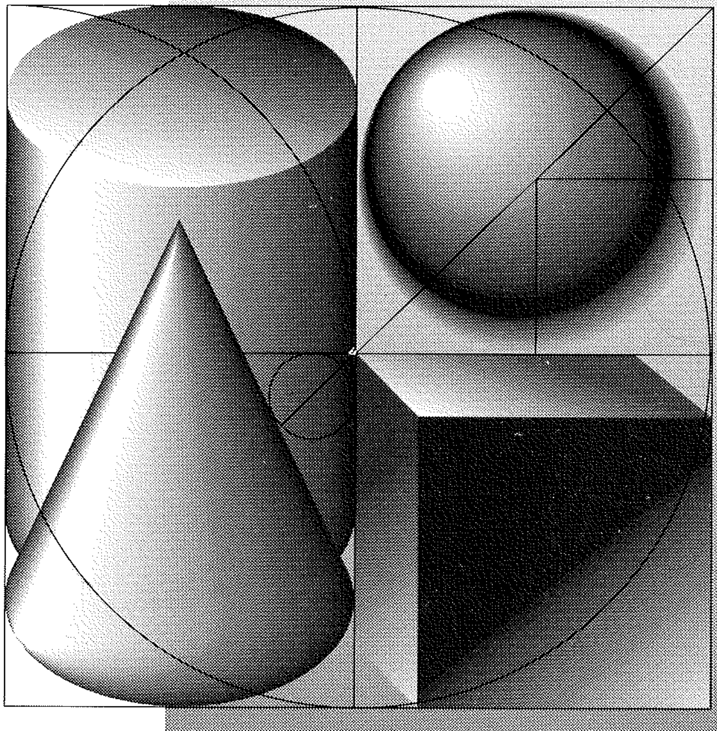
Apple IIGS® Toolbox Reference, Volume 3

Beta Draft

APDA™ # A0229LL/A



Apple IIGS® Toolbox Reference Volume 3

Beta Draft

This package contains:

1	Manual	*Apple IIGS Toolbox Reference, Volume 3*
	Set of Release Notes	*None*
	Disk	*None*
1	2-inch Binder Cover	
1	2-inch Spine Identification	

If you have any questions, please call:

1-800-282-2732 (U.S.)
1-408-562-3910 (International)
1-800-637-0029 (Canada)

Apple® IIGS® Toolbox Reference

Volume 3

Beta Draft

Developer Technical Publications

20525 Mariani Avenue
Cupertino, CA 95014-6299
(408) 996-1010

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manual or in the media on which a software product is distributed, APDA will replace the media or manual at no charge to you provided you return the item to be replaced with proof of purchase to APDA.

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Contents iii

Figures and tables xxii

Preface What's in this volume xxvii
Organization xxviii
Typographical conventions xxviii
Call format xxix
Indexes xxx

26 Apple Desktop Bus Tool Set Update 26-1
Error correction 26-2
Clarification 26-3

27 Audio Compression and Expansion Tool Set 27-1
Error correction 27-2
About Audio Compression and Expansion 27-2
Uses of the ACE Tool Set 27-3
How ADPCM works 27-4
ACE housekeeping routines 27-5
`ACEBootInit` $011D 27-5
`ACEStartUp` $021D 27-6
`ACEShutDown` $031D 27-7
`ACEVersion` $041D 27-8
`ACEReset` $051D 27-9
`ACEStatus` $061D 27-10
`ACEInfo` $071D 27-11

Audio Compression and Expansion tool calls 27-12
`ACECompBegin` $0B1D 27-12
`ACECompress` $091D 27-13
`ACEExpand` $0A1D 27-15
`ACEExpBegin` $0C1D 27-17
ACE Tool Set error codes 27-18

28 Control Manager Update 28-1
Error corrections 28-2
Clarifications 28-3
New features in the Control Manager 28-4
Keystroke processing in controls 28-4
The Control Manager and resources 28-5
New and changed controls 28-6
Simple button control 28-7
Check box control 28-7
Icon button control 28-7
LineEdit control 28-8
List control 28-9
Picture control 28-9
Pop-up control 28-10
Radio button control 28-11
Scroll bar control 28-11
Size box control 28-11
Static text control 28-11
TextEdit control 28-12
New control definition procedure messages 28-12
Initialize routine 28-13
Drag routine 28-13
Record size routine 28-13
Event routine 28-14
Target routine 28-15
Bounds routine 28-16
Window size routine 28-17
Tab routine 28-18
Notify multipart routine 28-19
Window change routine 28-20
New Control Manager calls 28-21
`CallCtlDefProc` $2C10 28-21

`CMLoadResource` $3210 28-23
`CMReleaseResource` $3310 28-24
`FindTargetCtl` $2610 28-25
`GetCtlHandleFromID` $3010 28-26
`GetCtlID` $2A10 28-27
`GetCtlMoreFlags` $2E10 28-28
`GetCtlParamPtr` $3510 28-29
`InvalCtls` $3710 28-30
`MakeNextCtlTarget` $2710 28-31
`MakeThisCtlTarget` $2810 28-32
`NewControl2` $3110 28-33
`NotifyCtls` $2D10 28-35
`SendEventToCtl` $2910 28-36
`SetCtlID` $2B10 28-38
`SetCtlMoreFlags` $2F10 28-39
`SetCtlParamPtr` $3410 28-40
Control Manager error codes 28-41
New Control Manager templates and records 28-42
`NewControl2` input templates 28-42
Control template standard header 28-43
Keystroke equivalent information 28-48
Simple button control template 28-49
Check box control template 28-51
Icon button control template 28-53
LineEdit control template 28-56
List control template 28-58
Picture control template 28-61
Pop-up control template 28-63
Radio button control template 28-69
Scroll bar control template 28-71
Size box control template 28-73
Static text control template 28-75
TextEdit control template 28-77
Control Manager code example 28-83
New control records 28-89
Generic extended control record 28-89
Extended simple button control record 28-95
Extended check box control record 28-98
Icon button control record 28-101
LineEdit control record 28-104

List control record 28-107
Picture control record 28-110
Pop-up control record 28-113
Extended radio button control record 28-118
Extended scroll bar control record 28-121
Extended size box control record 28-124
Static text control record 28-127
TextEdit control record 28-131

29 Desk Manager Update 29-1
New features in the Desk Manager 29-2
Scrollable CDA menu 29-2
Run queue 29-3
Run queue example 29-5
New Desk Manager calls 29-6
`AddToRunQ` $1F05 29-6
`RemoveFromRunQ` $2005 29-7
`RemoveCDA` $2105 29-8
`RemoveNDA` $2205 29-9

30 Dialog Manager Update 30-1
Error corrections 30-2

31 Event Manager Update 31-1
New features in Event Manager 31-2
Journaling changes 31-2
Keyboard input changes 31-3
New Event Manager calls 31-5
`GetKeyTranslation` $1B06 31-5
`SetAutoKeyLimit` $1A06 31-6
`SetKeyTranslation` $1C06 31-7

32 Font Manager Update 32-1
New features in the Font Manager 32-2
New call 32-3
`InstallWithStats` $1C1B 32-3

33 Integer Math Tool Set Update 33-1
Clarifications 33-2

34 LineEdit Tool Set Update 34-1
New features in LineEdit 34-2
New call 34-5
`GetLEDefProc` $2414 34-5

35 List Manager Update 35-1
Clarifications 35-2
List Manager definitions 35-2
New features in the List Manager 35-4
New List Manager calls 35-5
`DrawMember2` $111C 35-5
`NewList2` $161C 35-6
`NextMember2` $121C 35-8
`ResetMember2` $131C 35-9
`SelectMember2` $141C 35-10
`SortList2` $151C 35-11

36 Memory Manager Update 36-1
Error correction 36-2
Clarification 36-2
New features in the Memory Manager 36-2
Out-of-memory queue 36-2
Out-of-memory Routine example 36-5
New Memory Manager calls 36-8
`AddToOOMQueue` $0C02 36-8
`RealFreeMem` $2F02 36-9
`RemoveFromOOMQueue` $0D02 36-10

37 Menu Manager Update 37-1
Error corrections 37-2
Clarifications 37-2
New features in the Menu Manager 37-4

- Menu caching 37-6
- Caching with custom menus 37-7
- Pop-up menus 37-8
  - Pop-up menu scrolling options 37-10
  - How to use pop-up menus 37-12
- New Menu Manager data structures 37-15
  - Menu item template 37-15
  - Menu template 37-18
  - Menu bar template 37-20

New Menu Manager calls 37-21
- `GetPopUpDefProc` $3B0F 37-21
- `HideMenuBar` $450F 37-22
- `InsertMItem2` $3F0F 37-23
- `NewMenu2` $3E0F 37-24
- `NewMenuBar2` $430F 37-25
- `PopUpMenuSelect` $3C0F 37-27
- `SetMenuTitle2` $400F 37-29
- `SetMItem2` $410F 37-30
- `SetMItemName2` $420F 37-31
- `ShowMenuBar` $460F 37-32

38 MIDI Tool Set 38-1

About the MIDI Tool Set 38-2

Using the MIDI Tool Set 38-4
- Tool dependencies 38-7
- MIDI packet format 38-7
- MIDI Tool Set service routines 38-9
  - Real-time command routine 38-10
  - Real-time error routine 38-11
  - Input data routine 38-12
  - Output data routine 38-13
- Starting up the MIDI Tool Set 38-14
- Reading time-stamped MIDI data 38-16
- Fast access to MIDI Tool Set routines 38-20
- MIDI application considerations 38-22
  - MIDI and AppleTalk 38-22
  - Disabling interrupts 38-22
  - MIDI and other sound-related tool sets 38-23
  - The MIDI clock 38-23

Input and output buffer sizing 38-24
Loss of MIDI data 38-25
Number of MIDI interfaces 38-25
MIDI Housekeeping calls 38-26
`MidiBootInit` $0120 38-26
`MidiStartUp` $0220 38-27
`MidiShutDown` $0320 38-28
`MidiVersion` $0420 38-29
`MidiReset` $0520 38-30
`MidiStatus` $0620 38-31
MIDI tool calls 38-32
`MidiClock` $0B20 38-33
`MidiControl` $0920 38-36
`MidiDevice` $0A20 38-43
`MidiInfo` $0C20 38-46
`MidiReadPacket` $0D20 38-49
`MidiWritePacket` $0E20 38-51
MIDI Tool Set error codes 38-53

39 Miscellaneous Tool Set Update 39-1
Error corrections 39-2
New features in the Miscellaneous Tool Set 39-2
Queue handling 39-3
Interrupt state information 39-4
New Miscellaneous Tool Set calls 39-5
`AddToQueue` $2E03 39-5
`DeleteFromQueue` $2F03 39-6
`GetCodeResConverter` $3403 39-7
`GetInterruptState` $3103 39-8
`GetIntStateRecSize` $3203 39-9
`GetROMResource` $3503 39-9
`ReadMouse2` $3303 39-10
`ReleaseROMResource` $3603 39-10
`SetInterruptState` $3003 39-11

40 Note Sequencer 40-1
About the Note Sequencer 40-2
Using the Note Sequencer 40-3

Sequence timing 40-3
Using MIDI with the Note Sequencer 40-4
The Note Sequencer as a command interpreter 40-5
Error handlers and completion routines 40-6
Note commands 40-7
Filler notes 40-8
Filler note command 40-8
NoteOff command 40-9
NoteOn command 40-9
Control commands 40-10
CallRoutine command 40-11
Jump command 40-12
Pitch Bend command 40-13
Program Change command 40-14
Tempo command 40-14
Turn Notes Off command 40-15
Vibrato Depth command 40-15
Register commands 40-16
Dec Register command 40-17
IfGo Register command 40-17
Inc Register command 40-18
Set Register command 40-18
MIDI commands 40-19
MidiChannelPressure command 40-20
MidiControlChange command 40-20
MidiNoteOff command 40-20
MidiNoteOn command 40-21
MidiPitchBend command 40-21
MidiPolyphonicKeyPressure command 40-22
MidiProgramChange command 40-22
MidiSelectChannelMode command 40-22
MidiSetSysExlHighWord command 40-23
MidiSystemExclusive command 40-23
MidiSystemCommon command 40-24
MidiSystemRealTime command 40-25
Patterns and phrases 40-26
A sample Note Sequencer program 40-28
Note Sequencer housekeeping calls 40-37
`SeqBootInit` $011A 40-37
`SeqStartUp` $021A 40-38

`SeqShutDown` $031A 40-41
`SeqVersion` $041A 40-42
`SeqReset` $051A 40-43
`SeqStatus` $061A 40-44
Note Sequencer calls 40-45
`ClearIncr` $0A1A 40-45
`GetLoc` $0C1A 40-46
`GetTimer` $0B1A 40-47
`SeqAllNotesOff` $0D1A 40-48
`SetIncr` $091A 40-49
`SetInstTable` $121A 40-50
`SetTrkInfo` $0E1A 40-51
`StartInts` $131A 40-52
`StartSeq` $0F1A 40-53
`StartSeqRel` $151A 40-55
Sample sequence with relative addressing 40-58
`StepSeq` $101A 40-60
`StopInts` $141A 40-61
`StopSeq` $111A 40-62
Note Sequencer error codes 40-63

41 Note Synthesizer 41-1
About the Note Synthesizer 41-2
Using the Note Synthesizer 41-2
The sound envelope 41-2
Note Synthesizer envelopes 41-4
Instruments 41-6
DOC memory 41-10
Generators 41-10
Note Synthesizer housekeeping calls 41-14
`NSBootInit` $0119 41-14
`NSStartUp` $0219 41-15
`NSShutDown` $0319 41-16
`NSVersion` $0419 41-17
`NSReset` $0519 41-18
`NSStatus` $0619 41-19
Note Synthesizer calls 41-20
`AllNotesOff` $0D19 41-20
`AllocGen` $0919 41-21

`DeAllocGen` $0A19 41-22

`NoteOff` $0C19 41-23

`NoteOn` $0B19 41-24

`NSSetUpdateRate` $0E19 41-26

`NSSetUserUpdateRtn` $0F19 41-27

Note Synthesizer error codes 41-28

42 Print Manager Update 42-1

Error corrections 42-2

Clarifications 42-2

New features in the Print Manager 42-3

New Print Manager calls 42-4

`PMLoadDriver` $3513 42-4

`PMUnloadDriver` $3413 42-5

`PrGetDocName` $3613 42-6

`PrGetPgOrientation` $3813 42-7

`PrGetPrinterSpecs` $1813 42-8

`PrSetDocName` $3713 42-9

Previously undocumented Print Manager calls 42-10

`PrGetNetworkName` $2B13 42-10

`PrGetPortDvrName` $2913 42-11

`PrGetPrinterDvrName` $2813 42-12

`PrGetUserName` $2A13 42-13

`PrGetZoneName` $2513 42-14

Print Manager error codes 42-15

43 QuickDraw II Update 43-1

Error corrections 43-2

New features in QuickDraw II 43-3

QuickDraw II speed enhancement 43-3

New font header layout 43-5

44 QuickDraw II Auxiliary Update 44-1

New features in QuickDraw II Auxiliary 44-2

New QuickDraw II Auxiliary calls 44-3

`CalcMask` $0E12 44-3

`SeedFill` $0D12 44-8

SpecialRect $0C12 44-15

45 **Resource Manager** 45-1
About the Resource Manager 45-2
About resources 45-5
Identifying resources 45-5
Resource types 45-6
Resource IDs 45-6
Resource Names 45-7
Using resources 45-8
Resource attributes 45-9
Resource file format 45-11
Resource File IDs 45-11
Resource file search sequence 45-12
Resource file layout and data structures 45-13
Resource file header 45-15
Resource map 45-16
Resource free block 45-18
Resource reference record 45-19
Resource converter routines 45-20
ReadResource 45-21
WriteResource 45-23
ReturnDiskSize 45-25
Application switchers and desk accessories 45-26
Resource Manager housekeeping routines 45-28
ResourceBootInit $011E 45-28
ResourceStartUp $021E 45-29
ResourceShutDown $031E 45-30
ResourceVersion $041E 45-31
ResourceReset $051E 45-32
ResourceStatus $061E 45-33
Resource Manager tool calls 45-34
AddResource $0C1E 45-34
CloseResourceFile $0B1E 45-36
CountResources $221E 45-37
CountTypes $201E 45-38
CreateResourceFile $091E 45-39
DetachResource $181E 45-40
GetCurResourceApp $141E 45-41

GetCurResourceFile $121E 45-42
GetIndResource $231E 45-43
GetIndType $211E 45-45
GetMapHandle $261E 45-46
GetOpenFileRefNum $1F1E 45-48
GetResourceAttr $1B1E 45-50
GetResourceSize $1D1E 45-51
HomeResourceFile $151E 45-52
LoadAbsResource $271E 45-53
LoadResource $0E1E 45-55
MarkResourceChange $101E 45-57
MatchResourceHandle $1E1E 45-58
OpenResourceFile $0A1E 45-60
ReleaseResource $171E 45-62
RemoveResource $0F1E 45-63
ResourceConverter $281E 45-64
SetCurResourceApp $131E 45-66
SetCurResourceFile $111E 45-67
SetResourceAttr $1C1E 45-68
SetResourceFileDepth $251E 45-69
SetResourceID $1A1E 45-70
SetResourceLoad $241E 45-71
UniqueResourceID $191E 45-73
UpdateResourceFile $0D1E 45-75
WriteResource $161E 45-76
Resource Manager summary 45-77

46 Scheduler 46-1

47 Sound Tool Set Update 47-1
Error corrections 47-2
Clarification 47-3
FFStartSound 47-3
Moving a sound from the Macintosh to the Apple IIGS 47-5
Sample code 47-5
New information 47-6
Introduction to sound on the Apple IIGS 47-7
Note Sequencer 47-8

Note Synthesizer 47-8
Sound General Logic Unit (GLU) 47-9
Vocabulary 47-9
Oscillator 47-9
Generator 47-9
Voice 47-9
Sample rate 47-10
Drop sample tuning 47-10
Frequency 47-11
Sound RAM 47-11
Waveform 47-11
DOC registers 47-11
Frequency registers 47-12
Volume register 47-13
Waveform Data Sample register 47-13
Waveform Table Pointer register 47-13
Control register 47-13
Channel register 47-14
Bank-Select/Table-Size/Resolution register 47-14
Oscillator Interrupt register 47-16
Oscillator Enable register 47-16
A/D Converter register 47-16
MIDI and interrupts 47-17
New Sound Tool Set calls 47-18
`FFSetUpSound` $1508 47-18
`FFStartPlaying` $1608 47-20
`ReadDOCReg` $1808 47-21
`SetDOCReg` $1708 47-23

48 Standard File Operations Tool Set Update 48-1
New features in the Standard File Operations Tool Set 48-2
New filter procedure entry interface 48-4
Custom item drawing routines 48-5
Standard File data structures 48-6
Reply record 48-6
Multifile reply record 48-8
File type list record 48-9
Standard File dialog templates 48-12
Open File dialog box templates 48-13

Save File dialog box templates 48-19
New or changed Standard File calls 48-27
`SFAllCaps` $0D17 48-27
`SFGetFile2` $0E17 48-28
`SFMultiGet2` $1417 48-30
`SFPGetFile2` $1017 48-32
`SFPMultiGet2` $1517 48-34
`SFPPutFile2` $1117 48-36
`SFPutFile2` $0F17 48-39
`SFReScan` $1317 48-41
`SFShowInvisible` $1217 48-42
Standard File error codes 48-43

49 TextEdit Tool Set 49-1
About the TextEdit Tool Set 49-2
TextEdit call summary 49-4
How to use TextEdit 49-6
Standard TextEdit key sequences 49-11
Internal structure of the TextEdit Tool Set 49-14
TextEdit controls and the Control Manager 49-14
TextEdit filter procedures and hook routines 49-15
Generic filter procedure 49-16
`doEraseRect` ($0001) 49-17
`doEraseBuffer` ($0002) 49-18
`doRectChanged` ($0003) 49-19
Keystroke filter procedure 49-20
Word wrap hook 49-24
Word break hook 49-26
Custom scroll bars 49-28
TextEdit data structures 49-29
High-level TextEdit structures 49-30
`TEColorTable` 49-30
`TEFormat` 49-34
`TEParamBlock` 49-36
`TERuler` 49-44
`TEStyle` 49-46
Low-level TextEdit structures 49-47
`TERecord` 49-47
`KeyRecord` 49-59

StyleItem 49-62
SuperBlock 49-63
SuperHandle 49-64
SuperItem 49-65
TabItem 49-66
TextBlock 49-67
TextList 49-68
TextEdit housekeeping routines 49-69
TEBootInit $0122 49-69
TEStartUp $0222 49-70
TEShutDown $0322 49-71
TEVersion $0422 49-72
TEReset $0522 49-72
TEStatus $0622 49-73
TextEdit tool calls 49-74
TEActivate $0F22 49-74
TEClear $1922 49-75
TEClick $1122 49-76
TECompactRecord $1728 49-78
TECopy $1722 49-79
TECut $1622 49-80
TEDeactivate $1022 49-81
TEGetDefProc $2222 49-82
TEGetInternalProc $2622 49-83
TEGetLastError $2722 49-84
TEGetRuler $2322 49-85
TEGetSelection $1C22 49-87
TEGetSelectionStyle $1E22 49-88
TEGetText $0C22 49-91
TEGetTextInfo $0D22 49-95
TEIdle $0E22 49-98
TEInsert $1A22 49-99
TEKey $1422 49-102
TEKill $0A22 49-103
TENew $0922 49-104
TEOffsetToPoint $2022 49-106
TEPaintText $1322 49-108
TEPaste $1822 49-110
TEPointToOffset $2122 49-111
TEReplace $1B22 49-113

TEScroll $2522 49-116
TESetRuler $2422 49-118
TESetSelection $1D22 49-120
TESetText $0B22 49-121
TEStyleChange $1F22 49-124
TEUpdate $1222 49-127
TextEdit summary 49-128

50 Text Tool Set Update 50-1
New features in the Text Tool Set 50-2

51 Tool Locator Update 51-1
New features in the Tool Locator 51-2
Tool set startup and shutdown 51-2
Tool set numbers 51-6
Tool set dependencies 51-8
New Tool Locator calls 51-14
MessageByName $1701 51-14
SetDefaultTPT $1601 51-17
ShutDownTools $1901 51-18
StartUpTools $1801 51-19

52 Window Manager Update 52-1
Error corrections 52-2
New features in the Window Manager 52-3
Alert windows 52-5
Special characters 52-9
Alert window example 52-10
TaskMaster result codes 52-12
Window Manager data structures 52-14
Window record 52-14
Task record 52-17
New Window Manager calls 52-22
AlertWindow $590E 52-22
CompileText $600E 52-24
DrawInfoBar $550E 52-27
EndFrameDrawing $5B0E 52-28

- `ErrorWindow` $620E 52-29
- `GetWindowMgrGlobals` $580E 52-31
- `NewWindow2` $610E 52-32
- `ResizeWindow` $5C0E 52-35
- `StartFrameDrawing` $5A0E 52-36
- `TaskMasterContent` $5D0E 52-37
- `TaskMasterDA` $5F0E 52-39
- `TaskMasterKey` $5E0E 52-40
- `GDRPrivate` $540E 52-43

Error messages 52-44

E Resource Types E-1

- `rAlertString` $8015 E-2
- `rControlList` $8003 E-3
- `rControlTemplate` $8004 E-4
  - Control template standard header E-5
  - Keystroke equivalent information E-10
  - Simple button control template E-11
  - Check box control template E-13
  - Icon button control template E-15
  - LineEdit control template E-18
  - List control template E-20
  - Picture control template E-24
  - Pop-up control template E-26
  - Radio button control template E-30
  - Scroll bar control template E-32
  - Size box control template E-34
  - Static text control template E-36
  - TextEdit control template E-38
- `rC1InputString` $8005 E-43
- `rC1OutputString` $8023 E-44
- `rCString` $801D E-45
- `rIcon` $8001 E-46
- `rKTransTable` $8021 E-47
- `rListRef` $801C E-49
- `rMenu` $8009 E-50
- `rMenuBar` $8008 E-53
- `rMenuItem` $800A E-54
- `rPString` $8006 E-56

rResName $8014 E-57
rStringList $8007 E-59
rText $8016 E-60
rTextBlock $8011 E-61
rTextBox2 $800B E-62
rToolStartup $8013 E-63
rTwoRects $801A E-65
rWindColor $8010 E-66
rWindParam1 $800E E-68
rWindParam2 $800F E-72

F Delta Guide F-1
Apple Desktop Bus F-2
Error correction F-2
Clarification F-3
Control Manager F-4
Error corrections F-4
Clarifications F-5
Dialog Manager F-6
Error corrections F-6
Integer Math Tool Set F-7
Clarifications F-7
List Manager F-8
Clarifications F-8
List Manager definitions F-8
Memory Manager F-10
Error correction F-10
Clarification F-10
Menu Manager F-11
Error corrections F-11
Clarifications F-11
Miscellaneous Tool Set F-13
Error corrections F-13
Print Manager F-14
Error corrections F-14
Clarifications F-14
QuickDraw II F-15
Error corrections F-15
Sound Tool Set F-16

Error corrections F-16
Clarification F-16
`FFStartSound` F-17
Moving a sound from the Macintosh to the Apple IIGS F-18
Sample code F-19
Window Manager F-20
Error corrections F-20

