876
 TextMode CShapes : 879
 TextTransferBlockHeader CClipb : 309
 TGIT... CMultit : 923
 THIRTYSECOND CSound : 508
 ThreadAllocSem() CMultit : 929
 ThreadAllocThreadLock() CMultit : 930
 ThreadAttachToQueue() CMultit : 922
 ThreadCreate() CMultit : 922

ThreadDestroy() CMultit : 925
 ThreadException CMultit : 924
 ThreadFreeSem() CMultit : 930
 ThreadFreeThreadLock() CMultit : 931
 ThreadGetInfo() CMultit : 923
 ThreadGetInfoType CMultit : 923
 ThreadGrabThreadLock() CMultit : 930
 ThreadHandle CCoding : 145
 ThreadHandleException() CMultit : 924
 ThreadModify() CMultit : 923
 ThreadPSem() CMultit : 929
 ThreadPTimedSem() CMultit : 930
 ThreadReleaseThreadLock() CMultit : 930
 Threads CArch : 75, CCoding : 155,
 CMultit : 913-CMultit : 931
 blocking CCoding : 157
 event queues CCoding : 180
 handles CCoding : 145
 lock handles CCoding : 145
 ThreadVSem() CMultit : 930
 TicTac sample application
 CUIOver : 391-CUIOver : 415
 Time
 formatting CLocal : 344
 Math Conversions CMath : 956
 TimerGetCount() CAppI : 278
 TimerGetDateAndTime() CAppI : 277
 TimerHandle CCoding : 145
 Timers CAppI : 277-CAppI : 278
 handles CCoding : 145
 TimerSetDateAndTime() CAppI : 277
 TimerSleep() CAppI : 278
 TimerStart() CAppI : 278
 TimerStop() CAppI : 278
 ToggleState CInput : 440
 TOKEN... (DateTimeFormat string tokens)
 CLocal : 347
 TOKEN_LENGTH CLocal : 346
 TokenDefineToken() CAppI : 268
 TokenGetTokenInfo() CAppI : 268
 TokenLoadMonikerBuffer() CAppI : 268
 TokenLoadMonikerChunk() CAppI : 268
 TokenLoadTokenBlock() CAppI : 267
 TokenLoadTokenBuffer() CAppI : 267
 TokenLoadTokenChunk() CAppI : 267
 TokenLockMonikerBlock() CAppI : 268
 TokenLockTokenMoniker() CAppI : 267
 TokenLookupMoniker() CAppI : 267
 TokenRemoveToken() CAppI : 268
 Tokens
 database CAppI : 265-CAppI : 268
 icons CAppI : 265-CAppI : 268
 TokenUnlockTokenMoniker() CAppI : 267
 Transfer VM file CClipb : 303
 TransferBlockID CClipb : 307
 TransMatrix CGraph : 816

TravelOption
 with input heirarchies CInput : 453
TRIM CParse : 765
TRUE CCoding : 142, CParse : 765
TRUNC CParse : 765
TS... CInput : 440

U

UIFA... CInput : 429–CInput : 430
UIFunctionsActive CInput : 429
UPPER CParse : 765
User interface CArch : 76–CArch : 90
User notes CFile : 651
UserCreateInkDestinationInfo() CInput : 446
UserLoadApplication() CApl : 248
UserStandardSound() CSound : 505
 single notes CSound : 510

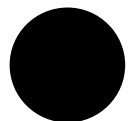
V

VALUE CParse : 765
VarDataCHandler tables CCoding : 201
VarDataFlags CCoding : 198
Variable data
 Goc syntax CCoding : 195–CCoding : 202
Variables
 naming conventions CGetSta : 113
variant (class flag) CCoding : 185
Variant classes CCoding : 156,
 CCoding : 171–CCoding : 180
 class flag CCoding : 169, CCoding : 185
 detecting CCoding : 166
 resolving CCoding : 156
VDF_EXTRA_DATA CCoding : 198
VDF_SAVE_TO_STATE CCoding : 198
VGA_NOTIFY_GEOMETRY_VALID CGeom : 471
Video drivers CGraph : 852–CGraph : 853
 standard path CFile : 635
Views CGetSta : 111
Virtual memory CVM : 673–CVM : 715
 handles CCoding : 144
VisClass CUIOver : 389
VisCompClass CUIOver : 389
VisContent CUIOver : 386
Visual monikers
 goc syntax CCoding : 215–CCoding : 216
Visual objects CUIOver : 385–CUIOver : 415
Visual upward queries CUIOver : 390
VMA... CVM : 681–CVM : 682
VMAccessFlags CVM : 685
VMAF... CVM : 685–CVM : 686
VMAlloc() CVM : 687

VMAllocLMem() CVM : 688
VMAttach() CVM : 689
VMAttributes CVM : 681
VMChain CVM : 700–CVM : 705
VMCHAIN_GET_VM_BLOCK() macro CVM : 705
VMCHAIN_IS_DBITEM() macro CVM : 705
VMCHAIN_MAKE_FROM_VM_BLOCK() macro
 CVM : 705
VMChainTree CVM : 702
VMClose() CVM : 695
VMCompareVMChains() CVM : 704
VMCopyVMChain() CVM : 704
VMCT... CVM : 702
VMDetach() CVM : 689
VMDirty() CVM : 690
VMFind() CVM : 691
VMFreeVMChain() CVM : 704
VMGetAttributes() CVM : 687
VMGetDirtyState() CVM : 695
VMGetMapBlock() CVM : 696
VMGrabExclusive() CVM : 698
VMInfo() CVM : 692
VMInfoStruct CVM : 692
VMLock() CVM : 688, CVM : 690, CVM : 691
VMMemBlockToVMBlock() CVM : 692
VMModifyUserID() CVM : 691
VMO... CVM : 684–CVM : 685, CVM : 699
VMOpen() CVM : 684
VMOpenTypes CVM : 684
VMOperations CVM : 699
VMPreserveBlocksHandle() CVM : 691
VMReleaseExclusive() CVM : 699
VMRevert() CVM : 694
VMSave() CVM : 694
VMSaveAs() CVM : 695
VMSetAttributes() CVM : 687
VMSetMapBlock() CVM : 696
VMSetReloc() CVM : 700
VMStartExclusiveReturnValue CVM : 698
VMUnlock() CVM : 690
VMUpdate() CVM : 693
VMVMBlockToMemBlock() CVM : 692
VOF_GEOMETRY_INVALID CGeom : 470
VOF_WINDOW_INVALID CGeom : 470
Volumes (file system) CFile : 614

W

WBFixed CCoding : 148
WHITE_NOISE CSound : 522
WHOLE CSound : 508
WINDING CShapes : 873
Windows CArch : 79–CArch : 80
 handles CCoding : 145



see also Clipping
word CCoding : 141
WordFlags CCoding : 142
WWFixed CCoding : 148
 Formatting as Strings CLocal : 342
WWFixedAsDWord CCoding : 148
WWFixedToFrac() macro CCoding : 148
WWFixedToInt() macro CCoding : 149

X

XMS CMemory : 544