Glossary GL-1

Index X-1

Figures and tables

Chapter 28 Control Manager Update 28-1

Figure 28-1 Control template standard header 28-43
Figure 28-2 Keystroke equivalent record layout 28-48
Figure 28-3 Item template for simple button controls 28-49
Figure 28-4 Control template for check box controls 28-51
Figure 28-5 Control template for icon button controls 28-53
Figure 28-6 Control template for LineEdit controls 28-56
Figure 28-7 Control template for list controls 28-58
Figure 28-8 Control template for picture controls 28-61
Figure 28-9 Control template for pop-up controls 28-63
Figure 28-10 "Unpopped" pop-up menu . 28-67
Figure 28-11 "Popped" pop-up menu, left-justified title 28-67
Figure 28-12 "Popped" pop-up menu, right-justified title 28-68
Figure 28-13 Control template for radio button controls 28-69
Figure 28-14 Control template for scroll bar controls 28-71
Figure 28-15 Control template for size box controls 28-73
Figure 28-16 Control template for static text controls 28-75
Figure 28-17 Control template for TextEdit controls 28-77
Figure 28-18 Generic extended control record 28-90
Figure 28-19 Extended simple button control record 28-96
Figure 28-20 Extended check box control record 28-99
Figure 28-21 Icon button control record 28-101
Figure 28-22 LineEdit control record 28-105
Figure 28-23 List control record 28-107
Figure 28-24 Picture control record 28-111
Figure 28-25 Pop-up control record 28-113
Figure 28-26 Extended radio button control record 28-119
Figure 28-27 Extended scroll bar control record 28-122
Figure 28-28 Extended size box control record 28-125
Figure 28-29 Static text control record 28-128
Figure 28-30 TextEdit control record 28-131

Chapter 34 LineEdit Tool Set Update 34-1
Figure 34-1 LineEdit Edit record (new layout) 34-3

Chapter 37 Menu Manager Update 37-1
Figure 37-1 Scrolling menus with indicator at bottom 37-5
Figure 37-2 Window with pop-up menus 37-9
Figure 37-3 Dragging through a pop-up menu 37-10
Figure 37-4 Type 1 pop-up menu 37-11
Figure 37-5 Type 2 pop-up menu 37-12
Figure 37-6 Menu item template 37-15
Figure 37-7 Menu template 37-18
Figure 37-8 Menu bar template 37-20
Table 37-1 Menu Manager calls that work with pop-up menus 37-13

Chapter 38 MIDI Tool Set 38-1
Figure 38-1 MIDI application environment 38-5

Chapter 39 Miscellaneous Tool Set Update 39-1
Figure 39-1 Queue header layout 39-3
Figure 39-2 Interrupt state record layout 39-4

Chapter 40 Note Sequencer 40-1
Figure 40-1 Format of a seqItem 40-5
Figure 40-2 Note command format 40-7
Figure 40-3 Control command format 40-10
Figure 40-4 Register command format 40-16
Figure 40-5 MIDI command format 40-19

Chapter 41 Note Synthesizer 41-1
Figure 41-1 Sound envelope, showing attack, decay, sustain, and release 41-3
Figure 41-2 Typical Note Synthesizer envelope 41-4
Figure 41-3 Instrument data structure 41-6
Figure 41-4 Generator control block 41-12

Chapter 43 QuickDraw II Update 43-1
Figure 43-1 New font header layout 43-5

Chapter 45 Resource Manager 45-1
Figure 45-1 Resource file internal layout 45-14
Table 45-1 Resource Manager constants 45-77
Table 45-2 Resource Manager data structures 45-78
Table 45-3 Resource Manager error codes 45-80

Chapter 47 Sound Tool Set Update 47-1
Figure 47-1 DOC registers 47-15

Chapter 48 Standard File Operations Tool Set Update 48-1
Figure 48-1 New-style reply record 48-6
Figure 48-2 Multifile reply record 48-8
Figure 48-3 File type list record 48-10

Chapter 49 TextEdit Tool Set 49-1
Figure 49-1 `TEColorTable` layout 49-31
Figure 49-2 `TEFormat` layout 49-34
Figure 49-3 `TEParamBlock` layout 49-37
Figure 49-4 `TERuler` layout 49-44
Figure 49-5 `TEStyle` layout 49-46
Figure 49-6 `TERecord` layout 49-48
Figure 49-7 `KeyRecord` layout 49-60
Figure 49-8 `StyleItem` layout 49-62
Figure 49-9 `SuperBlock` layout 49-63
Figure 49-10 `SuperHandle` layout 49-64
Figure 49-11 `SuperItem` layout 49-65
Figure 49-12 `TabItem` layout 49-66
Figure 49-13 `TextBlock` layout 49-67
Figure 49-14 `TextList` layout 49-68
Table 49-1 TextEdit constants 49-128
Table 49-2 TextEdit data structures 49-130
Table 49-3 TextEdit error codes 49-138

Chapter 51 Tool Locator Update 51-1

Figure 51-1 Tool set StartStop record 51-4
Table 51-1 Tool set numbers 51-6
Table 51-2 Tool set dependencies 51-8

Chapter 52 Window Manager Update 52-1

Figure 52-1 Window record definition 52-15
Figure 52-2 Task Record definition 52-18
Table 52-1 Standard alert window sizes 52-7
Table 52-2 Substitution string array 52-10
Table 52-3 TaskMaster result codes 52-12

Appendix E Resource Types E-1

Figure E–1 Alert string, type `rAlertString` ($8015) E-2
Figure E–2 Control List, type `rControlList` ($8003) E-3
Figure E-3 Control template standard header E-5
Figure E-4 Keystroke equivalent record layout E-10
Figure E-5 Item template for simple button controls E-11
Figure E-6 Control template for check box controls E-13
Figure E-7 Control template for icon button controls E-15
Figure E-8 Control template for LineEdit controls E-18
Figure E-9 Control template for list controls E-21
Figure E-10 Control template for picture controls E-24
Figure E-11 Control template for pop-up controls E-26
Figure E-12 Control template for radio button controls E-30
Figure E-13 Control template for scroll bar controls E-32
Figure E-14 Control template for size box controls E-34
Figure E-15 Control template for static text controls E-36
Figure E-16 Control template for TextEdit controls E-38
Figure E-17 GS/OS class1 input string, type `rC1InputString` ($8005) E-43
Figure E-18 GS/OS class1 output string, type `rC1OutputString` ($8023) E-44
Figure E-19 C string, type `rCString` ($801D) E-45
Figure E-20 Icon, type `rIcon` ($8001) E-46
Figure E–21 Keystroke translation table, type `rKTransTable` ($8021) E-47
Figure E–22 List member reference array element, type `rListRef` ($801C) E-49

Figure E-23 Menu template, type rMenu ($8009) E-50
Figure E-24 Menu bar record, type rMenuBar ($8008) E-53
Figure E-25 Menu item template, type rMenuItem ($800A) E-54
Figure E-26 Pascal string, type rPString ($8006) E-56
Figure E-27 Resource name array, type rResName ($8014) E-57
Figure E-28 Pascal string array, type rStringList ($8007) E-59
Figure E-29 Text block, type rText ($8016) E-60
Figure E-30 Text block, type rTextBlock ($8011) E-61
Figure E-31 LETextBox2 input text, type rTextBox2 ($800B) E-62
Figure E-32 Tool start stop record, type rToolStartup ($8013) E-63
Figure E-33 Two rectangles, type rTwoRects ($801A) E-65
Figure E-34 Window color table, type rWindColor ($8010) E-66
Figure E-35 Window template, type rWindParam1 ($800E) E-69
Figure E-36 Window template, type rWindParam2 ($800F) E-72

Preface **What's in this volume**

This third volume of the *Apple IIGS Toolbox Reference* contains new material describing numerous changes to the Apple IIGS® Toolbox. There are six previously undocumented tool sets, many new tool calls, and numerous corrections and additions. This document comprises both new material and information issued in a previous update that was available only from APDA™ (the Apple Programmers and Developers Association).

Organization

Like the first two volumes of the *Apple IIGS Toolbox Reference*, this book contains chapters that are devoted to individual tool sets or managers. The chapters are arranged alphabetically by tool set name. Chapters documenting the six new tool sets appear in alphabetical order among the other chapters. Chapters that discuss previously existing tool sets or managers carry the same titles as before, with the addition of the word *Update*. Note that chapters in this book have been numbered sequentially following the last chapter in Volume 2 of the *Toolbox Reference.*

Each chapter contains a brief introductory note, which indicates whether the chapter updates existing material or describes a new tool set or manager. Update chapters contain one or more of these sections:

Error corrections	Corrects errors in the previous toolbox documentation
Clarifications	Provides additional information about previously documented toolbox features, including bug fixes
New features	Describes new tool set features
New tool calls	Defines new tool calls

New chapters follow the organizational style of the first two volumes.

In addition to the chapters that discuss the various tool sets and managers, this book contains several appendixes:

Appendix E ("Resource Types")	Contains format and content information for all defined Apple IIGS resource types
Appendix F ("Delta Guide")	Collects all corrections to and clarifications of the previous volumes of the *Toolbox Reference* in a single location

Typographical conventions

This update largely obeys the typographical conventions of the *Apple IIGS Toolbox Reference.* New terms appear in **boldface** when they are introduced. Tool call parameter names are given in *italics.* Record field names, routine names, and code listings appear in the `Courier` font.

For the Beta draft, new or substantially changed text carries change bars.

Call format

This book documents tool calls for all the new tool sets and several of the existing tool sets.

Certain elements of this format may not appear in all calls. For example, not all calls return error codes, and so not all call descriptions contain a list of error codes. Similarly, stack diagrams are omitted from those calls that do not affect the stack.

`ToolCallName` $callnumber

A description of the call's function.

Parameters

Stack before call

Previous contents	
– *longParmName* –	Long—Description of *longParmName* parameter
wordParmName	Word—Description of *wordParmName* parameter
	<—SP

Stack after call

Previous contents	
– *Result* –	Long—Description of call result value (if any)
	<—SP

Errors	$xxxx	`Error name`	Description of the error code

C `The C language function declaration for the call.`

stackField Detailed description of stack input or output parameter, where appropriate.

Indexes

Volume 3 contains three indexes. First, there is an index of calls that are new in this update, arranged alphabetically. Next is an index listing all tool calls, both those in the *Apple IIGS Toolbox Reference* and those documented in this volume. This index is included to make it easier to find a particular call's description, whether it is a new call or one that was previously documented. Finally, there is a general index covering both this book and the first two volumes of the *Apple IIGS Toolbox Reference.*

Note that the Beta draft only has a single index, covering the contents of Volume 3.

Chapter 26 **Apple Desktop Bus Tool Set Update**

This chapter contains new information about the Apple® Desktop Bus™ Tool Set. The complete reference to this tool set is in Volume 1, Chapter 3 of the *Apple IIGS Toolbox Reference.*

Error correction

The parameter table for the `ReadKeyMicroData` tool call ($0A09) in Volume 1 of the *Toolbox Reference* incorrectly describes the format for the `readConfig` command ($0B). The description should be as follows.

Command	*dataLength*	Name	Action
$0B	3	`readConfig`	Read configuration; *dataPtr* refers to a 3-byte data structure: Byte ADB keyboard and mouse addresses low nibble - keyboard high nibble - mouse Byte Keyboard layout and display language low nibble - keyboard layout high nibble - display language Byte Repeat rate and delay low nibble - repeat rate high nibble - repeat delay

The description of this configuration record is also wrong in the tool set summary. The following table shows the correct information.

Name	Offset	Type	Definition
`ReadConfigRec` (configuration record for `ReadKeyMicroData`)			
`rcADBAddr`	$0000	Byte	ADB keyboard and mouse addresses low nibble - keyboard high nibble - mouse
`rcLayoutOrLang`		$0001	Byte Keyboard layout and display language low nibble - keyboard layout high nibble - display language
`rcRepeatDelay`	$0002	Byte	Repeat rate and delay low nibble - repeat rate high nibble - repeat delay

Clarification

This section presents new information about the `AsyncADBReceive` call.

You can call `AsyncADBReceive` to poll a device using register 2, and it will return certain useful information about the status of the keyboard. The call returns the following information in the specified bits of register 2:

Bit 5:	0-Caps Lock key down 1-Caps Lock key up
Bit 3:	0-Control key down 1-Control key up
Bit 2:	0-Shift key down 1-Shift key up
Bit 1:	0-Option key down 1-Option key up
Bit 0:	0-Command key down 1-Command key up

Chapter 27 Audio Compression and Expansion Tool Set

This chapter documents the features of the new Audio Compression and Expansion (ACE) Tool Set. This is a new tool set, not previously documented in the *Apple IIGS Toolbox Reference.*

Error correction

This is a note to call to your attention an error in the *Apple IIGS Toolbox Reference Update* (distributed by APDA™). The description for the ACEExpand tool call carried an incorrect parameter block. This book contains a corrected description.

About Audio Compression and Expansion

The Audio Compression and Expansion (ACE) tools are a set of utility routines that compress and expand digital audio data. The tool set is designed to support a variety of methods of audio signal compression, but at present only one method is implemented.

With the present method of compression supported by the ACE tools, you can choose either of two compression ratios. You can compress a digital audio signal to half its original size or to three-eighths its original size. The ratio used is determined by a parameter of the ACE call that does the compression or expansion.

The obvious advantages of compressing an audio signal are that it takes up less space on the disk, and less time is needed to transfer the data. A digital sample that is compressed to half its original size occupies only half the space and takes only half as long to transfer. Such a sample can load from the disk twice as fast as the uncompressed version, and is much more economical to upload to or download from a commercial computer network. Note, however, that data compression and expansion requires significant processor resources, and therefore takes some time.

Uses of the ACE Tool Set

Software often includes sound effects, music, or speech. The problem with digitized sound is that it requires considerable storage space. A faithful monophonic digitization of 30 seconds of an FM radio signal occupies nearly a megabyte (MB) of disk space. A user might be somewhat reluctant to use a program that occupies so much space only to achieve sound effects. The ACE Tool Set provides you with the means to compress digitized sound signals to minimize this problem.

ACE presently supports **Adaptive Differential Pulse Code Modulation (ADPCM)**. This compression method assumes that audio signals tend to be relatively smooth and continuous. If the amplitude (loudness) of a typical audio signal is plotted against time, the graph is relatively smooth compared to a spreadsheet, a text document, or other typical files that may contain arbitrarily distributed byte-values. As a result, it is possible to compute the probable value of the next sample in the signal. ADPCM uses a static model of what the change between any given value and the next might likely be and a dynamic model of what the next actual change should be, based on the values last observed. It examines the next signal to compare its predictions against the observed value, and then encodes the difference between its prediction and the actual value.

ADPCM relies on the relative predictability of audio signals. If the changes in an audio signal are too great or sudden, the value that ADPCM records will be erroneous. In general, there is a certain statistically predictable amount of error that appears in any signal that is compressed by this method. The errors appear, not as distortions of the quality of the sound, but as pink noise, or hiss, much like the hiss on ordinary cassette recordings. Thus, although ADPCM compression is suitable for many sound compression tasks, particularly for sound effects or speech in games or business software, it is not the best choice for very high-fidelity reproduction. A signal compressed by the ADPCM method will likely be too noisy for use in professional audio, such as film soundtrack recording.

How ADPCM works

The ADPCM method assumes that any particular digital sample in a block of audio data has a value that is relatively close to those on either side of it. In fact, the noise in the reproduced signal arises from samples that vary more than the method supports. ADPCM predicts what the next value will be, and compares it with the value that is actually there. The difference is encoded in a value that is some number of bits in size, that size being specified by the application code. With ADPCM the programmer can specify encoded values either 3 or 4 bits wide. Since the original data is stored in 8-bit samples, the compression rate is either 8 to 3 or 8 to 4, depending on which size a particular program specifies.

Errors result when the difference between the original signal and the value that ADPCM predicts is greater than can be encoded in the specified number of bits. The encoded value then effectively becomes a random value, and so is perceived as audio noise. If the target code is 3 bits wide, then the difference observed by the compression algorithm is more likely to be out of range than if the code size is 4 bits. Greater compression therefore results in greater loss of fidelity.

As stated earlier, the fidelity loss sounds like hiss, not like a gross distortion of the audio signal. Even using inaccurate predictive models, ADPCM tends to produce hiss rather than more offensive forms of distortion. The technique tracks the gross characteristics of audio signals well even when the rate of errors is high. At worst, an expanded signal sounds faithful to the original, though muffled by noise.

△ **Important** The noisier a sampled signal is, the noisier the sample compressed by using ADPCM will be. Any noise that is introduced into the signal produces discontinuities in the audio data and causes errors in the compression and expansion process. For this reason, any editing, equalization, or other sound-processing effects should be applied to the original signal before it is compressed. ADPCM compression should be the last process applied to an audio signal before it is stored on the final disk. △

ACE housekeeping routines

These routines allow you to start and stop the ACE tools and to obtain tool set status information.

`ACEBootInit` $011D

Performs any initializations of the ACE tools that are necessary at boot time.

▲ **Warning** Applications must not make this call. ▲

Parameters This call has no input or output parameters. The stack is unaffected.

Errors None

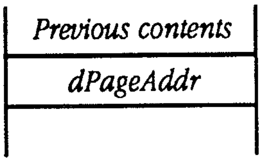
C `extern pascal void ACEBootInit();`

`ACEStartUp` $021D

Initializes the ACE tools for use by an application. `ACEStartUp` sets aside a region of bank $00, specified by *dPageAddr*, for use as the ACE tools' direct page. At present, ACE uses one 256-byte page of bank $00 memory as its direct page. Future versions of the ACE tools may use a different amount of memory for the direct page, so applications should determine the correct size for the direct page with a call to `ACEInfo`. The tool set's direct page should always begin on a page boundary.

Parameters

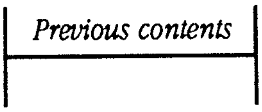
Stack before call



Word—Bank $0 starting address of direct-page space

<—SP

Stack after call



<—SP

Errors	$1D01	`aceIsActive`	ACE Tool Set already started up.
	$1D02	`aceBadDP`	Requested direct-page location invalid.

C

```
extern pascal void ACEStartUp(dPageAddr);

Word        dPageAddr;
```

`ACEShutDown` $031D

Performs any housekeeping that is required to shut down the ACE Tool Set. Applications that use the ACE tools should always make this call before quitting. The application, not the ACE Tool Set, must allocate and deallocate direct-page space in bank zero.

Parameters This call has no input or output parameters. The stack is unaffected.

Errors $1D03 `aceNotActive` ACE Tool Set not started up.

C `extern pascal void ACEShutDown();`

`ACEVersion` $041D

Returns the version number of the currently installed ACE Tool Set. This call can be made before a call to `ACEStartUp`. The *versionInfo* result will contain the information in the standard format defined in Appendix A, "Writing Your Own Tool Set," in Volume 2 of the *Toolbox Reference.*

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
	<—SP

Stack after call

Previous contents	
versionInfo	Word—Version number of ACE Tool Set
	<—SP

Errors None

C `extern pascal Word ACEVersion();`

ACEReset $051D

Resets the ACE Tool Set. This call is made by a system reset.

> ▲ **Warning** Applications should never make this call because it performs tool set initializations appropriate to a machine reset. ▲

Parameters This call has no input or output parameters. The stack is unaffected.

Errors None

C

```
extern pascal void ACEReset();
```

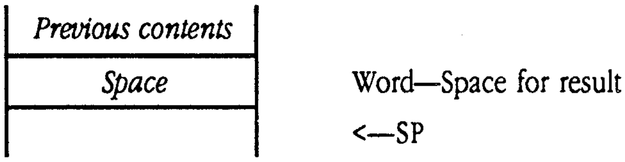
`ACEStatus` $061D

Returns a Boolean flag, which is TRUE (nonzero) if the tool set has been started up, and FALSE (zero) if it has not. This call can be made before a call to `ACEStartUp`.

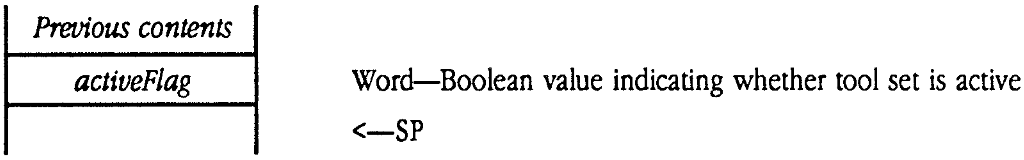
◆ *Note:* If your program issues this call in assembly language, initialize the result space on the stack to NIL. Upon return from `ACEStatus`, your program need only check the value of the returned flag. If the ACE Tool Set is not active, the returned value will be FALSE (NIL).

Parameters

Stack before call



Stack after call



Errors None

C `extern pascal Word ACEStatus();`

`ACEInfo` $071D

Returns certain information about the currently installed version of the ACE tools. This call can be made before a call to `ACEStartUp`. The *infoItemCode* parameter specifies what information the call is to return. At present, the only valid value is 0. This value specifies that the call will return the size in bytes of the direct page that ACE requires.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
infoItemCode	Word—What type of information to return
	<—SP

Stack after call

Previous contents	
– *infoItemValue* –	Long—Requested information
	<—SP

Errors	$1D04	`aceNoSuchParam`	Requested information type not supported.

C

```
extern pascal LongWord ACEInfo(infoItemCode);

Word        infoItemCode;
```

Audio Compression and Expansion tool calls

The Audio Compression and Expansion tool calls are all new calls, added to the Apple IIGS Toolbox since the first two volumes of the *Apple IIGS Toolbox Reference* were published.

ACECompBegin $0B1D

Prepares the ACE tools to compress a new audio sequence. After ACECompress completes the process of compression and returns, the ACE tools normally save certain relevant state information so that subsequent calls to ACECompress can be used on succeeding parts of the same audio sequence. It is often desirable to break a long audio signal into smaller parts for compression. The preservation of appropriate state variables allows a call to ACECompress to compress part of such a signal and then, for a subsequent call, to continue the compression process where the previous call left off.

Just before a program calls ACECompress to process a new audio sample, it should call ACECompBegin to ensure that all saved state information is cleared and that ACECompress is starting with a "clean slate." When an application is compressing a long audio sample as a number of smaller pieces, it should call ACECompBegin *only* before the *first* subsequence. Thereafter, the application should not make this call until all parts of the sequence have been processed. The state information that ACE preserves between calls allows ACECompress to process subsequent blocks, using appropriate information from previous ones.

Call ACECompBegin only before compressing the first sequence of a series of subsequences, or before compressing a single sequence that is not part of a longer sequence.

Parameters This call has no input or output parameters. The stack is unaffected.

Errors	$1D03	aceNotActive	ACE Tool Set not started up.

C `extern pascal void ACECompBegin();`

`ACECompress` $091D

Compresses a number of blocks of digital audio data·and stores the compressed data at a specified location. Each input block contains 512 bytes of data to be compressed. Your program also specifies the compression method, using the *method* parameter.

Before issuing the `ACECompress` tool call, your program should call `ACECompBegin` to prepare the ACE Tool Set for audio compression.

◆ *Note:* Because `ACECompress` is guaranteed to reduce the size of every byte of source data, the resulting data can be stored in the same place as the source data. That is, the source and destination locations in RAM can be the same.

Parameters

Stack before call

Previous contents	
– *src* –	Long—Handle to the source data
– *srcOffset* –	Long—Offset from *src* to the actual storage location
– *dest* –	Long—Handle to storage for the resulting data
– *destOffset* –	Long—Offset from *dest* to the actual storage location
nBlks	Word—Number of 512-byte blocks of source data
method	Word—Method of compression
	<—SP

Stack after call

Previous contents	
	<—SP

Errors			
	$1D05	`aceBadMethod`	Specified compression method not supported.
	$1D06	`aceBadSrc`	Specified source invalid.
	$1D07	`aceBadDest`	Specified destination invalid.
	$1D08	`aceDataOverlap`	Specified source and destination areas overlap in memory.

C

```
extern pascal void ACECompress(src, srcOffset, dest,
          destOffset, nBlks, method);

Handle    src, dest;
Long      srcOffset, destOffset;
Word      nBlks, method;
```

src, dest Contain handles to source and destination data locations, respectively.

srcOffset, destOffset Contain byte offsets from locations specified by *src* and *dest*, respectively. These parameters allow your program to position within an input sample or output buffer.

nBlks Specifies the number of 512-byte blocks of audio data to be compressed.

method Specifies the compression method to be used by `ACECompress` when processing the data. A value of 1 causes each byte of input data to be compressed to a 4-bit quantity; a value of 2 yields 3 bits per byte of input data.

Clearly, the value of the *method* parameter helps determine the size of the resulting data that `ACECompress` stores at *destOffset* bytes beyond the location specified by *dest*. When using method 1 (4-bit compression), you can calculate the number of bytes `ACECompress` will produce by multiplying the contents of the *nBlks* parameter by the number of bytes in a data block (512), multiplying that result by the number of result bits per input byte (4), and then dividing by the number of bits in a byte (8). Expressed as a formula, the calculation would be

$$((\mathit{nBlks} * 512) * 4) / 8$$

For method 2, the same basic calculation applies, except that each input byte results in 3 output bits

$$((\mathit{nBlks} * 512) * 3) / 8$$

`ACEExpand` $0A1D

Expands a previously compressed audio sample, using the method specified by the *method* parameter, and stores it at the specified location. Unlike `ACECompress`, `ACEExpand` cannot store its results in the same location as its source since the resulting data is 2 to 2.67 times as large as the source.

Parameters

Stack before call

Previous contents	
– *src* –	Long—Handle to the source data
– *srcOffset* –	Long—Offset from *src* to the actual storage location
– *dest* –	Long—Handle to storage for the resulting data
– *destOffset* –	Long—Offset from *dest* to the actual storage location
nBlks	Word—Number of 512-byte blocks to be stored at *dest*
method	Word—Method of compression
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$1D05	`aceBadMethod`	Specified compression method not supported.
	$1D06	`aceBadSrc`	Specified source invalid.
	$1D07	`aceBadDest`	Specified destination invalid.
	$1D08	`aceDataOverlap`	Specified source and destination areas overlap in memory.

C

```
extern pascal void ACEExpand(src, srcOffset, dest,
          destOffset, nBlks, method);

Handle    src, dest;
Long      srcOffset, destOffset;
Word      nBlks, method;
```

src, dest Contain handles to source and destination data locations, respectively.

srcOffset, destOffset Contain byte offsets from locations specified by *src* and *dest*, respectively. These parameters allow your program to position within the input compressed data or output buffer.

nBlks Specifies the number of 512-byte blocks of expanded data to be returned at the location *destOffset* bytes beyond *dest*. .

method Specifies the method used when the sample was compressed. A value of 1 indicates that `ACEExpand` is to expand each 4-bit quantity in the compressed sample into an 8-bit byte. A value of 2 causes `ACEExpand` to process 3-bit quantities in the compressed sample.

ACEExpBegin $0C1D

Prepares ACE to expand a new sequence. Like ACECompBegin, ACEExpBegin clears any stored state information from previous calls to expand compressed data. You can expand a large compressed sample by processing it as a series of subsequences with repeated calls to ACEExpand, because certain appropriate state variables are preserved from call to call. If you are calling ACEExpand to work on a new sequence that bears no relation to any other compressed sequence, or to expand a short sequence in just one call to ACEExpand, you should make this call first to clear these state variables. If, on the other hand, you are making a call to ACEExpand to expand a sequence that is a part of a longer sequence and is not the first subsequence, you should *not* make this call first, because it will throw away all information that ACE has recorded about the previous sequences.

Parameters This call has no input or output parameters. The stack is unaffected.

Errors $1D03 aceNotActive ACE Tool Set not started up.

C

```
extern pascal void ACEExpBegin();
```

ACE Tool Set error codes

This section lists the error codes that may be returned by Audio Compression and Expansion Tool Set calls.

Value	Name	Definition
$1D01	`aceIsActive`	ACE Tool Set already started up.
$1D02	`aceBadDP`	Requested direct-page location invalid.
$1D03	`aceNotActive`	ACE Tool Set not started up.
$1D04	`aceNoSuchParam`	Requested information type not supported.
$1D05	`aceBadMethod`	Specified compression method not supported.
$1D06	`aceBadSrc`	Specified source invalid.
$1D07	`aceBadDest`	Specified destination invalid.
$1D08	`aceDataOverlap`	Specified source and destination areas overlap in memory.

Chapter 28 **Control Manager Update**

This chapter documents new features of and information about the Control Manager. The complete Control Manager documentation is in Volume 1, Chapter 4 of the *Toolbox Reference.*

Error corrections

This section documents errors in Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference.*

- The color table for the size box control in the *Toolbox Reference* is incorrect. The correct table follows, with new information in boldface.

growOutline	word	Color of size box's outline
	Bits 8–15	= zero
	Bits 4–7	= outline color
	Bits 0–3	= zero
growNorBack	word	Color of interior when not highlighted
	Bits 8–15	= zero
	Bits 4–7	= background color
	Bits 0–3	= icon color
growSelBack	**word**	**Color of interior when highlighted**
	Bits 8–15	**= zero**
	Bits 4–7	**= background color**
	Bits 0–3	**= icon color**

- On page 4-76 of the *Toolbox Reference*, in the section that covers the SetCtlParams call, it states that the call "Sets new parameters to the control's definition procedure . . ." This description is misleading; the call does not directly set the parameters. Rather, it *sends* the new parameters to the control's definition procedure, unlike SetCtlValue, which actually sets the appropriate value in the control record and then passes the value on to the definition procedure.

Clarifications

The following items provide additional information about features previously described in the *Toolbox Reference.*

- The `barArrowBack` entry in the scroll bar table was never implemented as first intended, and is now no longer used.
- The Control Manager preserves the current port across Control Manager calls, including those that are passed through other tools, such as the Dialog Manager.
- The Control Manager preserves the following fields in the port of a window that contains controls:

`bkPat`	background pattern
`pnLoc`	pen location
`pnSize`	pen size
`pnMode`	pen mode
`pnPat`	pen pattern
`pnMask`	pen mask
`pnVis`	pen visibility
`fontHandle`	handle of current font
`fontID`	ID of current font
`fontFlags`	font flags
`txSize`	text size
`txFace`	text face
`txMode`	text mode
`spExtra`	value of space extra
`chExtra`	value of character extra
`fgColor`	foreground color
`bgColor`	background color

- The control definition procedures for simple buttons, check boxes, and radio buttons can now compute the size of their boundary rectangles automatically. The computed size is based on the size of the title string for the button.
- To ensure predictable color behavior, you should always align color table–based controls on an even pixel boundary in 640 mode. If you do not do so, the control will not appear in the colors you specify, due to the effect of dithering.

New features in the Control Manager

The Control Manager now supports a number of new features. This section discusses these new features in detail.

- Colors in control tables now use all four color bits in both modes; they formerly used only 2 bits in 640 mode. This change affects all control color tables defined in the *Toolbox Reference*. For any applications that use color controls in 640 mode, the effect is that controls will be a different color. This change was made so that dithered colors can be used with controls.
- The scroll bar control definition procedure now maintains the required relationship among the `ctlValue`, `viewSize`, and `dataSize` fields of a scroll bar record. Prior to *Apple IIGS System Software 5.0*, it was the responsibility of the application to ensure that the `ctlValue` field never exceeded the difference between `dataSize` and `viewSize` (`dataSize - viewSize`). The scroll bar control definition procedure now adjusts the `ctlValue` or `dataSize` field if the other quantities are set to invalid values.

  For example, if `viewSize` = 30 and `dataSize` = 100, then the maximum `ctlValue` allowed is 70. If an application set the `ctlValue` field to 80, the Control Manager would adjust `dataSize`to 110. In this same example, if `ctlValue` = 70 and the application set `dataSize` to 90, the Control Manager would adjust `ctlValue` to 60.

  Changes to the `viewSize` field can also invalidate the three settings. In the example mentioned before, in which `ctlValue` = 70, `viewSize` = 30, and `dataSize` = 100, setting `viewSize` to 40 would cause Control Manager to set `ctlValue` to 60.

Keystroke processing in controls

Apart from the normal use of keystrokes to enter data, the Control Manager now supports two special uses for keyboard data: **keystroke equivalents** and switching between certain types of controls.

Many types of controls support keystroke equivalents, which allow the user to select the control by pressing a keyboard key. You assign a keystroke equivalent for a control in its control template (see "New Control Manager templates and records" later in this chapter for specifics on control templates). When the user presses that key, your program receives an event just as if the user had clicked in the control. Further, the system will automatically highlight and dim the control. Note that this feature is only available to controls that have been created with the `NewControl2` tool call, and for which the `fCtlWantsEvents` bit has been set to 1 in the `moreFlags` word of the control template. See "New and changed controls" later in this chapter for information about which controls support keystroke equivalents.

Edit field controls (LineEdit controls and TextEdit controls) accept keystrokes as part of their normal function. Note, however, that there can be more than one edit field control in a window. Under these circumstances, the user moves among these controls by pressing the Tab key. In addition, the system must keep track of which control is meant to receive user keystrokes. To do so, the Control Manager now supports the notion of a **target** control. The target control is that edit field control which is the current recipient of user actions (keystrokes and menu items).

The Control Manager and resources

You can now specify most data for the Control Manager using either pointers, handles, or resource IDs (see "Chapter 45, "Resource Manager,"" in this book for complete information on resources). Because the form of the specification may differ, the Control Manager (as well as many other tool sets) also requires a **reference type**, which indicates whether a particular reference is a pointer, handle, or resource ID. You set the reference type and the reference as appropriate in the control template you pass to the Control Manager `NewControl2` tool call.

You can use resources to store a wide variety of items for the Control Manager. For example, the titles associated with simple buttons, radio buttons, and check boxes created with the `NewControl2` tool call may be stored as resources. As a result, your application may free the space devoted to the title string after the control has been created. Similarly, you can define control definition procedures as resources. The Control Manager will load the code when it is needed.

The Control Manager handles resources differently, based upon the data's degree of permanence. For temporary information, Control Manager loads the resource, uses the data, and then frees the resource (using the `ReleaseResource` tool call). For permanent information, the Control Manager loads the resource each time the resource is accessed. Such resources should be unlocked and unpurgeable.

The current version of the Apple IIGS system software keeps the control definition procedure for icon button controls in the system resource file. In the future, the sytsem may store other defProcs in this resource file. Consequently, you should ensure that the Resource Manager can reach the system resource file in any resource search path you set up (see Chapter 45, "Resource Manager," later in this book for more information on the resource file search path).

New and changed controls

The Control Manager now supports more standard control types. In addition to the original standard controls (buttons, check boxes, radio buttons, size boxes, and scroll bars), the Control Manager now supports the following controls:

- **Static text controls** display text messages in a rectangle that you define. The displayed text supports word wrap and character styling. This text cannot be edited by the user.
- **Picture controls** draw a picture into a defined rectangle.
- **Icon button controls** allow you to present an icon as part of a button control. A defined icon is displayed within the bounds of the rectangle that represents the button control on the screen. Icon buttons include support for keyboard equivalents.
- **LineEdit controls** allow the user to enter single-line items.
- **TextEdit controls**, supported by the new TextEdit tool set (see Chapter 49, "TextEdit," in this book), allow the user to edit text within a defined rectangle, which can extend beyond a single line.
- **Pop-up menu controls** support scrolling lists of possible selection options that appear when the user selects the control.
- **List controls** display scrollable lists of items.

In order to create any of these new controls, you must set up the appropriate **control template** and call `NewControl2`. Unlike the `NewControl` tool call, which accepts its control definition on the stack, `NewControl2` defines controls according to the contents of one or more control templates. These templates contain all the information necessary for the Control Manager to create controls. Your application fills each control template with the data appropriate to the control you wish to create. The Control Manager uses this input specification to construct the corresponding control record and create the control. You can use this technique to create any control, not just the new control types. For complete information on the format and content of these control templates, see "New Control Manager templates and records" later in this chapter.

All controls created by `NewControl2`, rather than `NewControl`, are referred to as **extended controls**. Functionally, extended controls do not differ from controls created by `NewControl`. In fact, extended control records will work with all Control Manager tool calls. However, the control record for an extended control contains more data than the old-style record. In addition, many new Control Manager calls and features are valid only for extended controls. Note that any control created by `NewControl2` is an extended control, not just the new control types. For complete information on the format and content of extended control records, see "New Control Manager templates and records" later in this chapter.

You may directly call `NewControl2` or you may invoke it indirectly by calling `NewWindow2`. See Chapter 45, "Resource Manager," and Chapter 52, "Window Manager Update," for details on new window calls.

The following sections discuss each type of control supported by the Control Manager. For the original controls, these sections address new features provided by the Control Manager. For new control types, these sections introduce you to the functionality now provided.

Simple button control

Simple button controls created with the `NewControl2` tool call can support keystroke equivalents, which allow the user to activate the button by pressing an assigned key on the keyboard. See "Keystroke processing" earlier in this chapter for details.

Check box control

Check box controls created with the `NewControl2` tool call can support keystroke equivalents, which allow the user to activate the box by pressing an assigned key on the keyboard. See "Keystroke processing" earlier in this chapter for details.

Icon button control

This new type of control can display an icon as well as text in a defined window. You specify the boundary rectangle for the window and a reference to the icon when you create the control (see Chapter 17, "QuickDraw II Auxiliary," in Volume 2 of the *Toolbox Reference* for information about icons). You can create icon button controls only with the `NewControl2` tool call.

Icon button controls operate much like simple button controls. Note, however, that with icon controls, the control rectangle is inset slightly from its specified coordinates before the button is drawn. As a result, outlined round buttons stay completely within the specified control rectangle (this is not the case for an outlined round simple button control). Icon button controls support keyboard equivalents (see "Keystroke processing" earlier in this chapter for details).

The icon is drawn each time the control is drawn. The icon and text are centered in the specified control rectangle. If the control has no text, the icon is still centered. The icon is not clipped to the control rectangle. If the icon is larger than the specified control rectangle, then when you erase the control, that portion of the icon that lay outside the rectangle will not be erased.

Note that icon controls require the QuickDraw II Auxiliary and Resource Manager tool sets. Note as well that the control definition procedure for icon buttons is kept in the system resources file.

LineEdit control

This new control type lets your application manage single-line, editable items in a window. You specify the boundary rectangle for the text, the maximum number of characters allowed, and an initial value for the displayed text string when you create the control with the `NewControl2` tool call. The text is updated each time the control is drawn. LineEdit controls also support password fields, which do not echo the characters entered by the user. Rather, the control echoes each typed character as an asterisk.

LineEdit controls respond to both mouse and keyboard events. If your application uses TaskMaster, the system will handle most events automatically. To take full advantage of TaskMaster, set the `tmContentControls`, `tmControlKey`, and `tmIdleEvents` flags in the `taskMask` field of the task record to 1 (see
Chapter 52, "Window Manager Update," for information about the new features in TaskMaster).

If your application does not use TaskMaster, when the user presses the mouse button in a LineEdit control your application must call `TrackControl` to track the mouse and perform appropriate text selection. TaskMaster will do this automatically if you have set the `tmContentControls` flag to 1 in the `taskMask` field of the task record.

Without TaskMaster, your application sends keyboard events to LineEdit controls using the `SendEventToCtl` tool call (see "New Control Manager calls" later in this chapter). First, your code must check for menu key equivalents. If none are found, then issue the `SendEventToCtl` call, setting `targetOnlyFlag` to FALSE (all controls that want events are searched), `windowPtr` to NIL (find the top window), and `extendedTaskRecPtr` to refer to the task record containing the keystroke information. Again, TaskMaster will do all this for you if you have set the `tmControlKey` flag to 1 in the `taskMask` field.

To keep the caret flashing, your application must send idle events to the LineEdit control. In order to do this, issue a `SendEventToCtl` call, setting `targetOnlyFlag` to TRUE (send event only to target control), `windowPtr` to NIL (use top window), and `extendedTaskRecPtr` to refer to the task record containing the event information. TaskMaster will do this for you if you have set the `tmIdleEvents` flag to 1 in the `taskMask` field.

The LineEdit tool set performs line editing in LineEdit controls. If you want to issue LineEdit tool calls directly from your program, retrieve the LineEdit record handle from the `ctlData` field of the control record for the LineEdit control.

List control

This new control type allows your program to display lists from which the user may select one or more items. You have the benefit of full List Manager functionality, with respect to such features as selection window scrolling and item selection (single item, arbitrary items, or ranges). You specify the parameters for the list as well as the initial conditions for its display when you define the control. The Control Manager and the List Manager take care of the rest. You can create list controls only with the `NewControl2` tool call.

List controls use the List Manager tool set. In order to understand how to use this control in your application, see Chapter 11, "List Manager," in Volume 1 of the *Toolbox Reference.*

Picture control

This new control type displays a QuickDraw picture in a specified window. You specify the boundary rectangle for the control and a reference to the picture when you create the control. The picture is drawn each time the control is drawn. You can create picture controls only with the `NewControl2` tool call.

Note that when the picture is drawn, the boundary rectangle for the control is used as the picture destination rectangle (see Chapter 17, "QuickDraw II Auxiliary," in Volume 2 of the *Toolbox Reference* for details about picture drawing). As a consequence, the picture may be scaled at draw time if the original picture frame does not have the same dimensions as the control rectangle. To force the picture to be displayed at its original size, and thus avoid scaling, set the lower-right corner of the control rectangle to (0,0). The Control Manager recognizes this value at control initialization time, and sets the control rectangle to be the same size as the picture frame.

In general, a click in a picture control is ignored. However, the Control Manager provides facilities to inform your application if the user clicks in the control. To make a picture control inactive, set the `ctlHilite` field to $FF, otherwise the control is active and may receive user events.

Note that picture controls require the QuickDraw II Auxiliary tool set.

Pop-up control

This new control type allows you to define and support pop-up menus inside a window. You specify the boundary rectangle for the control, along with a reference to the menu definition when you create the control with the `NewControl2` tool call. The menu title becomes the title of the control, and the current selection for the control is defined by the initial value.

Pop-up controls respond to both mouse and keyboard events. If your application uses TaskMaster, the system will handle most events automatically. To take full advantage of TaskMaster, set the `tmContentControls` and `tmControlKey` flags in the `taskMask` field of the task record to 1 (see Chapter 52, "Window Manager Update," for information about the new features in TaskMaster).

If your application does not use TaskMaster, when the user presses the mouse button inside a Pop-up control, your application must call `TrackControl` to track the mouse and present the pop-up menu to the user. TaskMaster will do this for you if you have set the `tmContentControls` flag to 1 in the `taskMask` field.

Without TaskMaster, your program sends keyboard events to pop-up menu controls using the `SendEventToCtl` tool call (see "New Control Manager calls" later in this chapter). First, check for menu key equivalents. If none are found, then issue the `SendEventToCtl` call, setting `targetOnlyFlag` to FALSE (all controls that want events are searched), `windowPtr` to NIL (find the top window), and `extendedTaskRecPtr` to refer to the task record containing the keystroke information. TaskMaster will do all this for you if you have set the `tmControlKey` flag to 1 in the `taskMask` field.

Note that the Control Manager places the current user selection value into `ctlValue`. If you need to retrieve the user selection number, you may do so from this field.

Radio button control

Radio button controls created with the `NewControl2` tool call can support keystroke equivalents, which allow the user to choose a button by pressing an assigned key on the keyboard. See "Keystroke processing" earlier in this chapter for details.

Scroll bar control

Scroll bar controls provide no new features.

Size box control

You can now set up size box controls to automatically invoke `GrowWindow` and `SizeWindow` if you create the control with the `NewControl2` tool call. When the user drags the size box, if the `fCallWindowMgr` bit in the `flag` field of the size box control template is set to 1 (see the description of the size box control template in "New Control Manager templates and records" later in this chapter), the Control Manager will call `GrowWindow` and `SizeWindow` to track the control and resize the window rectangle. If this flag is set to 0, then the control is merely highlighted.

Static text control

This new control type displays uneditable (hence, "static") text in a specified window. You can place font, style, size, and color changes into the displayed text, affording you great freedom to create a distinctive text display. In addition, static sext controls can accommodate text substitution. With this feature, you can customize the displayed text to fit run-time circumstances. You can create static text controls only with the `NewControl2` tool call.

If you are going to use text substitution in your static text, your application must set up the control template correctly (set `fSubstituteText` in `flag` to 1) and tell the system where the substitution array is kept (issue the `SetCtlParamPtr` Control Manager tool call). The text substitution array has the same format as that used by the `AlertWindow` call (see Chapter 52, "Window Manager Update," for information about `AlertWindow` and for substitution array format and content).

In general, applications ignore clicks in static text controls. However, the Control Manager provides facilities to inform your application if the user clicks in the control. To make a static text control inactive, set the `ctlHilite` field to $FF, otherwise the control is active and may receive user events.

Note that static text controls require the LineEdit, QuickDraw II Auxiliary, and Font Manager tool sets.

TextEdit control

This control lets the user create, edit, or view multiline items in a window. You specify the boundary rectangle for the edit window, parameters governing the amount of text to be entered, and, optionally, some initial text to display. The TextEdit control does the rest. You can only create TextEdit controls with the `NewControl2` tool call.

The TextEdit control uses the TextEdit tool set. TextEdit is a new tool set, and is completely described in Chapter 49, "TextEdit," later in this book. You should familiarize yourself with the material in that chapter before using this control.

New control definition procedure messages

Previously, control definition procedures had to support 13 message types (see Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* for a discussion of the original message types). When you create custom controls with new control records (see "New Control Manager templates and records" later in this chapter), your control will have to support some additional messages.

Value	Control Message	Description
13	`ctlHandleEvent`	Handle a keystroke or menu selection
14	`ctlChangeTarget`	Issued when control's target status has changed
15	`ctlChangeBounds`	Issued when control's boundary rectangle has changed
16	`ctlWindChangeSize`	Window has been grown or zoomed
17	`ctlHandleTab`	Control has been tabbed to
18	`ctlNotifyMultiPart`	A multipart control must be hidden, drawn, or shown
19	`ctlWinStateChange`	Window state has changed

In addition, the `initCtl`, `dragCntl`, and `recSize` messages have new control routine interfaces when used with extended controls. The following sections discuss each new or changed message in detail.

If you must draw when handling control messages, your control definition procedure should save the current GrafPort and set the port correctly for your control before drawing. After your defProc is finished drawing, restore the previous GrafPort. Note that saving the current GrafPort includes saving penstate, all pattern and color information, and all regions in the port to which your program draws.

To maintain compatibility with future versions of the Control Manager, control definition procedures should always return a *retValue* of 0 for unrecognized and unsupported control message types. In addition, if you use custom control messages, be careful to assign type values greater than $8000 (decimal 32,767).

Initialize routine

Previously, *ctlParam* contained *param1* and *param2* from `NewControl`. If you create your custom control with `NewControl`, these input parameters are the same. However, if you create your control with `NewControl2` (see "New Control Manager calls" later in this chapter), then *ctlParam* contains a pointer to the control template for the control.

Drag routine

The result code for the drag routine now contains additional information that allows control definition procedures to abort tracking. Previously, *retValue* indicated whether or not your defProc wanted the Control Manager to do the dragging. For controls created with `NewControl`, this is still the case. For controls created with `NewControl2`, your defProc uses the low-order word of *retValue* exactly as before (zero means that the Control Manager should drag the control; nonzero means your control definition procedure handled it). Your defProc returns the part code of the control in the high-order word (see Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* for information on control part codes). If this value is 0, then the Control Manager assumes that the user aborted the drag operation and performs no screen updates.

Record size routine

Previously, *ctlParam* was undefined for this routine. Now, the Control Manager sets *ctlParam* to 0 for controls created with `NewControl`. For controls created with `NewControl2`, *ctlParam* contains a pointer to the control template.

Event routine

To pass information for all events, including keystroke or mouse events, the Control Manager calls the control definition procedure with the `ctlHandleEvent` message. Only controls you create with either the `fCtlWantsEvents` bit or the `fCtlCanBeTarget` bit set to 1 in the `moreFlags` field of the control template will receive this message (see "New Control Manager templates and records" later in this chapter for detailed information on these flags). The first qualifying control in the control list gets the first opportunity to handle the event. If that control processes the event, then no other controls see it. If, however, that control does not process the event, the Control Manager passes the event to the next qualifying event in the list. This process continues until a control handles the event, or the list is exhausted. If no defProc handles the event, TaskMaster passes the event to the application.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
ctlMessage	Word—`ctlHandleEvent` message
– *ctlParam* –	Long—Pointer to task record containing event information
–*theControlHandle*–	Long—Handle to control
	<—SP

Stack after call

Previous contents	
– *retValue* –	Long—$FFFFFFFF if control took the event; $0 if control did not
	<—SP

Target routine

To signal a change in the control's target status (the control is now, or is no longer, the target), the Control Manager calls the control definition procedure with the `ctlChangeTarget` message. Note that this message is sent to both the previous target control and the new target control. Your control definition procedure can distinguish which control is the new target by examining the `fCtlTarget` bit in the `ctlMoreFlags` field of the control record. The new target control will have this bit set to 1 in its control record. The previous target will have the bit set to 0.

In response to the `ctlChangeTarget` message, some control definition procedures will change the appearance of their control on the screen or perform other actions as appropriate. For example, LineEdit and TextEdit controls display an insertion point or a text selection only when they are the target.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
ctlMessage	Word—`ctlChangeTarget` message
– *ctlParam* –	Long—Undefined
–*theControlHandle*–	Long—Handle to control
	<—SP

Stack after call

Previous contents	
– *retValue* –	Long—Undefined
	<—SP

Bounds routine

To signal to the control that its boundary rectangle has changed the Control Manager calls the control definition procedure with the `ctlChangeBounds` message. In response to this message, your control definition procedure should adjust its internal control record variables to account for the new rectangle. For example, any subrectangles defined for a control may need to change whenever the boundary rectangle changes.

◆ *Note:* This message is not supported by control definition procedures currently provided by Apple; however, you should handle this message in any custom controls you create.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
ctlMessage	Word—`ctlChangeBounds` message
– *ctlParam* –	Long—Undefined
–*theControlHandle*–	Long—Handle to control
	<—SP

Stack after call

Previous contents	
– *retValue* –	Long—Undefined
	<—SP

Window size routine

The Control Manager calls the control definition procedure with the `ctlWindChangeSize` message whenever the user has changed the size of the control window. In response to this message, your control definition procedure should do what is necessary to maintain a consistent screen presentation. This may entail resizing multipart controls, moving size boxes, and so on.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
ctlMessage	Word—`ctlWindChangeSize` message
– *ctlParam* –	Long—Undefined
–*theControlHandle*–	Long—Handle to control
	<—SP

Stack after call

Previous contents	
– *retValue* –	Long—Undefined
	<—SP

Tab routine

Your control definition procedure receives the `ctlHandleTab` message when the user has hit the Tab key while another control is the target. That control's defProc will have issued the `MakeNextCtlTarget` tool call before sending this control message. As a result, your control is the target control. The control definition procedure should perform the appropriate actions in response to becoming the target as a result of a Tab keystroke, rather than a mouse click. For example, in response to this message, LineEdit and TextEdit control definition procedures select all the text in the control in preparation for user input.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
ctlMessage	Word—`ctlHandleTab` message
– *ctlParam* –	Long—Undefined
–*theControlHandle*–	Long—Handle to control
	<—SP

Stack after call

Previous contents	
– *retValue* –	Long—Undefined
	<—SP

Notify multipart routine

The Control Manager calls the control definition procedure with the `ctlNotifyMultiPart` message to signal that a multipart control needs to be hidden, shown, or drawn. This message is relevant only to multipart controls, which may not fit within their boundary rectangle (for example, list controls consist of the list itself and a scroll control, which is separate). That is, the `fCtlIsMultiPart` bit in the `moreFlags` field of the control template must be set to 1 for a control to receive this message. In response to this message, your defProc must do what is needed to hide or show the control completely.

The low-order word of *ctlParam* tells the defProc what to do.

0	Hide the entire control
1	Erase the entire control
2	Show the entire control
3	Show one control

Parameter

Stack before call

Previous contents	
– *Space* –	Long—Space for result
ctlMessage	Word—`ctlNotifyMultiPart` message
– *ctlParam* –	Long—High-word is undefined; low-word contains option
–*theControlHandle*–	Long—Handle to control
	<—SP

Stack after call

Previous contents	
– *retValue* –	Long—Undefined
	<—SP

Window change routine

The Control Manager calls the control definition procedure with the `ctlWinStateChange` message to signal that the state of the window containing the control has changed. For example, a control definition procedure will receive this message whenever the control's window is activated or deactivated. At this time, the control definition procedure may draw dimmed controls in windows that have been unhidden.

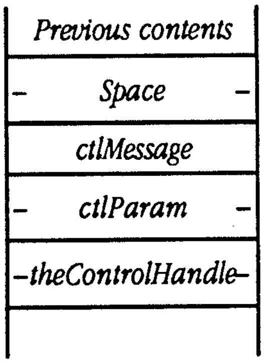
The low-order word of the *ctlParam* parameter contains the new state of the window:

$0000	The window has been deactivated
$0001	The window has been activated

The high-order word is undefined.

Parameter

Stack before call



Long—Space for result

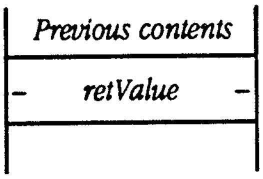
Word—`ctlWinStateChange` message

Long—Low word contains new window state; high word undefined

Long—Handle to control

<—SP

Stack after call



Long—Undefined

<—SP

New Control Manager calls

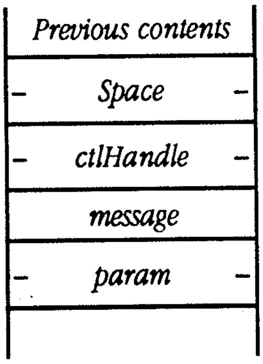
The following sections describe new Control Manager tool calls, in alphabetical order by call name.

`CallCtlDefProc` $2C10

This routine calls the specified control with the specified control message and parameter. Set the *param* parameter to 0 if the control definition procedure does not accept an input parameter (see "New control definition procedure messages" earlier in this chapter for information on input parameters for defProc messages).

Parameters

Stack before call



Long—Space for result from control definition procedure

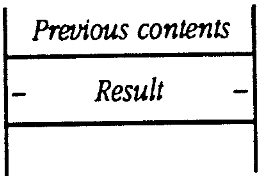
Long—Handle of control to be called

Word—Control message to send to control definition procedure

Long—Parameter to pass to control definition procedure

<—SP

Stack after call



Long—Result value from control definition procedure

<—SP

Errors None

C

```
extern pascal Long CallCtlDefProc(ctlHandle,
          message, param);

Handle    ctlHandle;
Word      message;
Long      param;
```

`CMLoadResource` $3210

This is an entry point to the internal Control Manager routine that loads resources. You specify the resource type and ID of the resource to be loaded. See Chapter 45, "Resource Manager," for more information on resources.

Any errors during resource load result in system death.

▲ **Warning** Applications must never issue this call. ▲

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
resourceType	Word—Type of resource to load
– *resourceID* –	Long—ID of resource to load
	<—SP

Stack after call

Previous contents	
– *resourceHandle* –	Long—Handle of loaded resource
	<—SP

Errors None

C

```
extern pascal Handle CMLoadResource(resourceType,
          resourceID);

Word      resourceType;
Long      resourceID;
```

`CMReleaseResource` $3310

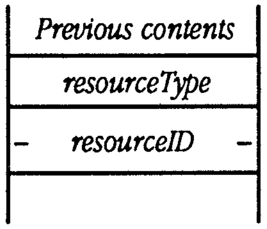
This is an entry point to the internal Control Manager routine that releases resources. You specify the resource type and ID of the resource to be released. The resource is released by marking it purgeable. See Chapter 45, "Resource Manager," for more information on resources.

Any errors result in system death.

▲ **Warning** Applications must never issue this call. ▲

Parameters

Stack before call

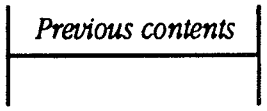


Word—Type of resource to release

Long—ID of resource to release

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void CMReleaseResource(resourceType,
               resourceID);

Word      resourceType;
Long      resourceID;
```

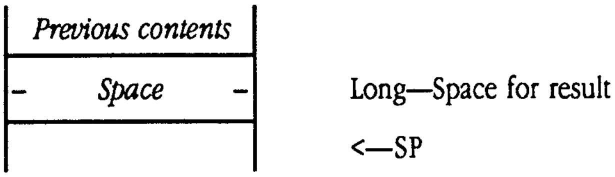
`FindTargetCtl` $2610

Searches the control list for the active window and returns the handle of the target control (the control that is currently the target of user keystrokes). `FindTargetCtl` returns the handle of the first control that has the `fCtlTarget` flag set to 1 in the `ctlMoreFlags` field of its control record. If no target control is found or an error occurs, then the call returns a NIL handle.

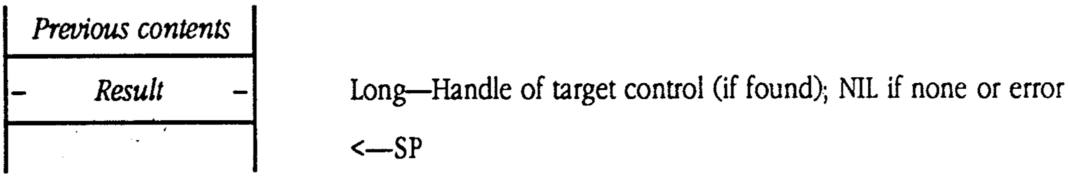
This call will only return a handle to an extended control.

Parameters

Stack before call



Stack after call


Previous contents
– Result –
Long—Handle of target control (if found); NIL if none or error
<—SP

Errors

$1004	`noCtlError`	No controls in window
$1005	`noSuperCtlError`	No extended controls in window
$1006	`noCtlTargetError`	No target extended control
$100C	`noWind_Err`	No front window

C

```
extern pascal Handle FindTargetCtl();
```

GetCtlHandleFromID $3010

Retrieves the handle to the control record for a control with a specified `ctlID` field value. The `ctlID` field is an application-defined tag for a control. Set the `ctlID` field with the `SetCtlID` tool call; read the contents of the `ctlID` field with `GetCtlID`.

If an error occurs, the returned handle is NIL.

This call is valid only for extended controls.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
– *windowPtr* –	Long—Pointer to window for control list search; NIL=top window
– *ctlID* –	Long—ID value for desired control
	<—SP

Stack after call

Previous contents	
– *ctlHandle* –	Long—Handle for specified control
	<—SP

Errors	$1004	`noCtlError`	No controls in window
	$1005	`noSuperCtlError`	No extended controls in window
	$1009	`noSuchIDError`	The specified ID cannot be found
	$100C	`noWind_Err`	There is no front window

C

```
extern pascal Long GetCtlHandleFromID(windowPtr,
          ctlID);

Pointer   windowPtr;
Long      ctlID;
```

`GetCtlID` $2A10

Returns the `ctlID` field from the control record of a specified control. The `ctlID` field is an application-defined tag for a control. Your application can use this field in many ways. For example, since the value of `ctlID` is known at compile time, you can construct efficient routing code for handling control messages for many different controls.

Use the `SetCtlID` Control Manager tool call to set the `ctlID` field.

If the specified control is not an extended control, the resulting ID is undefined, and an error is returned.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
– *ctlHandle* –	Long—Handle to control
	<—SP

Stack after call

Previous contents	
– *ctlID* –	Long—`ctlID` for specified control
	<—SP

Errors			
	$1004	`noCtlError`	No controls in window
	$1007	`notSuperCtlError`	Action valid only for extended controls

C

```
extern pascal Long GetCtlID(ctlHandle);

Handle    ctlHandle;
```

`GetCtlMoreFlags` $2E10

Gets the contents of the `ctlMoreFlags` field of the control record for a specified control. The `ctlMoreFlags` field contains flags governing target status, event processing, and other aspects of the control.

Use the `SetCtlMoreFlags` Control Manager tool call to set the `ctlMoreFlags` field.

If the specified control is not an extended control, the result is undefined, and an error is returned.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
– *ctlHandle* –	Long—Handle to control
	<—SP

Stack after call

Previous contents	
ctlMoreFlags	Word—`ctlMoreFlags` for specified control
	<—SP

Errors	$1004	`noCtlError`	No controls in window
	$1007	`notSuperCtlError`	Action valid only for extended controls

C

```
extern pascal Word GetCtlMoreFlags(ctlHandle);

Handle    ctlHandle;
```

`GetCtlParamPtr` $3510

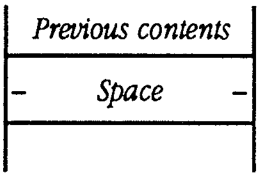
Retrieves the pointer to the current text substitution array for the Control Manager. This array contains the information used for text substitution in static text controls (see "Static text control" elsewhere in this chapter for details).

Set the contents of this field with the `SetCtlParamPtr` Control Manager tool call.

◆ *Note:* This pointer is global to the Control Manager; it is not associated with a specific control. As a result, desk accessories should be very careful when using this feature to save and restore the previous contents of the field.

Parameters

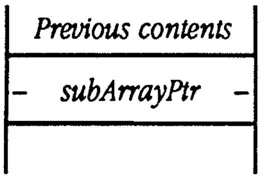
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to text substitution array

<—SP

Errors None

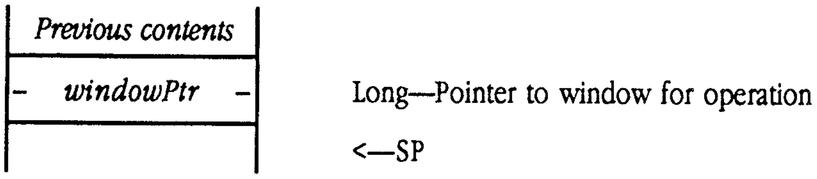
C `extern pascal Pointer GetCtlParamPtr();`

InvalCtls $3710

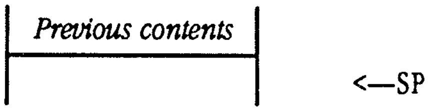
This call invalidates all rectangles for all controls in a specified window.

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal void InvalCtls(windowPtr);

Pointer   windowPtr;
```

`MakeNextCtlTarget` $2710

Makes the next eligible control the target control. This routine searches the control list of the active window for the first target control (`fCtlTarget` bit on in the `ctlMoreFlags` field of the control record). It then clears the target flag for this control, and searches for the next control in the control list that can be the target (`fCtlCanBeTarget` bit set to 1 in `ctlMoreFlags`), and makes that control the target. The call returns the handle of the new target control.

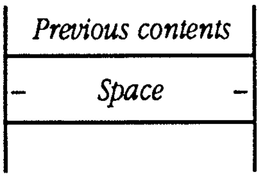
Both affected controls (the old and new target) will receive `ctlChangeTarget` messages from the Control Manager.

If an error occurs, the returned handle is NIL.

This call is valid only for extended controls.

Parameters

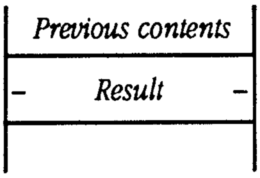
Stack before call



Long—Space for result (handle)

<—SP

Stack after call



Long—Handle of new target control; NIL if error

<—SP

Errors			
	$1004	`noCtlError`	No controls in window
	$1005	`noSuperCtlError`	No extended controls in window
	$100B	`noCtlToBeTargetError`	No control could be made target

C `extern pascal Handle MakeNextCtlTarget();`

`MakeThisCtlTarget` $2810

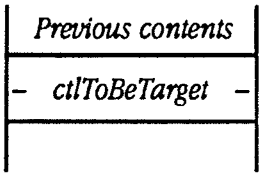
This routine makes the specified control the target. You specify the control that is to become the target control by passing its handle to this routine. This call will work for both active and inactive windows.

Both affected controls (the old and new target) will receive `ctlChangeTarget` messages from the Control Manager.

This call is valid only for extended controls.

Parameters

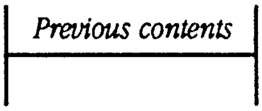
Stack before call



Long—Handle to control to be made target

<—SP

Stack after call



<—SP

Errors			
	$1007	`notSuperCtlError`	Action valid only for extended controls
	$1008	`canNotBeTargetError`	Specified control cannot be made target

C

```
extern pascal void MakeThisCtlTarget(ctlToBeTarget);

Handle    ctlToBeTarget;
```

`NewControl2` $3110

Creates one or more new controls. You specify the parameters governing those controls in control templates that are passed to `NewControl2` (see "New Control Manager templates and records" later in this chapter). If `NewControl2` creates a single control, it returns the handle to that control in *Result*. If `NewControl2` creates two or more controls, it returns 0. For sample code showing how to use the `NewControl2` tool call, see "Control Manager code example" later in this chapter.

All controls created by `NewControl2` have new style control records and are extended controls.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
– *ownerPtr* –	Long—Pointer to window for control(s)
referenceDesc	Word—Describes contents of reference
– *reference* –	Long—Reference of a type defined by *referenceDesc*
	<—SP

Stack after call

Previous contents	
– *Result* –	Long—Control handle (if single control created) or 0
	<—SP

Errors None

C

```
extern pascal Handle NewControl2(ownerPtr,
          referenceDesc, reference);

Pointer   ownerPtr;
Word      referenceDesc;
Long      reference;
```

referenceDesc Defines the contents and type of item referenced by *reference.* Possible values for *referenceDesc* are:

`singlePtr`	0	*reference* is a pointer to a single item template
`singleHandle`	1	*reference* is a handle for a single item template
`singleResource`	2	*reference* is a resource ID of a single item template
`ptrToPtr`	3	*reference* is a pointer to a list of pointers to item templates
`ptrToHandle`	4	*reference* is a pointer to a list of handles for item templates
`ptrToResource`	5	*reference* is a pointer to a list of resource IDs of item templates
`handleToPtr`	6	*reference* is a handle to a list of pointers to item templates
`handleToHandle`	7	*reference* is a handle to a list of handles for item templates
`handleToResource`	8	*reference* is a handle to a list of resource IDs of item templates
`resourceToResource`	9	*reference* is a résource ID of a list of resource IDs of item templates

If *reference* defines a list, that list is a contiguous array of template references (pointers, handles, or resource IDs), terminated with a NULL entry.

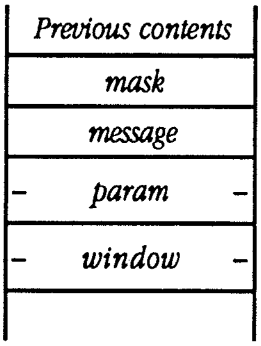
`NotifyCtls` $2D10

Calls the control definition procedures for extended controls in a specified window, sending a specified control message and parameter. You determine which controls are to be called by setting up the *mask* field. This routine compares the value of *mask* with that of the `ctlMoreFlags` field of the control record for each control in the window. If any of the bits you have specified in *mask* are set to 1 in `ctlMoreFlags`, the control is sent the message you have specified (*mask* is bitwise ANDed with `ctlMoreFlags`; a nonzero result yields a call to the control).

Set the *param* parameter to 0 if the control definition procedure does not accept an input parameter (see "New control definition procedure messages" earlier in this chapter for information on input parameters for defProc messages).

Parameters

Stack before call



Word—Bit mask to be compared with `ctlMoreFlags`

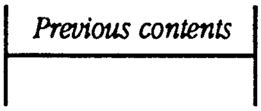
Word—Control message to send to control definition procedures

Long—Parameter to pass to control definition procedures

Long—GrafPort of window whose control list is to be searched

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void NotifyCtls(mask, message, param,
                    window);

Word          mask, message;
Long          param, window;
```

`SendEventToCtl` $2910

Passes a specified extended task record (which must comply with the new format defined in Chapter 52, "Window Manager Update," in this book) to the appropriate control or controls. This call returns a Boolean indicating whether the event was fielded by a control, and returns the handle of the control that serviced the event in `taskData2` of the task record for the event.

The *targetOnlyFlag* parameter governs the way the Control Manager searches for a control to field the event. If *targetOnlyFlag* is set to TRUE, `SendEventToCtl` sends the event to the target control. If there is no target control, *Result* is FALSE and `taskData2` is undefined.

If *targetOnlyFlag* is set to FALSE, `SendEventToCtl` conducts a two-part search for a control to field the event. First, Control Manager looks for non-edit field controls that want keystrokes (for example, buttons with keystroke equivalents). The Control Manager tries to send the event to each such control (with the `ctlHandleEvent` control message). If no control accepts the event, the Control Manager looks for an edit field control (LineEdit or TextEdit) that can become the target. If no control accepts the event and there is no target, *Result* is FALSE and `taskData2` is undefined. Otherwise, *Result* is TRUE and `taskData2` contains the handle of the accepting control.

This call is valid only for extended controls.

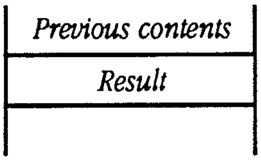
◆ *Note:* If a control can be made the target (`fCtlCanBeTarget` is set to 1 in `ctlMoreFlags` of its control record), then the Control Manager will send it events regardless of the setting of the `fCtlWantsEvents` bit.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result Boolean
targetOnlyFlag	Word—(Boolean) TRUE=send to target only; FALSE=all controls
– *windowPtr* –	Long—Pointer to window to search; NIL for top window
– *eTaskRecPtr* –	Long—Pointer to extended task record for event
	<—SP

Stack after call



Word—(Boolean) TRUE if event accepted; otherwise FALSE

<—SP

Errors	$1005	`noSuperCtlError`	No extended controls in window
	$100C	`noWind_Err`	There is no front window

C

```
extern pascal Boolean SendEventToCtl(targetOnlyFlag,
                windowPtr, eTaskRecPtr);

Word        targetOnlyFlag;
Pointer     windowPtr, eTaskRecPtr;
```

`SetCtlID` $2B10

Sets the `ctlID` field in the control record of a specified control. The `ctlID` field is an application-defined tag for a control. Your application can use this field in many ways. For example, since the value of `ctlID` is knowable at compile time, you can construct efficient routing code for handling control messages for many different controls.

Retrieve the `ctlID` field for a control with the `GetCtlID` Control Manager call.

If the specified control is not an extended control, an error is returned.

Parameters

Stack before call

Previous contents	
– *newID* –	Long—New `ctlID` value for the control
– *ctlHandle* –	Long—Handle to control
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$1004	`noCtlError`	No controls in window
$1007	`notSuperCtlError`	Action valid only for extended controls

C

```
extern pascal void SetCtlID(newID, ctlHandle);

Long      newID;
Handle    ctlHandle;
```

`SetCtlMoreFlags` $2F10

Sets the contents of the `ctlMoreFlags` field of the control record for a specified control. The `ctlMoreFlags` field contains flags governing target status, event processing, and other aspects of the control.

Retrieve the `ctlMoreFlags` field for a control with the `GetCtlMoreFlags` Control Manager call.

If the specified control is not an extended control, an error is returned.

Parameters

Stack before call

Previous contents	
newMoreFlags	Word—New `ctlMoreFlags` value for the control
– *ctlHandle* –	Long—Handle to control
	<—SP

Stack after call

Previous contents	
	<—SP

Errors			
	$1004	`noCtlError`	No controls in window
	$1007	`notSuperCtlError`	Action valid only for extended controls

C

```
extern pascal void SetCtlMoreFlags(newMoreFlags,
                    ctlHandle);

Word      newMoreFlags;
Handle    ctlHandle;
```

`SetCtlParamPtr` $3410

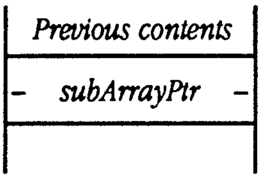
Sets the pointer to the current text substitution array for the Control Manager. This array contains the information used for text substitution in static text controls (see "Static text controls" elsewhere in this chapter).

Retrieve the contents of this field with the `GetCtlParamPtr` Control Manager tool call.

◆ *Note:* This pointer is global to the Control Manager; it is not associated with a specific control. As a result, desk accessories should be very careful when using this feature to save and restore the previous contents of the field.

Parameters

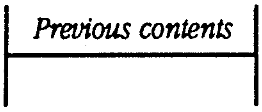
Stack before call



Long—New pointer to text substitution array

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void SetCtlParamPtr(subArrayPtr);

Pointer   subArrayPointer;
```

Control Manager error codes

Value	Name	Definition
$1001	`wmNotStartedUp`	Window Manager not initialized
$1002	`cmNotInitialized`	Control Manager not initialized
$1003	`noCtlInList`	Control not in window list
$1004	`noCtlError`	No controls in window
$1005	`noSuperCtlError`	No extended controls in window
$1006	`noCtlTargetError`	No target extended control.
$1007	`notSuperCtlError`	Action valid only for extended controls
$1008	`canNotBeTargetError`	Specified control cannot be made target
$1009	`noSuchIDError`	The specified ID cannot be found
$100A	`tooFewParmsError`	Too few parameters specified
$100B	`noCtlToBeTargetError`	No control could be made target
$100C	`noWind_Err`	There is no front window

New Control Manager templates and records

This section describes the format and content of all Control Manager control templates and records. In addition, "Control Manager code example" shows how to use control templates with the `NewControl2` tool call.

`NewControl2` input templates

Each type of control has its own control template, corresponding to the control record definition for the control type. The item template is an extensible mechanism for defining new controls. Rather than placing all the control parameters on the stack at run time, the template holds these parameters in a standard format that can be defined at compile time. Furthermore, the templates can be created as a resource, simplifying program development and maintenance, reducing code size, and reducing fixed memory usage. Your program can pass more than one input template to `NewControl2` at a time.

All control templates have the same seven-field header. Some templates have additional fields that further define the control. In order to provide extensible support for variable length templates, one of the header fields is a parameter count. The value of the parameter count field tells the Control Manager how many parameters to use, which allows for optional template fields.

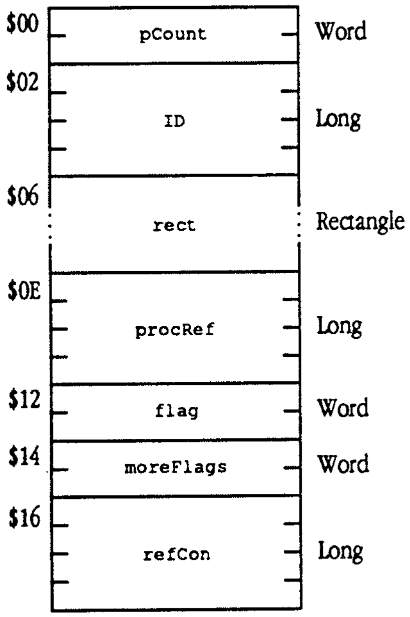
The following sections define the item templates for each control type. Field names marked with an asterisk (*) represent optional fields.

Control template standard header

Each control template contains the standard header, which consists of seven fields. Following that header, some templates have additional fields, which further define the control to be created. The format and content of the standard template header is shown in Figure 28-1.

Custom control definition procedures establish their own item template layout. The only restriction placed on these templates is that the standard header be present and well formed. Custom data for the control procedure may follow the standard header.

■ **Figure 28-1** Control template standard header



`pCount` Count of parameters in the item template, not including the `pCount` field. Minimum value is 6, maximum value varies depending upon the type of control template.

`ID` Sets the `ctlID` field of the control record for the new control. The `ctlID` field may be used by the application to provide a straightforward mechanism for keeping track of controls. The control ID is a value assigned by your application, which the control "carries around" for your convenience. Your application can use the ID, which has a known value, to identify a particular control.

`rect` Sets the `ctlRect` field of the control record for the new control. Defines the boundary rectangle for the control.

`procRef` Sets the `ctlProc` field of the control record for the new control. This field contains a reference to the control definition procedure for the control. The value of this field is either a pointer to a control definition procedure, or the ID of a standard routine. The standard values are:

`simpleButtonControl`	$80000000	Simple button
`checkControl`	$82000000	Check box
`iconButtonControl`	$07FF0001	Icon button
`editLineControl`	$83000000	LineEdit
`listControl`	$89000000	List
`pictureControl`	$8D000000	Picture
`popUpControl`	$87000000	Pop-up
`radioControl`	$84000000	Radio control
`scrollBarControl`	$86000000	Scroll bar
`growControl`	$88000000	Size box
`statTextControl`	$81000000	Static Text
`editTextControl`	$85000000	TextEdit

`flag` A word used to set both `ctlHilite` and `ctlFlag` in the control record for the new control. Since this is a word, the bytes for `ctlHilite` and `ctlFlag` are reversed. The high-order byte of `flag` contains `ctlHilite`, while the low-order byte contains `ctlFlag`. The bits in `flag` are mapped as follows:

Highlight	bits 8–15	Indicates highlighting style: 0 Control active, no highlighted parts 1–254 Part code of highlighted part 255 Control inactive
Invisible	bit 7	Governs visibility of control: 0 - Control visible 1 - Control invisible
Variable	bits 0–6	Values and meaning depends upon control type

`moreFlags` Used to set the `ctlMoreFlags` field of the control record for the new control.

The high-order byte is used by the Control Manager to store its own control information. The low-order byte is used by the control definition procedure to define reference types.

The defined Control Manager flags are:

`fCtlTarget`	$8000	If set to 1, this control is currently the target of any typing or editing commands.
`fCtlCanBeTarget`	$4000	If set to 1 then this control can be made the target control.
`fCtlWantEvents`	$2000	If set to 1 then this control can be called when events are passed via the `SendEventToCtl` Control Manager call. Note that, if the `fCtlCanBeTarget` flag is set to 1, this control will receive events sent to it regardless of setting of this flag.
`fCtlProcRefNotPtr`	$1000	If set to 1, then Control Manager expects `ctlProc` to contain the ID of a standard control procedure. If set to 0, then `ctlProc` contains a pointer to the custom control procedure.
`fCtlTellAboutSize`	$0800	If set to 1, then this control needs to be notified when the size of the owning window has changed. This flag allows custom control procedures to resize their associated control images in response to changes in window size.
`fCtlIsMultiPart`	$0400	If set to 1, then this is a multipart control. This flag allows control definition procedures to manage multi-part controls (necessary since the Control Manager does not know about all the parts of a multi-part control).

The low-order byte uses the following convention to describe references to color tables and titles (note, though, that some control templates do not follow this convention):

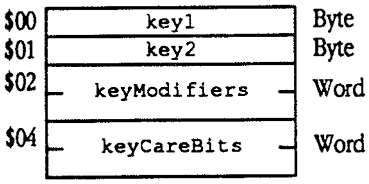
`titleIsPtr`	$00	Title reference is by pointer
`titleIsHandle`	$01	Title reference is by handle
`titleIsResource`	$02	Title reference is by resource ID
`colorTableIsPtr`	$00	Color table reference is by pointer
`colorTableIsHandle`	$04	Color table reference is by handle
`colorTableIsResource`	$08	Color table reference is by resource ID

`refCon` Used to set the `ctlRefCon` field of the control record for the new control. Reserved for application use.

Keystroke equivalent information

Many of these control templates allow you to specify keystroke equivalent information for the associated controls. Figure 28-2 shows the standard format for that keystroke information.

- **Figure 28-2** Keystroke equivalent record layout



`key1` This is the ASCII code for the upper or lower case of the key equivalent.

`key2` This is the ASCII code for the lower or upper case of the key equivalent. Taken with `key1`, this field completely defines the values against which key equivalents will be tested. If only a single key code is valid, then set `key1` and `key2` to the same value.

`keyModifiers` These are the modifiers that must be set to 1 in order for the equivalence test to pass. The format of this flag word corresponds to that defined for the event record in Chapter 7, "Event Manager," in Volume 1 of the *Toolbox Reference.* Note that only the modifiers in the high-order byte are used here.

`keyCareBits` These are the modifiers that must match for the equivalence test to pass. The format for this word corresponds to that for `keyModifiers`. This word allows you to discriminate between double-modified keystrokes. For example, if you want Control-7 to be an equivalent, but not Option-Control-7, you would set the *controlKey* bit in `keyModifiers` and both the *optionKey* and the *controlKey* bits in `keyCareBits` to 1. If you want Return and Enter to be treated the same, the *keyPad* bit should be set to 0.

Simple button control template

Figure 28-3 shows the template that defines a simple button control.

■ **Figure 28-3** Item template for simple button controls

$00	pCount	Word—Parameter count for template: 7, 8, or 9
$02	ID	Long—Application-assigned control ID
$06	rect	Rectangle—Boundary rectangle for control
$0E	procRef	Long—`simpleButtonControl`=$80000000
$12	flag	Word—Highlight and control flags for control
$14	moreFlags	Word—Additional control flags
$16	refCon	Long—Application-defined value
$1A	titleRef	Long—Reference to title for button
$1E	*colorTableRef	Long—Reference to color table for control (optional)
$22	*keyEquivalent	Block, 6 bytes—Keystroke equivalent data (optional)

Defined bits for `flag` are

Reserved	bits 8–15	Must be set to 0
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 2–6	Must be set to 0
Button type	bits 0–1	Describes button type: 0 = single-outlined round-cornered button 1 = bold-outlined round-cornered button 2 = single-outlined square-cornered button 3 = single-outlined square-cornered drop-shadowed button

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Set to 1 if button has keystroke equivalent
`fCtlProcNotPtr`	bit 12	Must be set to 1
fCtlTellAboutSize	bit 11	Must be set to 0
Reserved	bits 4–10	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in `colorTableRef`. See Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* for the definition of the simple button color table. 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Title reference	bits 0–1	Defines type of title reference in `titleRef`: 00 - title reference is pointer 01 - title reference is handle 10 - title reference is resource ID 11 - invalid value

`keyEquivalent` Keystroke equivalent information stored at `keyEquivalent` is formatted as shown in Figure 28–2.

Check box control template

Figure 28-4 shows the template that defines a check box control.

- **Figure 28-4** Control template for check box controls

$00	`pCount`	Word—Parameter count for template: 8, 9, or 10
$02	`ID`	Long—Application-assigned control ID
$06	`rect`	Rectangle—Boundary rectangle for control
$0E	`procRef`	Long— `checkBoxControl` =$82000000
$12	`flag`	Word—Highlight and control flags for control
$14	`moreFlags`	Word—Additional control flags
$16	`refCon`	Long—Application-defined value
$1A	`titleRef`	Long—Reference to title for box
$1E	`initialValue`	Word—Initial box setting: 0 for clear, 1 for checked
$20	`*colorTableRef`	Long—Reference to color table for control (optional)
$24	`*keyEquivalent`	Block, 6 bytes—Keystroke equivalent data (optional)

Defined bits for `flag` are

Reserved	bits 8–15	Must be set to 0
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 0–6	Must be set to 0

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Set to 1 if check box has keystroke equivalent
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 4–10	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in `colorTableRef` (see Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* for the definition of the check box color table) 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Title reference	bits 0–1	Defines type of title reference in `titleRef`: 00 - title reference is pointer 01 - title reference is handle 10 - title reference is resource ID 11 - invalid value

`keyEquivalent` Keystroke equivalent information stored at `keyEquivalent` is formatted as shown in Figure 28-2.

Icon button control template

Figure 28-5 shows the template that defines an icon button control. For more information about icon button controls, see "Icon button control" in this chapter.

- **Figure 28-5** Control template for icon button controls

$00	`pCount`	Word—Parameter count for template: 7, 8, 9, 10, or 11
$02	`ID`	Long—Application-assigned control ID
$06	`rect`	Rectangle—Boundary rectangle for control
$0E	`procRef`	Long—`iconButtonControl` =$07FF0001
$12	`flag`	Word—Highlight and control flags for control
$14	`moreFlags`	Word—Additional control flags
$16	`refCon`	Long—Application-defined value
$1A	`iconRef`	Long—Reference to icon for control
$1E	`*titleRef`	Long—Reference to title for control (optional)
$22	`*colorTableRef`	Long—Reference to color table for control (optional)
$26	`*displayMode`	Word—Bit flag controlling icon appearance (optional)
$28	`*keyEquivalent`	Block, 6 bytes—Key equivalent information (optional)

Defined bits for `flag` are

`ctlHilite`	bits 8–15	Sets the `ctlHilite` field of the control record
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 3–6	Must be set to 0
`showBorder`	bit 2	1=No border, 0=Show border
`buttonType`	bits 0–1	Defines button type: 00 - single-outlined round-cornered button 01 - bold-outlined round-cornered button 10 - single-outlined square-cornered button 11 - single-outlined square-comered and drop-shadowed button

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 0
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 6–10	Must be set to 0
Icon reference	bits 4–5	Defines type of icon reference in `iconRef`: 00 - icon reference is pointer 01 - icon reference is handle 10 - icon reference is resource ID 11 - invalid value
Color table reference	bits 2–3	Defines type of reference in `colorTableRef`; the color table for an icon button is the same as that for a simple button (see Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* for the definition of the simple button color table) 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Title reference	bits 0–1	Defines type of title reference in `titleRef`: 00 - title reference is pointer 01 - title reference is handle 10 - title reference is resource ID 11 - invalid value

`titleRef` Reference to the title string, which must be a Pascal string. If you are not using a title but are specifying other optional fields, set `moreFlags` bits 0 and 1 to 0, and set this field to zero.

`displayMode` Passed directly to the `DrawIcon` routine, and defines the display mode for the icon. The field is defined as follows (for more information on icons, see Chapter 17, "QuickDraw II Auxiliary," in Volume 2 of the *Toolbox Reference*):

Background Color	bits 12–15	Defines the background color to apply to black part of black-and-white icons.
Foreground Color	bits 8–11	Defines the foreground color to apply to white part of black-and-white icons.
Reserved	bits 3–7	Must be set to 0
`offLine`	bit 2	1=AND light-gray pattern to image being copied 0=Don't AND the image
`openIcon`	bit 1	1=Copy light-gray pattern instead of image 0=Don't copy light-gray pattern
`selectedIcon`	bit 0	1=Invert image before copying 0=Don't invert image

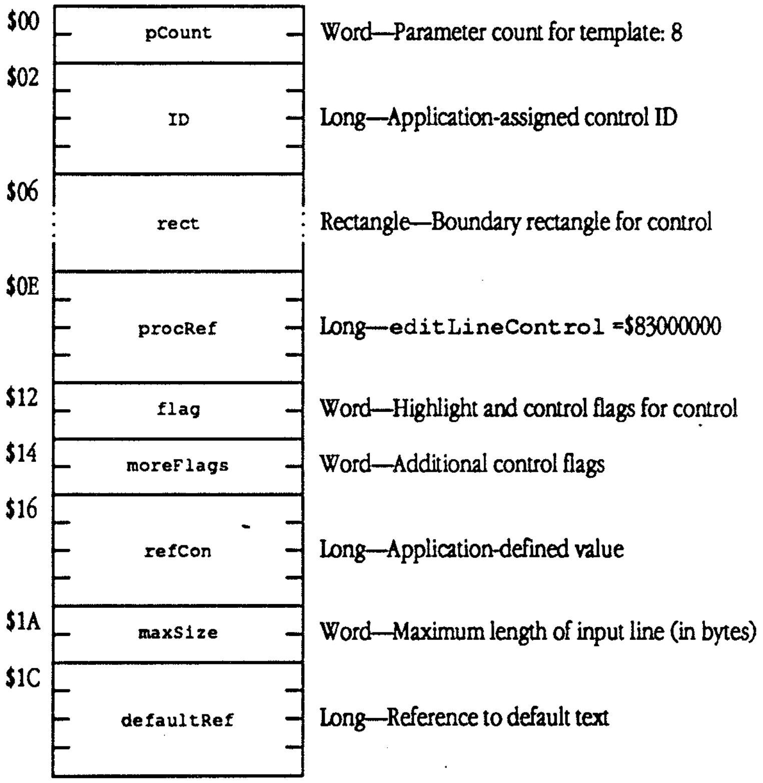
Color values (both foreground and background) are indexes into the current color table. See Chapter 16, "QuickDraw II," in Volume 2 of the *Toolbox Reference* for details about the format and content of these color tables.

`keyEquivalent` Keystroke equivalent information stored at `keyEquivalent` is formatted as shown in Figure 28-2.

LineEdit control template

Figure 28-6 shows the template that defines a LineEdit control. For more information about LineEdit controls, see "LineEdit control" in this chapter.

- **Figure 28-6** Control template for LineEdit controls



$00	`pCount`	Word—Parameter count for template: 8
$02	`ID`	Long—Application-assigned control ID
$06	`rect`	Rectangle—Boundary rectangle for control
$0E	`procRef`	Long—`editLineControl` =$83000000
$12	`flag`	Word—Highlight and control flags for control
$14	`moreFlags`	Word—Additional control flags
$16	`refCon`	Long—Application-defined value
$1A	`maxSize`	Word—Maximum length of input line (in bytes)
$1C	`defaultRef`	Long—Reference to default text

Defined bits for `flag` are

Reserved	bits 8–15	Must be set to 0
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 0–6	Must be set to 0

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 1
`fCtlWantsEvents`	bit 13	Must be set to 1
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 2–10	Must be set to 0
Text reference	bits 0–1	Defines type of text reference in `defaultRef` 00 - text reference is pointer 01 - text reference is handle 10 - text reference is resource ID 11 - invalid value

`maxSize` Specifies the maximum number of characters allowed in the LineEdit field. Valid values lie in the range from 1 to 255.

The high-order bit indicates whether the LineEdit field is a password field. Password fields protect user input by echoing asterisks, rather than the actual user input. If this bit is set to 1, then the LineEdit field is a password field.

Note that LineEdit controls do not support color tables.

List control template

Figure 28-7 shows the template that defines a list control. For more information about list controls, see "List control" in this chapter.

- **Figure 28-7** Control template for list controls

$00	`pCount`	Word—Parameter count for template: 14 or 15
$02	`ID`	Long—Application-assigned control ID
$06	`rect`	Rectangle—Boundary rectangle for control
$0E	`procRef`	($0E) Long—`listControl` =$89000000
$12	`flag`	Word—Highlight and control flags for control
$14	`moreFlags`	Word—Additional control flags
$16	`refCon`	Long—Application-defined value
$1A	`listSize`	Word—Number of members in list
$1C	`listView`	Word—Number of members visible in window
$1E	`listType`	Word—Type of list entries, selection options, etc.
$20	`listStart`	Word—First visible list member
$22	`listDraw`	Long—Pointer to member drawing routine
$26	`listMemHeight`	Word—Height of each list item (in pixels)
$28	`listMemSize`	Word—Size of list entry (in bytes)
$2A	`listRef`	Long—Reference to list of member records
$2E	`*colorTableRef`	Long—Reference to color table for control (optional)

Defined bits for `flag` are

Reserved	bits 8–15	Must be set to 0
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 0–6	Must be set to 0

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 0
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
`fCtlIsMultiPart`	bit 10	Must be set to 1
Reserved	bits 4–9	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in `colorTableRef` (the color table for a List control is described in Chapter 11, "List Manager," in Volume 1 of the *Toolbox Reference*) 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
List reference	bits 0–1	Defines type of reference in `listRef` (the format for a list member record is described in Chapter 11, "List Manager," in Volume 1 of the *Toolbox Reference*) 00 - list reference is pointer 01 - list reference is handle 10 - list reference is resource ID 11 - invalid value

`listType` Valid values for `listType` are as follows:

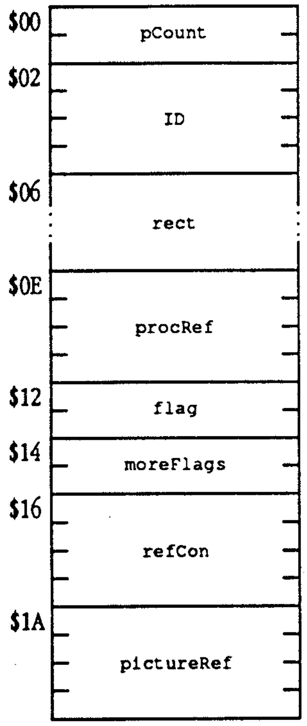
Reserved	bits3–15	Must be set to 0.
`fListScrollBar`	bit 2	Allows you to control where the scroll bar for the list is drawn: 1 - Scroll bar drawn on inside of boundary rectangle. The List Manager calculates space needed, adjusts dimensions of boundary rectangle, and resets this flag. 0 - Scroll bar drawn on outside of boundary rectangle.
`fListSelect`	bit 1	Controls type of selection options available to the user: 1 - Only single selection allowed 0 - Arbitrary and range selection allowed
`fListString`	bit 0	Defines the type of strings used to define list items: 1 - C-strings ($00-terminated) 0 - Pascal strings

For details on the remaining custom fields in this template, see the discussion of "List Controls and List Records" in Chapter 11, "List Manager," of Volume 1 of the *Toolbox Reference.*

Picture control template

Figure 28-8 shows the template that defines a picture control. For more information about picture controls, see "Picture control" in this chapter.

- **Figure 28-8** Control template for picture controls



Offset	Field	Description
$00	`pCount`	Word—Parameter count for template: 7
$02	`ID`	Long—Application-assigned control ID
$06	`rect`	Rectangle—Boundary rectangle for control
$0E	`procRef`	Long—`pictureControl` =$8D000000
$12	`flag`	Word—Highlight and control flags for control
$14	`moreFlags`	Word—Additional control flags
$16	`refCon`	Long—Application-defined value
$1A	`pictureRef`	Long—Reference to picture for control

Defined bits for `flag` are

`ctlHilite`	bits 8–15	Specifies whether the control wants to receive mouse selection events; the values for `ctlHilite` are as follows: 0 Control is active 255 Control is inactive
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 0–6	Must be set to 0

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 0
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 2–10	Must be set to 0
Picture reference	bits 0–1	Define type of picture reference in `pictureRef`: 00 - invalid value 01 - reference is handle 10 - reference is resource ID 11 - invalid value

Pop-up control template

Figure 28-9 shows the template that defines a pop-up control. For more information about pop-up controls, see "Pop-up control" in this chapter.

- **Figure 28-9** Control template for pop-up controls

$00	pCount	Word—Parameter count for template: 9 or 10
$02	ID	Long—Application-assigned control ID
$06	rect	Rectangle—Boundary rectangle for control
$0E	procRef	Long—popUpControl=$87000000
$12	flag	Word—Highlight and control flags for control
$14	moreFlags	Word—Additional control flags
$16	refCon	Long—Application-defined value
$1A	titleWidth	Word—Width in pixels of title string area
$1C	menuRef	Long—Reference to menu definition
$20	initialValue	Word—Item ID of initial item
$22	*colorTableRef	Long—Reference to color table for control (optional)

Defined bits for `flag` are

`ctlHilite`	bits 8–15	Specifies whether the control wants to receive mouse selection events; the values for `ctlHilite` are as follows: 0 Control is active 255 Control is inactive
`ctlInvis`	bit 7	1=invisible, 0=visible
`fType2PopUp`	bit 6	Tells the Control Manager whether to create a pop-up menu with white space for scrolling (see Chapter 37, "Menu Manager Update," for details on Type 2 pop-up menus): 1 - Draw pop-up with white space (Type 2) 0 - Draw normal pop-up
`fDontHiliteTitle`	bit 5	Controls highlighting of the control title: 1 - Do not highlight title when control is popped up 0 - Highlight title
`fDontDrawTitle`	bit 4	Allows you to prevent the title from being drawn (note that you must supply a title in the menu definition, whether or not it will be displayed); if `titleWidth` is defined and this bit is set to 1, then the entire menu is offset to the right by `titleWidth` pixels: 1 - Do not draw the title 0 - Draw the title
`fDontDrawResult`	bit 3	Allows you to control whether the selection is drawn in the pop-up rectangle: 1 - Do not draw the result in the result area after a selection 0 - Draw the result
`fInWindowOnly`	bit 2	Controls the extent to which the pop-up menu can grow; this is particularly relevant to Type 2 pop-ups (see Chapter 37, "Menu Manager Update," for details on Type 2 pop-up menus): 1 - Keep the pop-up in the current window 0 - Allow the pop-up to grow to screen size

`fRightJustifyTitle`	bit 1	Controls title justification: 1 - Right justify the title; note that if the title is right justified, then the control rectangle is adjusted to eliminate unneeded pixels (see Figure 28-12), the value for `titleWidth` is also adjusted 0 - Left justify the title
`fRightJustifyResult`	bit 0	Controls result justification: 1 - Right justify the selection 0 - Left justify the selection `titleWidth` pixels from the left of the pop-up rectangle.

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 1 if the pop-up has any keystroke equivalents defined
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 5–10	Must be set to 0
Color table reference	bits 3–4	Defines type of reference in `colorTableRef` (the color table for a menu is described in Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference*) 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
`fMenuDefIsText`	bit 2	Defines type of data referred to by `menuRef`: 1 - `menuRef` is a pointer to a text stream in `NewMenu` format (see Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference* for details) 0 - `menuRef` is a reference to a Menu Template (again, see Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference* for details on format and content of a Menu Template)

Menu reference bits 0–1 Defines type of menu reference in `menuRef` (if `fMenuDefIsText` is set to 1, then these bits are ignored):
00 - menu reference is pointer
01 - menu reference is handle
10 - menu reference is resource ID
11 - invalid value

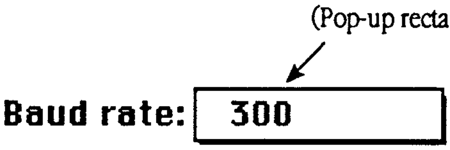
`rect` Defines the boundary rectangle for the pop-up and its title, before the menu has been "popped" by the user. The Menu Manager will calculate the lower, right coordinates of the rectangle for you, if you specify those coordinates as (0,0).

`initialValue` The initial value to be displayed for the menu. The initial value is the default value for the menu, and is displayed in the pop-up rectangle of "unpopped" menus. You specify an item by its ID, that is, its relative position within the array of items for the menu (see Chapter 37, "Menu Manager Update," for information on the layout and content of the pop-up menu template). If you pass an invalid item ID then no item is displayed in the pop-up rectangle.

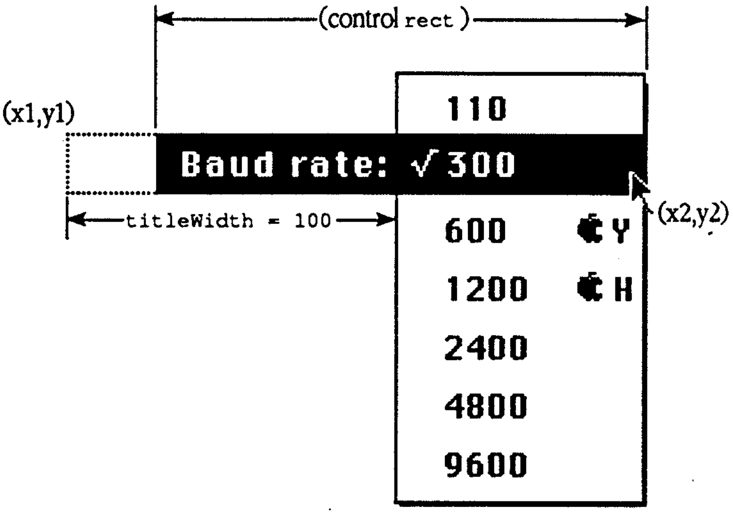
`titleWidth` Provides you with additional control over placement of the menu on the screen. The `titleWidth` field defines an offset from the left edge of the control (boundary) rectangle to the left edge of the pop-up rectangle (see Figure 28-11). If you are creating a series of pop-up menus and you want them to be vertically aligned, you can do this by giving all menus the same x1 coordinate and `titleWidth` value. You may use `titleWidth` for this even if you are not going to display the title (`fDontDrawTitle` flag is set to 1 in `flag`). If you set `titleWidth` to 0, then the Menu Manager determines its value based upon the length of the menu title, and the pop-up rectangle immediately follows the title string. If the actual width of your title exceeds the value of `titleWidth`, results are unpredictable.

`menuRef` Reference to menu definition (see Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference* and Chapter 37, "Menu Manager Update," in this book for details on menu templates). The type of reference contained in `menuRef` is defined by the menu reference bits in `moreFlags`.

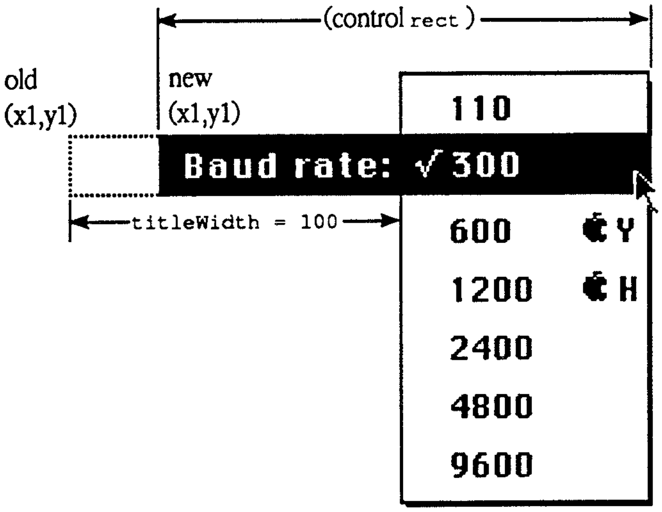
- **Figure 28-10** "Unpopped" pop-up menu



- **Figure 28-11** "Popped" pop-up menu, left-justified title



- **Figure 28-12** "Popped" pop-up menu, right-justified title



Radio button control template

Figure 28-13 shows the template that defines a radio button control:

- **Figure 28-13** Control template for radio button controls

$00	`pCount`	Word—Parameter count for template: 8, 9, or 10
$02	`ID`	Long—Application-assigned control ID
$06	`rect`	Rectangle—Boundary rectangle for control
$0E	`procRef`	Long— `radioButtonControl` =$84000000
$12	`flag`	Word—Highlight and control flags for control
$14	`moreFlags`	Word—Additional control flags
$16	`refCon`	Long—Application-defined value
$1A	`titleRef`	Long—Reference to title for button
$1E	`initialValue`	Word—Initial setting: 0 for clear, 1 for set
$20	`*colorTableRef`	Long—Reference to color table for control (optional)
$24	`*keyEquivalent`	Block, 6 bytes—Keystroke equivalent data (optional)

Defined bits for `flag` are

Reserved	bits 8–15	Must be set to 0
`ctlInvis`	bit 7	1=invisible, 0=visible
Family number	bits 0–6	Family numbers define associated groups of radio buttons; radio buttons in the same family are logically linked, that is, setting one radio button in a family clears all other buttons in the same family

Defined bits for `moreFlags` are as follows:

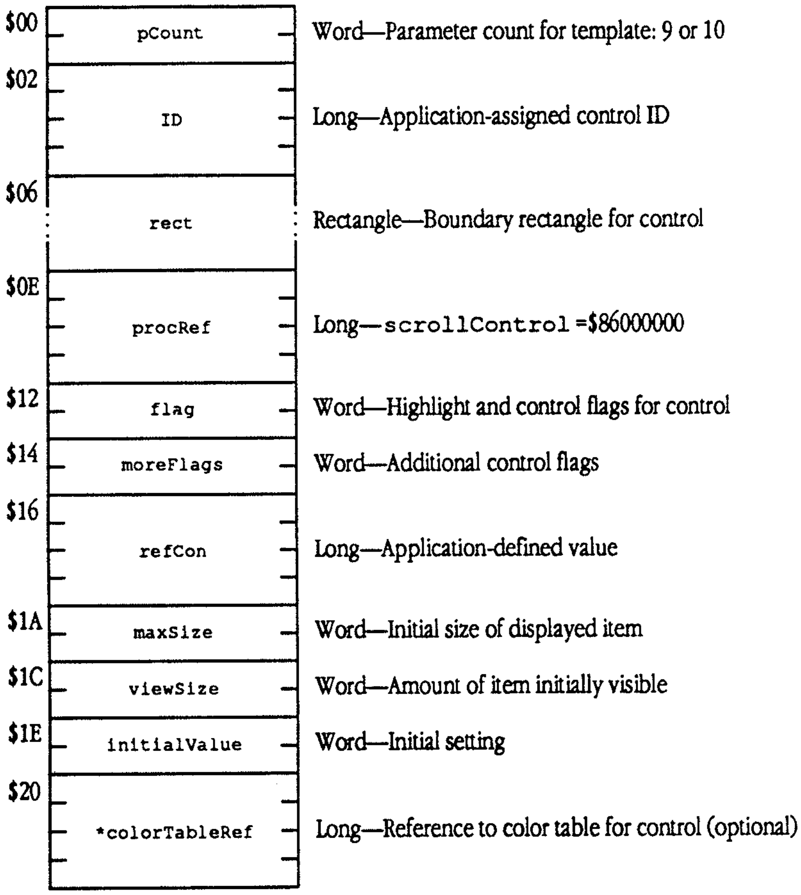
`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Set to 1 if button has keystroke equivalent
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 4–10	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in `colorTableRef` (see Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* for the definition of the radio button color table) 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Title reference	bits 0–1	Defines type of title reference in `titleRef`: 00 - title reference is pointer 01 - title reference is handle 10 - title reference is resource ID 11 - invalid value

`keyEquivalent` Keystroke equivalent information stored at `keyEquivalent` is formatted as shown in Figure 28-2.

Scroll bar control template

Figure 28-14 shows the template that defines a scroll bar control:

- **Figure 28-14** Control template for scroll bar controls



$00	pCount	Word—Parameter count for template: 9 or 10
$02	ID	Long—Application-assigned control ID
$06	rect	Rectangle—Boundary rectangle for control
$0E	procRef	Long—`scrollControl` =$86000000
$12	flag	Word—Highlight and control flags for control
$14	moreFlags	Word—Additional control flags
$16	refCon	Long—Application-defined value
$1A	maxSize	Word—Initial size of displayed item
$1C	viewSize	Word—Amount of item initially visible
$1E	initialValue	Word—Initial setting
$20	*colorTableRef	Long—Reference to color table for control (optional)

Defined bits for `flag` are

Reserved	bits 8–15	Must be set to 0
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 5–6	Must be set to 0
`horScroll`	bit 4	1=horizontal scroll bar, 0=vertical scroll bar
`rightFlag`	bit 3	1=bar has right arrow, 0=bar has no right arrow
`leftFlag`	bit 2	1=bar has left arrow, 0=bar has no left arrow
`downFlag`	bit 1	1=bar has down arrow, 0=bar has no down arrow
`upFlag`	bit 0	1=bar has up arrow, 0=bar has no up arrow

Note that extraneous flag bits are ignored, based upon state of `horScroll` flag. For example, for vertical scroll bars, `rightFlag` and `leftFlag` are ignored.

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 0
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 4–10	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in *colorTableRef* (see Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* and "Clarifications" earlier in this chapter for the definition of the scroll bar color table) 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Reserved	bits 0–1	Must be set to 0

Size box control template

Figure 28-15 shows the template that defines a size box control:

- **Figure 28-15** Control template for size box controls

$00	`pCount`	Word—Parameter count for template: 6 or 7
$02	`ID`	Long—Application-assigned control ID
$06	`rect`	Rectangle—Boundary rectangle for control
$0E	`procRef`	Long— `growControl`=$88000000
$12	`flag`	Word—Highlight and control flags for control
$14	`moreFlags`	Word—Additional control flags
$16	`refCon`	Long—Application-defined value
$20	`*colorTableRef`	Long—Reference to color table for control (optional)

Defined bits for `flag` are

Reserved	bits 8–15	Must be set to 0
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 1–6	Must be set to 0
`fCallWindowMgr`	bit 0	1=call `GrowWindow` and `SizeWindow` to track this control 0=just highlight control

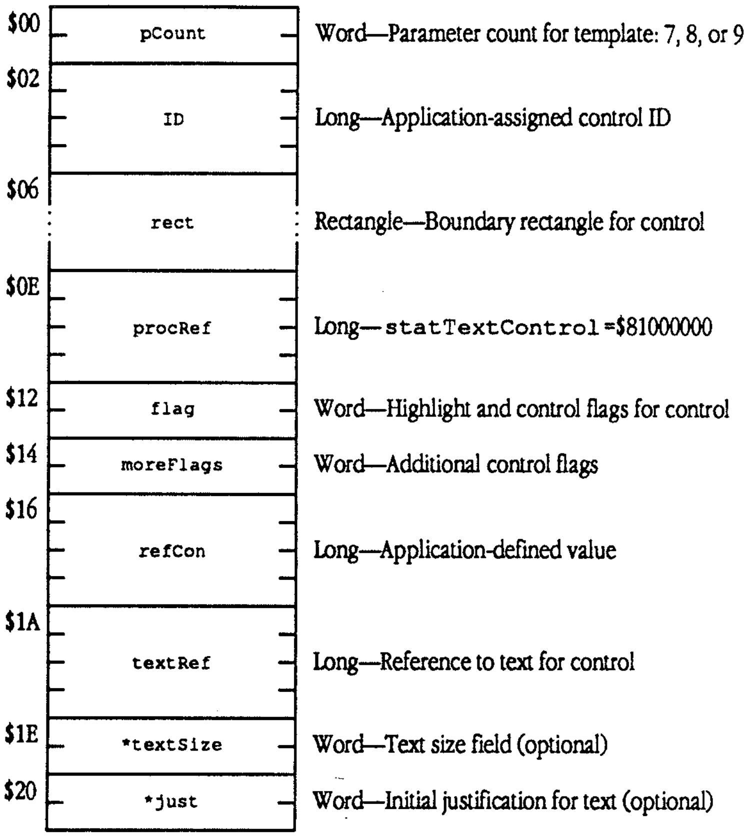
Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 0
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 4–10	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in `colorTableRef` (see "Error Corrections" earlier in this chapter for the definition of the size box color table) 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Reserved	bits 0–1	Must be set to 0

Static text control template

Figure 28-16 shows the template that defines a static text control. For more information about static text controls, see "Static text control" in this chapter.

- **Figure 28-16** Control template for static text controls



$00	`pCount`	Word—Parameter count for template: 7, 8, or 9
$02	`ID`	Long—Application-assigned control ID
$06	`rect`	Rectangle—Boundary rectangle for control
$0E	`procRef`	Long—`statTextControl`=$81000000
$12	`flag`	Word—Highlight and control flags for control
$14	`moreFlags`	Word—Additional control flags
$16	`refCon`	Long—Application-defined value
$1A	`textRef`	Long—Reference to text for control
$1E	`*textSize`	Word—Text size field (optional)
$20	`*just`	Word—Initial justification for text (optional)

Defined bits for `flag` are

Reserved	bits 8–15	Must be set to 0
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 2–6	Must be set to 0
`fSubstituteText`	bit 1	0=no text substitution to perform 1=there is text substitution to perform
`fSubTextType`	bit 0	0=C strings 1=Pascal strings

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 0
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 2–10	Must be set to 0
Text Reference	bits 0–1	Defines type of text reference in `textRef`: 00 - text reference is pointer 01 - text reference is handle 10 - text reference is resource ID 11 - invalid value

`textSize` The size of the referenced text in characters, but only if the text reference in `textRef` is a pointer. If the text reference is either a handle or a resource ID, then the Control Manager can extract the length from the handle.

`just` The justification word is passed on to `LETextBox2` (see Chapter 10, "LineEdit Tool Set," in Volume 1 of the *Toolbox Reference* for details on the `LETextBox2` tool call), and is used to set the initial justification for the text being drawn. Valid values for `just` are

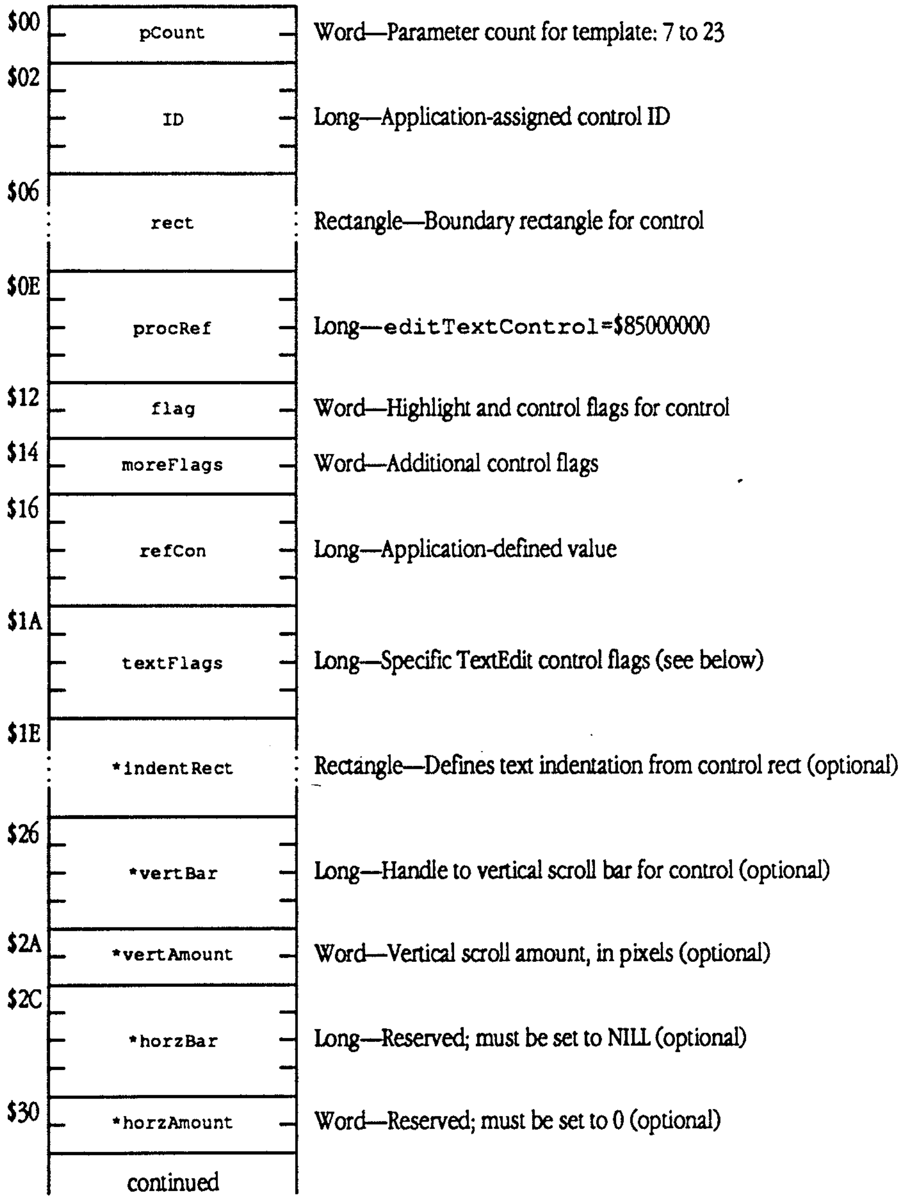
`leftJustify`	0	Text is left justified in the display window
`centerJustify`	1	Text is centered in the display window
`rightJustify`	-1	Text is right justified in the display window
`fullJustify`	2	Text is fully justified (both left and right) in the display window

Static text controls do not support color tables. In order to display text of different color, you must embed the appropriate commands into the text string you are displaying. See the discussion of `LETextBox2` in Chapter 10, "LineEdit Tool Set," in Volume 1 of the *Toolbox Reference* for details on command format and syntax.

TextEdit control template

Figure 28-17 shows the template that defines a TextEdit control. For more information about TextEdit controls, see "TextEdit control" in this chapter.

- **Figure 28-17** Control template for TextEdit controls



$00	`pCount`	Word—Parameter count for template: 7 to 23
$02	`ID`	Long—Application-assigned control ID
$06	`rect`	Rectangle—Boundary rectangle for control
$0E	`procRef`	Long—`editTextControl`=$85000000
$12	`flag`	Word—Highlight and control flags for control
$14	`moreFlags`	Word—Additional control flags
$16	`refCon`	Long—Application-defined value
$1A	`textFlags`	Long—Specific TextEdit control flags (see below)
$1E	`*indentRect`	Rectangle—Defines text indentation from control rect (optional)
$26	`*vertBar`	Long—Handle to vertical scroll bar for control (optional)
$2A	`*vertAmount`	Word—Vertical scroll amount, in pixels (optional)
$2C	`*horzBar`	Long—Reserved; must be set to NIL (optional)
$30	`*horzAmount`	Word—Reserved; must be set to 0 (optional)
	continued	

	continued	
$32	*styleRef	Long—Reference to initial style information for text (optional)
$36	*textDescriptor	Word—Defines format of initial text and `textRef` (optional)
$38	*textRef	Long—Reference to initial text for edit window (optional)
$3C	*textLength	Long—Length of initial text (optional)
$40	*maxChars	Long—Maximum number of characters allowed (optional)
$44	*maxLines	Long—Reserved; must be set to 0 (optional)
$48	*maxCharsPerLines	Word—Reserved; must be set to 0 (optional)
$4A	*maxHeight	Word—Reserved; must be set to 0 (optional)
$4C	*colorRef	Long—Reference to TextEdit color table (optional)
$50	*drawMode	Word—QuickDraw II text mode for edit window (optional)
$52	*filterProcPtr	Long—Pointer to filter routine for this control (optional)

Defined bits for `flag` are

Reserved	bits 8–15	Must be set to 0
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 0–6	Must be set to 0

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 1
`fCtlWantsEvents`	bit 13	Must be set to 1
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fTellAboutSize`	bit 11	If set to 1, a size box will be created in the lower-right corner of the window. Whenever the control window is resized, the edit text will be resized and redrawn.
`fCtlIsMultiPart`	bit 10	Must be set to 1
Reserved	bits 4–9	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in `colorRef`; the color table for a TextEdit control (`TEColorTable`) is described in Chapter 49, "TextEdit," in this book: 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Style reference	bits 0–1	Defines type of style reference in `styleRef`; the format for a TextEdit style descriptor is described in Chapter 49, "TextEdit," in this book: 00 - style reference is pointer 01 - style reference is handle 10 - style reference is resource ID 11 - invalid value

△ **Important** Do not set `fTellAboutSize` to 1 unless the control also has a vertical scroll bar. △

Valid values for `textFlags` are

`fNotControl`	bit 31	Must be set to 0
`fSingleFormat`	bit 30	Must be set to 1
`fSingleStyle`	bit 29	Allows you to restrict the style options available to the user: 1 - Allow only one style in the text 0 - Do not restrict the number of styles in the text

`fNoWordWrap`	bit 28	Allows you to control TextEdit word wrap behavior: 1 - Do not word wrap the text; only break lines on CR ($0D) characters 0 - Perform word wrap to fit the ruler
`fNoScroll`	bit 27	Controls user access to scrolling: 1 - Do not allow either manual or auto-scrolling 0 - Scrolling permitted
`fReadOnly`	bit 26	Restricts the text in the window to read-only operations (copying from the window will still be allowed): 1 - No editing allowed 0 - Editing permitted
`fSmartCutPaste`	bit 25	Controls TextEdit support for smart cut and paste (see Chapter 49, "TextEdit," for details on smart cut and paste support): 1 - Use smart cut and paste 0 - Do not use smart cut and paste
`fTabSwitch`	bit 24	Defines behavior of the Tab key (see Chapter 49, "TextEdit," for details): 1 - Tab to next control in the window 0 - Tab inserted in TextEdit document
`fDrawBounds`	bit 23	Tells TextEdit whether to draw a box around the edit window, just inside `rect`; the pen for this box is two pixels wide and one pixel high 1 - Draw rectangle 0 - Do not draw rectangle
`fColorHilight`	bit 22	Must be set to 0.
`fGrowRuler`	bit 21	Tells TextEdit whether to resize the ruler in response to the user resizing the edit window; if set to 1, TextEdit will automatically adjust the right margin value for the ruler: 1 - Resize the ruler 0 - Do not resize the ruler
`fDisableSelection`	bit 20	Controls whether user can select text: 1 - User cannot select text 0 - User can select text

`fDrawInactiveSelection`	bit 19	Controls how inactive selected text is displayed: 1 - TextEdit draws a box around inactive selections 0 - TextEdit does not display inactive selections
Reserved	bits 0–18	Must be set to 0

`indentRect` Each coordinate of this rectangle specifies the amount of white space to leave between the boundary rectangle for the control and the text itself, in pixels. Default values are (2,6,2,4) in 640 mode and (2,4,2,2) in 320 mode. Each indentation coordinate may be specified individually. In order to assert the default for any coordinate, specify its value as $FFFF.

`vertBar` Handle of the vertical scroll bar to use for the TextEdit window. If you do not want a scroll bar at all, then set this field to NIL. If you want TextEdit to create a scroll bar for you, just inside the right edge of the boundary rectangle for the control, then set this field to $FFFFFFFF.

`vertAmount` Specifies the number of pixels to scroll whenever the user presses the up or down arrow on the vertical scroll bar. In order to use the default value (9 pixels), set this field to $0000.

`horzBar` Must be set to NIL.

`horzAmount` Must be set to 0.

`styleRef` Reference to initial style information for the text. See the description of the `TEFormat` record in Chapter 49, "TextEdit," for information about the format and content of a style descriptor. Bits 1 and 0 of `moreFlags` define the type of reference (pointer, handle, resource ID). To use the default style and ruler information, set this field to NULL.

`textDescriptor` Input text descriptor that defines the reference type for the initial text (which is in `textRef`) and the format of that text. See Chapter 49, "TextEdit," for detailed information on text and reference formats.

`textRef` Reference to initial text for the edit window. If you are not supplying any initial text, then set this field to NULL.

`textLength` If `textRef` is a pointer to the initial text, then this field must contain the length of the initial text. For other reference types, TextEdit extracts the length from the reference itself.

◆ *Note:* You must specify or omit the `textDescriptor`, `textRef`, and `textLength` fields as a group.

`maxChars` Maximum number of characters allowed in the text. If you do not want to define any limit to the number of characters, then set this field to NULL.

`maxLines` Must be set to 0.

`maxCharsPerLines`
Must be set to NULL.

`maxHeight` Must be set to 0.

`colorRef` Reference to the color table for the text. This is a Text Edit color table (see Chapter 49, "TextEdit," for format and content of `TEColorTable`). Bits 2 and 3 of `moreFlags` define the type of reference stored here.

`drawMode` This is the text mode used by QuickDraw II for drawing text. See Chapter 16, "QuickDraw II," in Volume 2 of the *Toolbox Reference* for details on valid text modes.

`filterProcPtr` Pointer to a filter routine for the control. See Chapter 49, "TextEdit," for details on TextEdit generic filter routines. If you do not want to use a filter routine for the control, set this field to NIL.

Control Manager code example

This section contains an example of how to create a list of controls for a window with a single `NewControl2` call. If you wish to try this in your own program, you will need to create a window that is 160 lines high and 600 pixels wide.

```
; Equates for the new control manager features
; ctlMoreFlags
;
fCtlTarget              equ $8000
fCtlCanBeTarget         equ $4000
fCtlWantEvents          equ $2000
fCtlProcRefNotPtr       equ $1000
fCtlTellAboutSize       equ $0800
titleIsPtr              equ $0000
titleIsHandle           equ $0001
titleIsResource         equ $0002
colorTableIsPtr         equ $0000
colorTableIsHandle      equ $0004
colorTableIsResource    equ $0008
;
; NewControl2 ProcRef values for standard control types
;
simpleButtonControl     equ $80000000
checkControl            equ $82000000
radioControl            equ $84000000
scrollBarControl        equ $86000000
growControl             equ $88000000
statTextControl         equ $81000000
editLineControl         equ $83000000
editTextControl         equ $85000000
popUpControl            equ $87000000
listControl             equ $89000000
iconButtonControl       equ $07FF0001
pictureControl          equ $8D000000
```

```
;
; Here is the definition of my control list, note it is simply a list
;  of pointers. These do not have to be in any special order. This list
;  should always be terminated with a zero.
;
MyControls  dc.L theButton,theScroll,theCheck
            dc.L Radio1,Radio2,StatControl
            dc.L LEditControl,PopUp,IconButton,0

; Scroll bar color table as defined by the original control manager.
;  The structure of these tables has not changed for the existing
;  control types.
;
MyColorTable
            dc.W 0                   ; outline color
            dc.W $00F0               ; arrow unhilited black on
                                     ;  white
            dc.W $0005               ; arrow hilite blue on black
            dc.W $00F0               ; arrow Background color
            dc.W $00F0               ; Thumb unhilited
            dc.W $0000               ; Thumb hilited
            dc.W $0030               ; Page region soIid
                                     ;  black/white
            dc.W $00F0               ; Inactive bar color
;
; Definition of a simple vertical scroll bar
;
theScroll   dc.W 10                  ; number of parms
            dc.L 1                   ; application ID
            dc.W 10,10,110,36        ; rectangle
            dc.L scrollBarControl    ; scrollbar def proc
            dc.W 3                   ; vertical scroll bar w/
                                     ;  arrows
            dc.W fCtlProcRefNotPtr   ; set procnotptr flag
            dc.L 0                   ; refcon
            dc.W 100                 ; max size
            dc.W 10                  ; size of view
            dc.W 5                   ; initial value
            dc.L MyColorTable        ; color table to use
```

```
;
; Definition of a simple button
;
SimpTitle   str 'Button'
theButton   dc.W 7                  ; num params
            dc.L 2                  ; app ID
            dc.W 10,40,0,0          ; a 25x30 button
            dc.L simpleButtonControl      ; simple button
            dc.W 0                  ; visible, round corner
            dc.W fCtlProcRefNotPtr+fCtlWantEvents
            dc.L 0
            dc.L simpTitle          ; button Title
;
; Definition of a check box control
;
CheckTitle  str 'CheckBox'
theCheck    dc.W 8                  ; num params
            dc.L 3                  ; app ID
            dc.W 25,40,0,0          ; bounding rect
            dc.L checkControl       ; control type
            dc.W 0                  ; flags
            dc.W fCtlProcRefNotPtr  ; MoreFlags
            dc.L 0                  ; RefCon
            dc.L CheckTitle         ; TitlePointer
            dc.W 0
;
; Definition of a radio button control
;
Radio1Title str 'Radio1'
Radio1      dc.W 8
            dc.L 4
            dc.W 45,40,0,0
            dc.L radioControl
            dc.W 1
            dc.W fCtlProcRefNotPtr
            dc.L 0
            dc.L Radio1Title
            dc.W 1
```

```
;
; Definition of another radio button control
;
Radio2Title str 'Radio2'
Radio2      dc.W 8
            dc.L 5
            dc.W 65,40,0,0
            dc.L radioControl
            dc.W 1
            dc.W fCtlProcRefNotPtr
            dc.L 0
            dc.L Radio2Title
            dc.W 0
;
; Definition of a static text control
;
StatTitle   dc.B 'This is Stat Text'
StatControl dc.W 8
            dc.L 6
            dc.W 120,10,135,210
            dc.L statTextControl
            dc.W 0
            dc.W fCtlProcRefNotPtr
            dc.L 0
            dc.L StatTitle
            dc.W 17
;
; Definition of an edit line control
;
EditDefault str 'DefaultText'
LEditControl
            dc.W 8
            dc.L 7
            dc.W 120,240,135,440
            dc.L editLineControl
            dc.W 0
            dc.W fCtlProcRefNotPtr
            dc.L 0
            dc.W 30
            dc.L EditDefault
```

```
;
; Definition of a pop up menu control (and its menu)
;
PopUpMenu   dc.B '$$PopUpMenu:\N6',$00
            dc.B '--Selection 1\N259',$00
            dc.B '--Selection 2\N260',$00
            dc.B '--Selection 3\N261',$00
            dc.B '--Selection 4\N262',$00
            dc.b '.'
;
PopUp       dc.W 9
            dc.L 8
            dc.W 25,140,40,380
            dc.L popUpControl
            dc.W 0
            dc.W fCtlProcRefNotPtr+fMenuDefIsText
            dc.L 0
            dc.W 100
            dc.L PopUpMenu
            dc.W 259                ; initial value
;
; Definition of an icon button control
;
IconButtonTitle
            str 'Icon Button'

Icon        dc.w 0                  ;black and white icon
            dc.w 200
            dc.w 10                 ;icon height in pixels
            dc.w 40                 ;icon width in pixels
```

```
;
; Data for icon goes here (omitted)
;

IconButton
            dc.w 10                 ; pCount
            ds.l 1                  ; ID
            dc.w 40,40,80,100       ; button rectangle
            dc.l iconButtonControl  ; defproc
            dc.w 0                  ; single outline,
                                    ;  round-cornered
            dc.w FctlProcRefNotPtr  ; get defproc from
                                    ;  resource
            dc.l 0
            dc.l Icon               ; pointer to icon
            dc.l IconButtonTitle    ; pointer to p-string
                                    ;  title
            dc.l MyColorTable       ; pointer to color table
            dc.w 0                  ; standard drawing of icon
```

To create the above new controls in a window use the NewControl2 call:

```
            pha                     ; room for result
            pha
            PushLong WindPointer    ; Pointer to Owner window
            PushWord #ptrToPtr      ; Input verb for ptr to table
            PushLong #MyControls    ; pointer to table of templates
            _NewControl2
            pla                     ; discard these bytes, only verb
            pla                     ; for single ctl returns a value
```

New control records

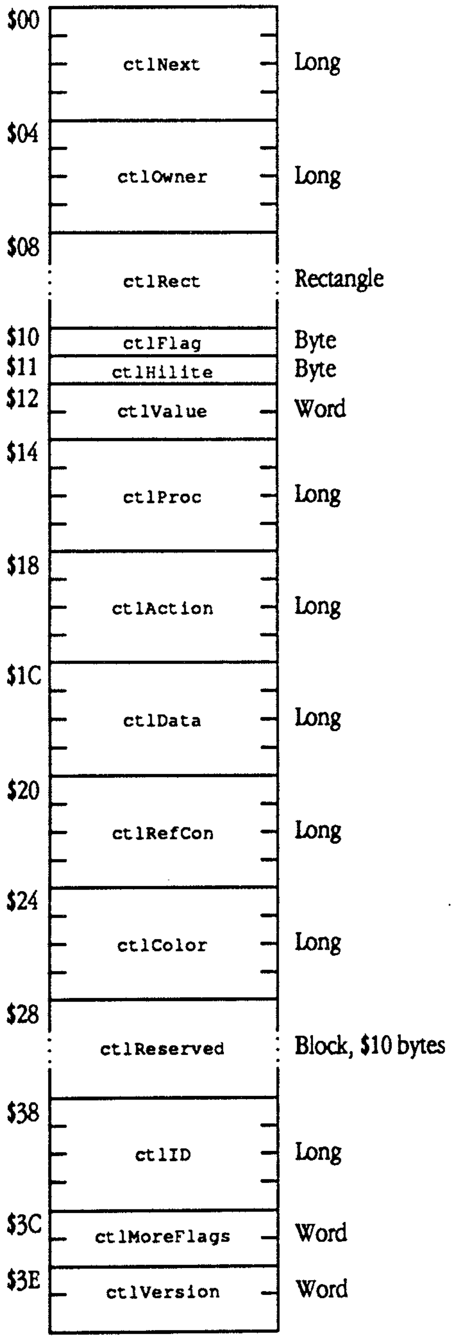
The NewControl2 tool call creates extended control records (as discussed earlier in this chapter in "New and changed controls"). This section describes the format and content of the control records created by NewControl2.

▲ **Warning** All control record layouts and field descriptions are provided so that programs may read these records for needed information. Your program should *never* set values into control records.▲

Generic extended control record

Currently, the Control Manager's standard, or generic, control record is $28 bytes long (see Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* for information about existing control records). To support the new controls (those created with NewControl2), the generic control record has several new fields. The layout of the new generic control record follows.

- **Figure 28-18** Generic extended control record



$00	ctlNext	Long
$04	ctlOwner	Long
$08	ctlRect	Rectangle
$10	ctlFlag	Byte
$11	ctlHilite	Byte
$12	ctlValue	Word
$14	ctlProc	Long
$18	ctlAction	Long
$1C	ctlData	Long
$20	ctlRefCon	Long
$24	ctlColor	Long
$28	ctlReserved	Block, $10 bytes
$38	ctlID	Long
$3C	ctlMoreFlags	Word
$3E	ctlVersion	Word

`ctlNext` A handle to the next control associated with this control's window. All the controls belonging to a given window are kept in a linked list, beginning in the `wControl` field of the window record and chained together through the `ctlNext` fields of the individual control records. The end of the list is marked by a zero value; as new controls are created, they're added to the beginning of the list.

`ctlOwner` A pointer to the window port to which the control belongs.

`ctlRect` The rectangle that defines the control's position and size in the local coordinates of the control's window.

`ctlFlag` A bit flag that further describes the control. The appropriate values are shown for each control in the sections that follow.

`ctlHilite` Specifies whether and how the control is to be highlighted and indicates whether the control is active or inactive. This field also specifies whether the control wants to receive selection events. The values for `ctlHilite` are as follows:

0 Control active; no highlighted parts—this value will cause events to be generated when the mouse button is pressed in the control

1–254 Part code of a highlighted part of the control

255 Control inactive—this value indicates that no events are to be generated when the mouse button is pressed in the control

Only one part of a control can be highlighted at any one time, and no part can be highlighted on an inactive control. See Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* for more information on highlighting.

`ctlValue` The control's current setting. For check boxes and radio buttons, zero means the control is off, and a nonzero value means it's on. For scroll bars, the value is between 0 and the data size minus the view size. The field is also available for custom controls to use as appropriate.

`ctlProc` For standard controls, this field indicates the control type, identified by its ID. For custom controls, this field contains a pointer to the control definition procedure (defProc) for this type of control.

For controls created with `NewControl`, valid ID values are:

`simpleProc`	$00000000	Simple button
`checkProc`	$02000000	Check box
`radioProc`	$04000000	Radio button
`scrollProc`	$06000000	Scroll bar
`growProc`	$08000000	Size box

For controls created with `NewControl2`, the `fCtlProcRefNotPtr` flag in `ctlMoreFlags` allows the Control Manager to discriminate between pointers and IDs. Valid ID values (used with `fCtlProcRefNotPtr` set to 1) are:

`simpleButtonControl`	$80000000	Simple button
`checkControl`	$82000000	Check box
`iconButtonControl`	$07FF0001	Icon button
`editLineControl`	$83000000	LineEdit
`listControl`	$89000000	List
`pictureControl`	$8D000000	Picture
`popUpControl`	$87000000	Pop-up
`radioControl`	$84000000	Radio control
`scrollBarControl`	$86000000	Scroll bar
`growControl`	$88000000	Size box
`statTextControl`	$81000000	Static text
`editTextControl`	$85000000	TextEdit

`ctlAction` Pointer to the control's custom action procedure, if any. `TrackControl` may call the custom action procedure to respond to the user dragging the mouse inside the control. See Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* for more information about `TrackControl`.

`ctlData` Reserved for use by the control definition procedure, typically to hold additional information for a particular control type. For example, the standard definition procedure for scroll bars uses the low-order word as the view size and the high-order word as the data size. The standard definition procedures for simple buttons, check boxes, and radio buttons store the address of the control's title.

`ctlRefCon` This field is reserved for application use.

`ctlColor` This field contains a reference to the color table to use when drawing the control. If the field is set to NIL, the Control Manager uses a default color table defined by the control's definition procedure. Otherwise, `ctlColor` contains a pointer, handle, or resource ID for the color table to use. Bits 2 and 3 of `ctlMoreFlags` usually allow the Control Manager to discriminate between these different data types.

`ctlReserved` This space is reserved for use by the control definition procedure. In some cases, the use is prescribed by the system. For example, keyboard equivalent information is stored here for controls that support keyboard equivalents.

`ctlID` This field may be used by the application to provide a straightforward mechanism for keeping track of controls. The control ID is a value assigned by your application with the `ID` field of the control template used to create the control. Your application can use the ID, which has a known value, to identify a particular control.

`ctlMoreFlags` This field contains bit flags that provide additional control information needed for new-style controls (those created with `NewControl2`). You can use the `GetCtlMoreFlags` Control Manager call to read the value of this field from a specified control record. Use the `SetCtlMoreFlags` call to change the value.

The high-order byte is used by the Control Manager to store its own control information. The low-order byte is used by the control definition procedure to define reference types.

The defined Control Manager flags are:

`fCtlTarget`	$8000	If set to 1, this control is currently the target of any typing or editing commands.
`fCtlCanBeTarget`	$4000	If set to 1 then this control can be made the target control.
`fCtlWantEvents`	$2000	If set to 1 then this control can be called when events are passed via the `SendEventToCtl` Control Manager call. Note that, if the `fCtlCanBeTarget` flag is set to 1, this control will receive events sent to it regardless of setting of this flag.

fCtlProcRefNotPtr $1000 If set to 1, then Control Manager expects ctlProc to contain the ID of a standard control procedure. If set to 0, then ctlProc contains a pointer to the custom control procedure.

fCtlTellAboutSize $0800 If set to 1, then this control needs to be notified when the size of the owning window has changed. This flag allows custom control procedures to resize their associated control images in response to changes in window size.

fCtlIsMultiPart $0400 If set to 1, then this is a multipart control. This flag allows control definition procedures to manage multi-part controls (necessary since the Control Manager does not know about all the parts of a multi-part control).

The low-order byte uses the following convention to describe references to color tables and titles (note, though, that some control templates do not follow this convention):

titleIsPtr	$00	Title reference is by pointer
titleIsHandle	$01	Title reference is by handle
titleIsResource	$02	Title reference is by resource ID
colorTableIsPtr	$00	Color table reference is by pointer
colorTableIsHandle	$04	Color table reference is by handle
colorTableIsResource	$08	Color table reference is by resource ID

ctlVersion This field is reserved for future use by the Control Manager to distinguish between different versions of control records.

Extended simple button control record

Figure 28-19 shows the format of the extended control record for simple button controls.

■ **Figure 28-19** Extended simple button control record

$00	`ctlNext`	Long—Handle to next control; NIL for last control
$04	`ctlOwner`	Long—Pointer to window to which control belongs
$08	`ctlRect`	Rectangle—Defines button's boundary rectangle
$10	`ctlFlag`	Byte—Defines button style
$11	`ctlHilite`	Byte—Current type of highlighting
$12	`ctlValue`	Word—Not used; set to 0
$14	`ctlProc`	Long—`simpleButtonControl`=$80000000
$18	`ctlAction`	Long—Pointer to custom procedure; NIL if none
$1C	`ctlData`	Long—Reference to button title string
$20	`ctlRefCon`	Long—Reserved for application use
$24	`ctlColor`	Long—Optional color table reference; NIL if none
$28	`keyEquiv`	Block, $06 Bytes—Key equivalent record
$2E	`ctlReserved`	Block, $0A bytes—Reserved
$38	`ctlID`	Long—Application-assigned ID
$3C	`ctlMoreFlags`	Word—Additional control flags
$3E	`ctlVersion`	Word—Set to 0

Valid values for `ctlFlag` are

`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 2–6	Must be set to 0
Button type	bits 0–1	Describes button type: 0 = single-outlined round-cornered button 1 = bold-outlined round-cornered button 2 = single-outlined square-cornered button 3 = single-outlined square-cornered drop-shadowed button

Valid values for `ctlMoreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Set to 1 if button has keystroke equivalent
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 4–10	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in `ctlColor` (if it is not NIL). See Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* for the definition of the simple button color table. 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Title reference	bits 0–1	Defines type of title reference in *ctlData*: 00 - title reference is pointer 01 - title reference is handle 10 - title reference is resource ID 11 - invalid value

`keyEquiv` Keystroke equivalent information stored at `keyEquiv` is formatted as shown in Figure 28-2.

Extended check box control record

Figure 28-20 shows the format of the extended control record for check box controls.

- **Figure 28-20** Extended check box control record

$00	`ctlNext`	Long—Handle to next control; NIL for last control
$04	`ctlOwner`	Long—Pointer to window to which control belongs
$08	`ctlRect`	Rectangle—Defines check box's boundary rectangle
$10	`ctlFlag`	Byte—Defines check box visibility
$11	`ctlHilite`	Byte—Current type of highlighting
$12	`ctlValue`	Word—0 if not checked; 1 if checked
$14	`ctlProc`	Long—`checkControl`=$82000000
$18	`ctlAction`	Long—Pointer to custom procedure; NIL if none
$1C	`ctlData`	Long—Reference to check box title string
$20	`ctlRefCon`	Long—Reserved for application use
$24	`ctlColor`	Long—Optional color table reference; NIL if none
$28	`keyEquiv`	Block, $06 Bytes—Key equivalent record
$2E	`ctlReserved`	Block, $0A bytes—Reserved
$38	`ctlID`	Long—Application-assigned ID
$3C	`ctlMoreFlags`	Word—Additional control flags
$3E	`ctlVersion`	Word—Set to 0

Valid values for `ctlFlag` are

`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 0–6	Must be set to 0

Valid values for `ctlMoreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Set to 1 if check box has keystroke equivalent
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 4–10	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in `ctlColor` (if it is not NIL). See Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* for the definition of the check box color table. 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Title reference	bits 0-1	Defines type of title reference in `ctlData`: 00 - title reference is pointer 01 - title reference is handle 10 - title reference is resource ID 11 - invalid value

`keyEquiv` Keystroke equivalent information stored at `keyEquiv` is formatted as shown in Figure 28-2.

Icon button control record

Figure 28-21 shows the format of the control record for icon button controls.

■ **Figure 28-21** Icon button control record

$00	`ctlNext`	Long—Handle to next control; NIL for last control
$04	`ctlOwner`	Long—Pointer to window to which control belongs
$08	`ctlRect`	Rectangle—Defines icon boundary rectangle
$10	`ctlFlag`	Byte—Defines control visibility and button style
$11	`ctlHilite`	Byte—Controls highlighting
$12	`ctlValue`	Word—Not used; set to 0
$14	`ctlProc`	Long—`iconButtonControl`=$07FF0001
$18	`ctlAction`	Long—Pointer to custom procedure; NIL if none
$1C	`ctlData`	Long—Optional reference to title string for button
$20	`ctlRefCon`	Long—Reserved for application use
$24	`ctlColor`	Long—Optional color table reference; NIL if none
$28	`keyEquiv`	Block, $06 bytes—Key equivalent record
	continued	

	continued	
$2E	`ctlReserved`	Block, $0A bytes—Reserved
$38	`ctlID`	Long—Application-assigned ID
$3C	`ctlMoreFlags`	Word—Additional control flags
$3E	`ctlVersion`	Word—Set to 0
$40	`iconRef`	Long—Reference to icon
$44	`displayMode`	Word—Bit flag defining icon's appearance

Valid values for `ctlFlag` are

`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 3–6	Must be set to 0
`showBorder`	bit 2	1=No border, 0=Show border
`buttonType`	bits 0–1	Defines button type: 00 - single-outlined round-cornered button 01 - bold-outlined round-cornered button 10 - single-outlined square-cornered button 11 - single-outlined square-cornered and drop-shadowed button

Valid values for `ctlMoreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 0
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 6–10	Must be set to 0
Icon reference	bits 4–5	Defines type of icon reference in `iconRef`: 00 - icon reference is pointer 01 - icon reference is handle 10 - icon reference is resource ID 11 - invalid value
Color table reference	bits 2–3	Defines type of reference in `ctlColor` (if it is not NIL). The color table for an icon button is the same as that for a simple button. See Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* for the definition of the simple button color table. 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Title reference	bits 0–1	Defines type of title reference in `ctlData`: 00 - title reference is pointer 01 - title reference is handle 10 - title reference is resource ID 11 - invalid value

`ctlData` Holds the reference to the title string, which must be a Pascal string.

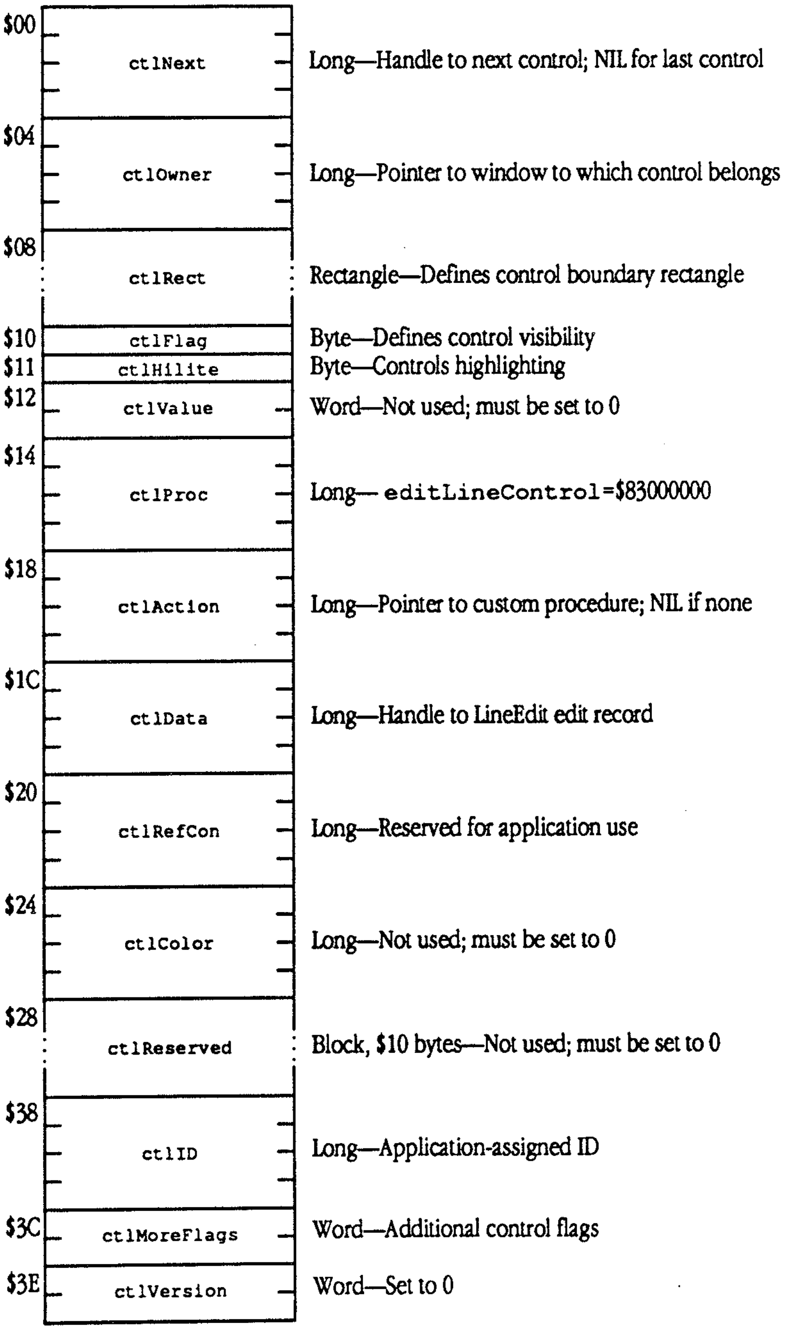
`displayMode` Passed directly to the `DrawIcon` routine, and defines the display mode for the icon. The Control Manager sets this field from the `displayMode` field in the icon button control template used to create the control.

`keyEquiv` Keystroke equivalent information stored at `keyEquiv` is formatted as shown in Figure 28-2.

LineEdit control record

Figure 28-22 shows the format of the control record for LineEdit controls.

- **Figure 28-22** LineEdit control record



$00	`ctlNext`	Long—Handle to next control; NIL for last control
$04	`ctlOwner`	Long—Pointer to window to which control belongs
$08	`ctlRect`	Rectangle—Defines control boundary rectangle
$10	`ctlFlag`	Byte—Defines control visibility
$11	`ctlHilite`	Byte—Controls highlighting
$12	`ctlValue`	Word—Not used; must be set to 0
$14	`ctlProc`	Long— `editLineControl`=$83000000
$18	`ctlAction`	Long—Pointer to custom procedure; NIL if none
$1C	`ctlData`	Long—Handle to LineEdit edit record
$20	`ctlRefCon`	Long—Reserved for application use
$24	`ctlColor`	Long—Not used; must be set to 0
$28	`ctlReserved`	Block, $10 bytes—Not used; must be set to 0
$38	`ctlID`	Long—Application-assigned ID
$3C	`ctlMoreFlags`	Word—Additional control flags
$3E	`ctlVersion`	Word—Set to 0

Valid values for `ctlFlag` are

`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 0–6	Must be set to 0

Valid values for `ctlMoreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 1
`fCtlWantsEvents`	bit 13	Must be set to 1
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 2–10	Must be set to 0
Text reference	bits 0–1	Defines type of text reference in `ctlData` 00 - text reference is pointer 01 - text reference is handle 10 - text reference is resource ID 11 - invalid value

`ctlData` Control Manager stores the handle to the LineEdit edit record in the `ctlData` field. If you want to issue LineEdit tool calls directly, you can retrieve the handle from that field.

Note that LineEdit controls do not support color tables.

List control record

Figure 28-23 shows the format of the control record for list controls.

- **Figure 28-23** List control record

$00	ctlNext	Long—Handle to next control; NIL for last control
$04	ctlOwner	Long—Pointer to window to which control belongs
$08	ctlRect	Rectangle—Defines control boundary rectangle
$10	ctlFlag	Byte—Defines style of scroll bar for list window
$11	ctlHilite	Byte—Not used; must be set to 0
$12	ctlValue	Word—Reserved
$14	ctlProc	Long—`listControl` =$89000000
$18	ctlAction	Long—Pointer to custom procedure; NIL if none
$1C	ctlData	Long—High-word is `listSize`; low-word is `viewSize`
$20	ctlRefCon	Long—Reserved for application use
$24	ctlColor	Long—Reference to the color table for the control
$28	ctlMemDraw	Long—Pointer to list member drawing routine
$2C	ctlMemHeight	Word—List member height in pixels
$2E	ctlMemSize	Word—List member record size in bytes
	continued	

	continued	
$30	ctlListRef	Long—Reference to list member records
$34	ctlListBar	Long—Handle of control's scroll bar control
$38	ctlID	Long—Application-assigned ID
$3C	ctlMoreFlags	Word—Additional control flags
$3E	ctlVersion	Word—Set to 0

Valid values for `ctlFlag` are

`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 0–6	Must be set to 0

Valid values for `ctlMoreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 0
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
`fCtlIsMultiPart`	bit 10	Must be set to 1
Reserved	bits 4–9	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in `ctlColor` (if it is not NIL). The color table for a List control is described in Chapter 11, "List Manager," in Volume 1 of the *Toolbox Reference*. 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
List reference	bits 0–1	Defines type of reference in `listRef`. The format for a list member record is described in Chapter 11, "List Manager," in Volume 1 of the *Toolbox Reference*. 00 - list reference is pointer 01 - list reference is handle 10 - list reference is resource ID 11 - invalid value

Picture control record

Figure 28-24 shows the format of the control record for picture controls.

- **Figure 28-24** Picture control record

$00	`ctlNext`	Long—Handle to next control; NIL for last control
$04	`ctlOwner`	Long—Pointer to window to which control belongs
$08	`ctlRect`	Rectangle—Defines picture boundary rectangle
$10	`ctlFlag`	Byte—Defines picture visibility
$11	`ctlHilite`	Byte—Controls event generation for control
$12	`ctlValue`	Word—Not used; set to 0
$14	`ctlProc`	Long—`pictureControl` =$8D000000
$18	`ctlAction`	Long—Pointer to custom procedure; NIL if none
$1C	`ctlData`	Long—Reference to picture
$20	`ctlRefCon`	Long—Reserved for application use
$24	`ctlColor`	Long—Not used; must be set to 0
$28	`ctlReserved`	Block, $10 bytes—Not used
$38	`ctlID`	Long—Application-assigned ID
$3C	`ctlMoreFlags`	Word—Additional control flags
$3E	`ctlVersion`	Word—Set to 0

Valid values for `ctlFlag` are

ctlInvis	bit 7	1=invisible, 0=visible
Reserved	bits 0–6	Must be set to 0

Valid values for `ctlMoreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 0
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 2–10	Must be set to 0
Picture reference	bits 0–1	Define type of picture reference in `ctlData`: 00 - invalid value 01 - reference is handle 10 - reference is resource ID 11 - invalid value

`ctlHilite` Specifies whether the control wants to receive mouse events. The values for `ctlHilite` are as follows:

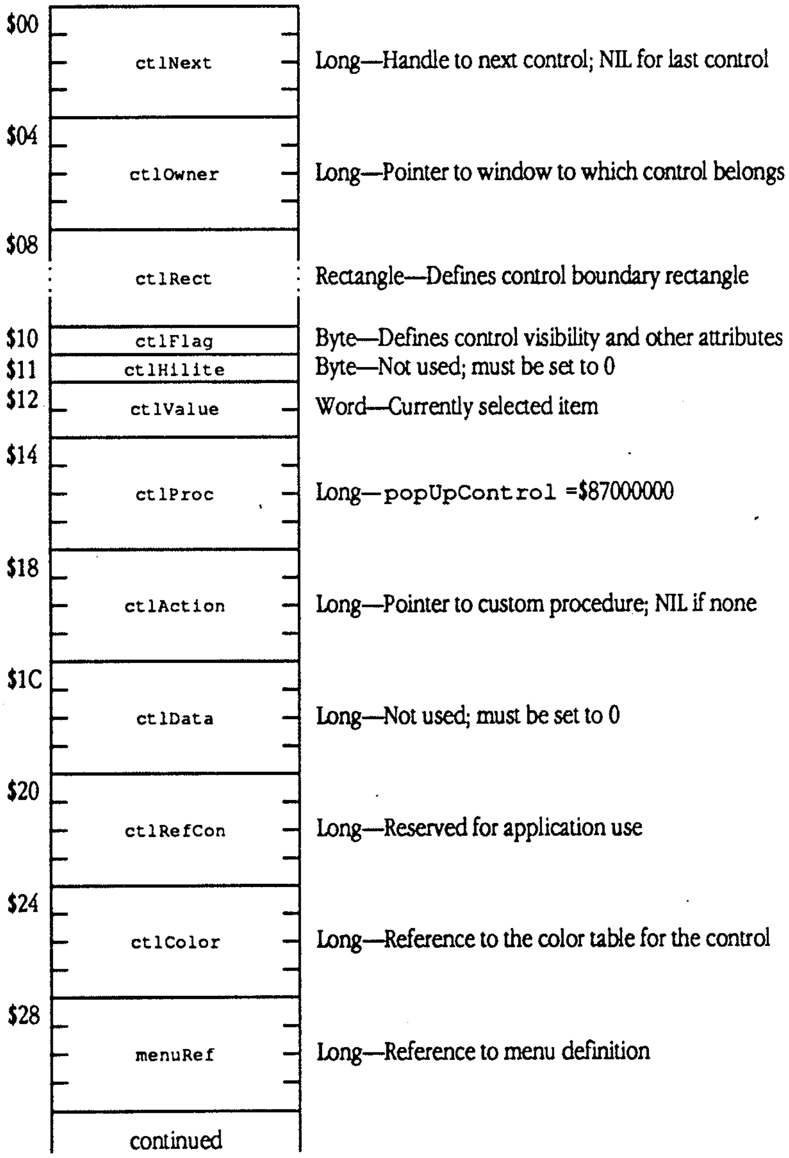
0 Events will be generated when the mouse button is pressed in the control

255 No events will be generated when the mouse buttron is pressed in the control

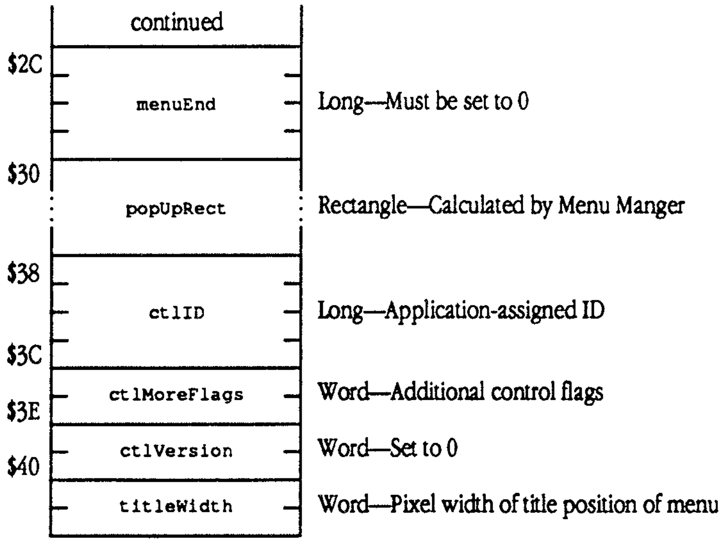
Pop-up control record

Figure 28-25 shows the format of the control record for pop-up menu controls.

- **Figure 28-25** Pop-up control record



$00	ctlNext	Long—Handle to next control; NIL for last control
$04	ctlOwner	Long—Pointer to window to which control belongs
$08	ctlRect	Rectangle—Defines control boundary rectangle
$10	ctlFlag	Byte—Defines control visibility and other attributes
$11	ctlHilite	Byte—Not used; must be set to 0
$12	ctlValue	Word—Currently selected item
$14	ctlProc	Long—`popUpControl` =$87000000
$18	ctlAction	Long—Pointer to custom procedure; NIL if none
$1C	ctlData	Long—Not used; must be set to 0
$20	ctlRefCon	Long—Reserved for application use
$24	ctlColor	Long—Reference to the color table for the control
$28	menuRef	Long—Reference to menu definition
	continued	


continued
$2C
menuEnd
Long—Must be set to 0
$30
popUpRect
Rectangle—Calculated by Menu Manger
$38
ctlID
Long—Application-assigned ID
$3C
ctlMoreFlags
Word—Additional control flags
$3E
ctlVersion
Word—Set to 0
$40
titleWidth
Word—Pixel width of title position of menu

Valid values for `ctlFlag` are

ctlInvis	bit 7	1=invisible, 0=visible
`fType2PopUp`	bit 6	Indicates type of pop-up menú: 1 - Draw pop-up with white space (Type 2) 0 - Draw normal pop-up
`fDontHiliteTitle`	bit 5	Controls highlighting of the control title: 1 - Do not highlight title when control is popped up 0 - Highlight title
`fDontDrawTitle`	bit 4	Indicates whether the Control Manager is to draw the menu title: 1 - Do not draw the title 0 - Draw the title
`fDontDrawResult`	bit 3	Indicates whether result is shown: 1 - Do not draw the result in the result area after a selection 0 - Draw the result

`fInWindowOnly` bit 2 Controls the extent to which the pop-up menu can grow; this is particularly relevant with respect to Type 2 pop-ups (see Chapter 37, "Menu Manager Update," for details on Type 2 pop-up menus):
1 - Keep the pop-up in the current window
0 - Allow the pop-up to grow to screen size

`fRightJustifyTitle` bit 1 Controls title justification:
1 - Right justify the title; note that if the title is right justified, then the control rectangle is adjusted to eliminate unneeded pixels (see Figure 28-12), the value for `titleWidth` is also adjusted
0 - Left justify the title

`fRightJustifyResult` bit 0 Controls result justification:
1 - Right justify the selection
0 - Left justify the selection `titleWidth` pixels from the left of the pop-up rectangle.

Valid values for `ctlMoreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 1 if the pop-up has any keystroke equivalents defined
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 5–10	Must be set to 0
Color table reference	bits 3–4	Defines type of reference in `colorTableRef` (the color table for a menu is described in Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference)* 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
`fMenuDefIsText`	bit 2	Defines type of data referred to by `menuRef`: 1 - `menuRef` is a pointer to a text stream in NewMenu format (see Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference* for details) 0 - `menuRef` is a reference to a Menu Template (again, see Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference* for details on format and content of a Menu Template)
Menu reference	bits 0-1	Defines type of menu reference in `menuRef` (if `fMenuDefIsText` is set to 1, then these bits are ignored): 00 - menu reference is pointer 01 - menu reference is handle 10 - menu reference is resource ID 11 - invalid value

`ctlRect` Defines the boundary rectangle for the pop-up and its title, before the menu has been "popped" by the user. The Menu Manager will calculate the lower-right coordinates of the rectangle for you, if you specify those coordinates as (0,0).

`ctlValue` Contains the item number of the currently selected item.

`titleWidth` Contains the value set in the `titleWidth` field of the pop-up menu control template used to create the control.

`menuRef` Reference to menu definition (see Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference* and Chapter 37, "Menu Manager Update," in this book for details on menu templates). The type of reference contained in `menuRef` is defined by the Menu reference bits in `ctlMoreFlags`. This field is set from the `menuRef` field of the pop-up menu control template used to create the control.

Extended radio button control record

Figure 28-26 shows the format of the extended control record for radio button controls.

- **Figure 28-26** Extended radio button control record

$00	`ctlNext`	Long—Handle to next control; NIL for last control
$04	`ctlOwner`	Long—Pointer to window to which control belongs
$08	`ctlRect`	Rectangle—Defines radio button's boundary rectangle
$10	`ctlFlag`	Byte—Defines button visibility and family affinity
$11	`ctlHilite`	Byte—Current type of highlighting
$12	`ctlValue`	Word—0 if off; 1 if on
$14	`ctlProc`	Long—`radioControl`=$84000000
$18	`ctlAction`	Long—Pointer to custom procedure; NIL if none
$1C	`ctlData`	Long—Reference to radio button title string
$20	`ctlRefCon`	Long—Reserved for application use
$24	`ctlColor`	Long—Optional color table reference; NIL if none
$28	`keyEquiv`	Block, $06 Bytes—Key equivalent record
$2E	`ctlReserved`	Block, $0A bytes—Reserved
$38	`ctlID`	Long—Application-assigned ID
$3C	`ctlMoreFlags`	Word—Additional control flags
$3E	`ctlVersion`	Word—Set to 0

Valid values for `ctlFlag` are

`ctlInvis`	bit 7	1=invisible, 0=visible
Family number	bits 0–6	Family numbers define associated groups of radio buttons. Radio buttons in the same family are logically linked. That is, setting one radio button in a family clears all other buttons in the same family

Valid values for `ctlMoreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Set to 1 if button has keystroke equivalent
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 4–10	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in `ctlColor` (if it is not NIL). See Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* for the definition of the radio button color table. 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Title reference	bits 0–1	Defines type of title reference in `ctlData`: 00 - title reference is pointer 01 - title reference is handle 10 - title reference is resource ID 11 - invalid value

`keyEquiv` Keystroke equivalent information stored at `keyEquiv` is formatted as shown in Figure 28-2.

Extended scroll bar control record

Figure 28-27 shows the format of the extended control record for scroll bar controls.

- **Figure 28-27** Extended scroll bar control record

$00	`ctlNext`	Long—Handle to next control; NIL for last control
$04	`ctlOwner`	Long—Pointer to window to which control belongs
$08	`ctlRect`	Rectangle—Defines scroll bar's boundary rectangle
$10	`ctlFlag`	Byte—Style of scroll bar
$11	`ctlHilite`	Byte—Current type of highlighting
$12	`ctlValue`	Word—Thumb position between 0 and (`dataSize` — `viewSize`)
$14	`ctlProc`	Long—`scrollControl`=$86000000
$18	`ctlAction`	Long—Pointer to custom procedure; NIL if none
$1C	`ctlData`	Long—High-order word= `dataSize`, low-order word= `viewSize`
$20	`ctlRefCon`	Reserved for application use
$24	`ctlColor`	Long—Optional color table reference; NIL if none
$28	`thumbRect`	Rectangle—Defines thumb rectangle
$30	`pageRegion`	Rectangle—Defines page region, thumb bounds
$38	`ctlID`	Long—Application-assigned ID
$3C	`ctlMoreFlags`	Word—Additional control flags
$3E	`ctlVersion`	Word—Set to 0

Valid values for `ctlFlag` are

`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 5–6	Must be set to 0
`horScroll`	bit 4	1=horizontal scroll bar, 0=vertical scroll bar
`rightFlag`	bit 3	1=bar has right arrow, 0=bar has no right arrow
`leftFlag`	bit 2	1=bar has left arrow, 0=bar has no left arrow
`downFlag`	bit 1	1=bar has down arrow, 0=bar has no down arrow
`upFlag`	bit 0	1=bar has up arrow, 0=bar has no up arrow

Note that extraneous flag bits are ignored, based upon the state of the `horScroll` flag. For example, for vertical scroll bars, `rightFlag` and `leftFlag` are ignored.

Valid values for `ctlMoreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 0
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 4–10	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in `ctlColor` (if it is not NIL). See Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* and "Clarifications" in this chapter for the definition of the scroll bar color table. 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Reserved	bits 0–1	Must be set to 0

Extended size box control record

Figure 28-28 shows the format of the extended control record for size box controls.

- **Figure 28-28** Extended size box control record

$00	`ctlNext`	Long—Handle to next control; NIL for last control
$04	`ctlOwner`	Long—Pointer to window to which control belongs
$08	`ctlRect`	Rectangle—Defines size box's boundary rectangle
$10	`ctlFlag`	Byte—Define size box visibility
$11	`ctlHilite`	Byte—Current type of highlighting
$12	`ctlValue`	Word—Not used; set to 0
$14	`ctlProc`	Long—`growControl`=$88000000
$18	`ctlAction`	Long—Pointer to custom procedure; NIL if none
$1C	`ctlData`	Long—Not used; set to 0
$20	`ctlRefCon`	Long—Reserved for application use
$24	`ctlColor`	Long—Optional color table reference; NIL if none
$28	`ctlReserved`	Block, $10 bytes—Not used; set to 0
$38	`ctlID`	Long—Application-assigned ID
$3C	`ctlMoreFlags`	Word—Additional control flags
$3E	`ctlVersion`	Word—Set to 0

Valid values for `ctlFlag` are as follows:

`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 1–6	Must be set to 0
`fCallWindowMgr`	bit 0	1=call `GrowWindow` and `SizeWindow` to track this control; 0=just highlight control

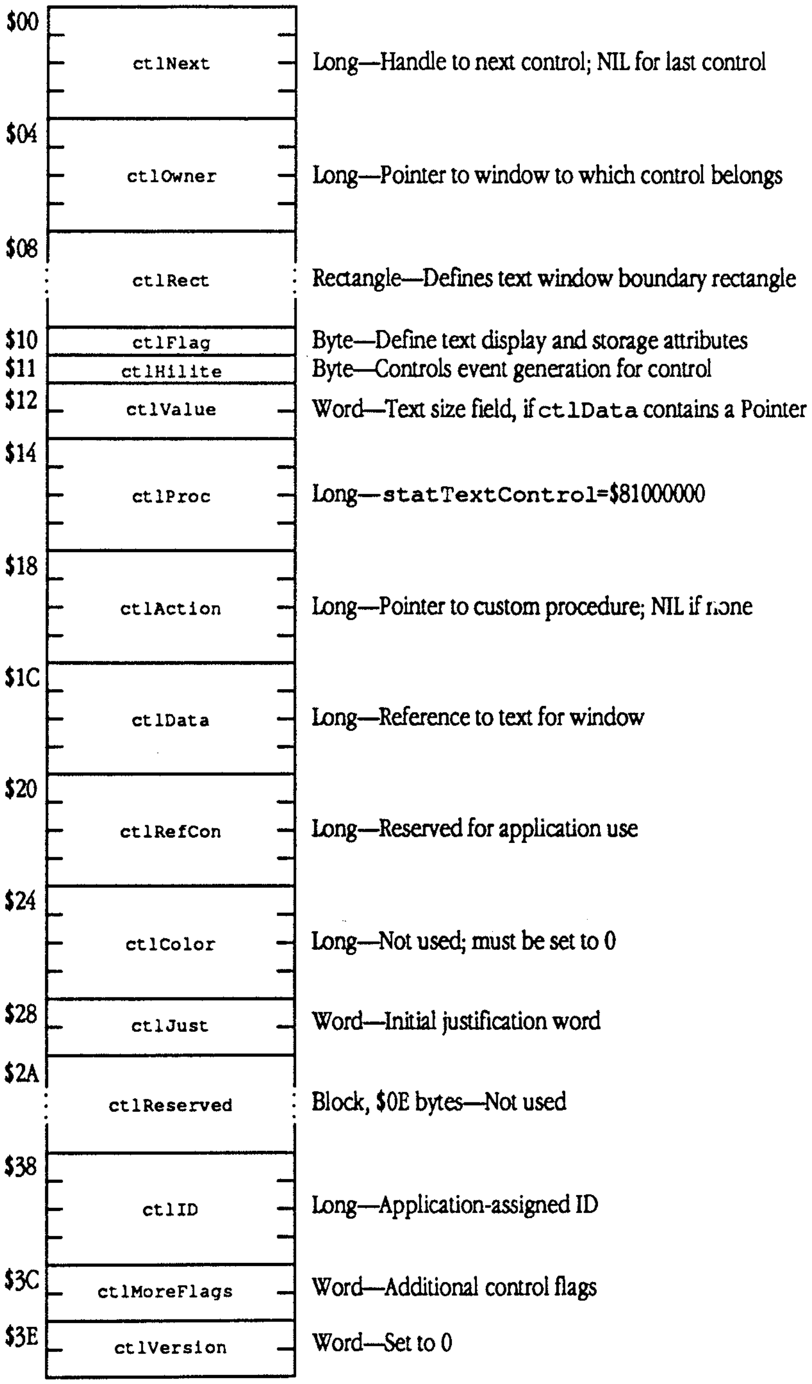
Valid values for `ctlMoreFlags` are as follows:

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 0
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 4–10	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in `ctlColor` (if it is not NIL). See "Error Corrections" in this chapter for the definition of the size box color table. 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Reserved	bits 0–1	Must be set to 0

Static text control record

Figure 28-29 shows the format of the control record for static text controls.

- **Figure 28-29** Static text control record



Offset	Field	Description
$00	`ctlNext`	Long—Handle to next control; NIL for last control
$04	`ctlOwner`	Long—Pointer to window to which control belongs
$08	`ctlRect`	Rectangle—Defines text window boundary rectangle
$10	`ctlFlag`	Byte—Define text display and storage attributes
$11	`ctlHilite`	Byte—Controls event generation for control
$12	`ctlValue`	Word—Text size field, if `ctlData` contains a Pointer
$14	`ctlProc`	Long—`statTextControl`=$81000000
$18	`ctlAction`	Long—Pointer to custom procedure; NIL if none
$1C	`ctlData`	Long—Reference to text for window
$20	`ctlRefCon`	Long—Reserved for application use
$24	`ctlColor`	Long—Not used; must be set to 0
$28	`ctlJust`	Word—Initial justification word
$2A	`ctlReserved`	Block, $0E bytes—Not used
$38	`ctlID`	Long—Application-assigned ID
$3C	`ctlMoreFlags`	Word—Additional control flags
$3E	`ctlVersion`	Word—Set to 0

Valid values for `ctlFlag` are

`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 2–6	Must be set to 0
`fSubstituteText`	bit 1	0=no text substitution to perform 1=there is text substitution to perform
`fSubTextType`	bit 0	0=C strings 1=Pascal strings

Valid values for `ctlMoreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 0
`fCtlWantsEvents`	bit 13	Must be set to 0
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fCtlTellAboutSize`	bit 11	Must be set to 0
Reserved	bits 2–10	Must be set to 0
Text reference	bits 0–1	Defines type of text reference in `ctlData`: 00 - text reference is pointer 01 - text reference is handle 10 - text reference is resource ID 11 - invalid value

`ctlHilite` Specifies whether the control wants to receive mouse selection events. The values for `ctlHilite` are as follows:

0	Events will be generated when the mouse button is pressed in the control
255	No events will be generated when the mouse button is pressed in the control

`ctlValue` Contains the size of the referenced text in characters, but only if the text reference in `ctlData` is a pointer. If the text reference is either a handle or a resource ID, then the Control Manager can extract the length from the handle.

`ctlJust` The justification word is passed on to `LETextBox2` (see Chapter 10, "LineEdit Tool Set," in Volume 1 of the *Toolbox Reference* for details on the `LETextBox2` tool call), and is used to set the initial justification for the text being drawn. Valid values for `ctlJust` are

`leftJustify`	0	Text is left justified in the display window
`centerJustify`	1	Text is centered in the display window
`rightJustify`	-1	Text is right justified in the display window
`fullJustify`	2	Text is fully justified (both left and right) in the display window

Static text controls do not support color tables. To display text of different color, you must embed the appropriate commands into the text string you are displaying. See the discussion of `LETextBox2` in Chapter 10, "LineEdit Tool Set," in Volume 1 of the *Toolbox Reference* for details on command format and syntax.

TextEdit control record

Figure 28-30 shows the format of the control record for TextEdit controls.

- **Figure 28-30** TextEdit control record

$00	`ctlNext`	Long—Handle to next control; NIL for last control
$04	`ctlOwner`	Long—Pointer to window to which control belongs
$08	`ctlRect`	Rectangle—Defines control boundary rectangle
$10	`ctlFlag`	Byte—Defines control visibility
$11	`ctlHilite`	Byte—Not used; must be set to 0
$12	`ctlValue`	Word—Contains the last reported TextEdit error code
$14	`ctlProc`	Long—`editTextControl`=$85000000
$18	`ctlAction`	Long—Pointer to custom procedure; NIL if none
$1C	`ctlData`	Long—Pointer to filter procedure
$20	`ctlRefCon`	Long—Reserved for application use
$24	`ctlColor`	Long—Reference to the color table for the control
$28	`textFlags`	Long—TextEdit bit flags
$2C	`textLength`	Long—Length of text

continued

	continued	
$30	`blockList`	TextList—Cached link into `TextBlock` list
$38	`ctlID`	Long—Application-assigned ID
$3C	`ctlMoreFlags`	Word—Additional control flags
$3E	`ctlVersion`	Word—Set to 0
$40	`viewRect`	Rectangle—Boundary rectangle for text
$48	`totalHeight`	Long—Height, in pixels, of text
$4C	`lineSuper`	SuperHandle—Cached link into text lines
$58	`styleSuper`	SuperHandle—Cached link into style list
$64	`styleList`	Long—Handle to array of `TEStyle` records
$68	`rulerList`	Long—Handle to array of `TERuler` records
$6C	`lineAtEndFlag`	Word—Line break flag
$6E	`selectionStart`	Long—Starting text offset for current selection
$72	`selectionEnd`	Long—Ending text offset for current selection
	continued	

	continued	
$76	`selectionActive`	Word—Flag indicating whether current selection is active
$78	`selectionState`	Word—State information about current selection
$7A	`caretTime`	Long—Blink interval for caret, in system ticks
$7E	`nullStyleActive`	Word—Flag indicating whether null style is active
$80	`nullStyle`	TEStyle—Null style definition
$8C	`topTextOffset`	Long—Offset to top line of displayed text
$90	`topTextVPos`	Word—Position of display window into text, in pixels
$92	`vertScrollBar`	Long—Handle to vertical scroll bar control record
$96	`vertScrollPos`	Long—Current position of vertical scroll bar
$9A	`vertScrollMax`	Long—Maximum allowable vertical scroll
$9E	`vertScrollAmount`	Word—Number of pixels to scroll on each click
$0A	`horzScrollBar`	Long—Currently not supported
$A4	`horzScrollPos`	Long—Currently not supported
$A8	`horzScrollMax`	Long—Currently not supported
	continued	

continued

$AC	`horzScrollAmount`	Word—Currently not supported
$AE	`growBoxHandle`	Long—Handle of size box control record
$B2	`maximumChars`	Long—Maximum number of characters allowed in text
$B6	`maximumLines`	Long—Currently not supported
$BA	`maxCharsPerLine`	Word—Currently not supported
$BC	`maximumHeight`	Word—Currently not supported
$BE	`textDrawMode`	Word—QuickDraw II drawing mode for text
$C0	`wordBreakHook`	Long—Pointer to word break hook routine
$C4	`wordWrapHook`	Long—Pointer to word wrap hook routine
$C8	`keyFilter`	Long—Pointer to keystroke filter routine
$CC	`theFilterRect`	Rectangle—Rectangle for generic filter procedure
$D4	`theBufferVPos`	Word—Vertical component of current position
$D6	`theBufferHPos`	Word—Horizontal component of current position

continued

	continued	
$D8	theKeyRecord	KeyRecord—Parameters for keystroke filter routine
$E6	cachedSelcOffset	Long—Cached selection text offset
$EA	cachedSelcVPos	Word—Vertical component of cached buffer position
$EC	cachedSelcHPos	Word—Horizontal component of cached buffer position
$EE	mouseRect	Rectangle—Boundary rectangle for multiclick mouse commands
$F6	mouseTime	Long—Time of last mouse click
$FA	mouseKind	Word—Kind of mouse click last performed
$FC	lastClick	Long—Location of last mouse click
$100	savedHPos	Word—Cached horizontal character position
$102	anchorPoint	Long—Starting point of current selection

Valid values for `ctlFlag` are

`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 0–6	Must be set to 0

Valid values for `textFlags` are

`fNotControl`	bit 31	Must be set to 0
`fSingleFormat`	bit 30	Must be set to 1
`fSingleStyle`	bit 29	Indicates the style options available to the user 1 - Allow only one style in the text 0 - Do not restrict the number of styles in the text
`fNoWordWrap`	bit 28	Indicates TextEdit word wrap behavior 1 - Do not word wrap the text; only break lines on CR ($0D) characters 0 - Perform word wrap to fit the ruler
`fNoScroll`	bit 27	Controls user access to scrolling 1 - Do not allow either manual or auto-scrolling 0 - Scrolling permitted
`fReadOnly`	bit 26	Restricts the text in the window to read-only operations (copying from the window will still be allowed) 1 - No editing allowed 0 - Editing permitted
`fSmartCutPaste`	bit 25	Controls TextEdit support for smart cut and paste (see Chapter 49, "TextEdit," for details on smart cut and paste support) 1 - Use smart cut and paste 0 - Do not use smart cut and paste
`fTabSwitch`	bit 24	Defines behavior of the Tab key (see Chapter 49, "TextEdit," for details) 1 - Tab to next control in the window 0 - Tab inserted in TextEdit document
`fDrawBounds`	bit 23	Indicates whether TextEdit will draw a box around the edit window, just inside `ctlRect` (the pen for this rectangle is 2 pixels wide and 1 pixel high) 1 - Draw rectangle 0 - Do not draw rectangle
`fColorHilight`	bit 22	Must be set to 0.
`fGrowRuler`	bit 21	Indicates whether TextEdit will resize the ruler in response to the user resizing the edit window. If set to 1, TextEdit will automatically adjust the right margin value for the ruler 1 - Resize the ruler 0 - Do not resize the ruler

`fDisableSelection` bit 20 Controls whether user can select text
1 - User cannot select text
0 - User can select text

`fDrawInactiveSelection`
bit 19 Controls how inactive selected text is displayed
1 - TextEdit draws a box around inactive selections
0 - TextEdit does not display inactive selections

Reserved bits 0–18 Must be set to 0

`textLength` Number of bytes of text in the record. Your program must not modify this field.

`blockList` Cached link into the linked list of `TextBlock` structures, which contain the actual text for the record. The actual `TextList` structure resides here. Your program must not modify this field.

Valid values for `ctlMoreFlags` are

`fCtlTarget`	bit 15	Must be set to 0
`fCtlCanBeTarget`	bit 14	Must be set to 1
`fCtlWantsEvents`	bit 13	Must be set to 1
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fTellAboutSize`	bit 11	If set to 1, a size box will be created in the lower-right corner of the window. Whenever the control window is resized, the edit text will be resized and redrawn.
`fCtlIsMultiPart`	bit 10	Must be set to 1
Reserved	bits 4–9	Must be set to 0
Color table reference	bits 2–3	Defines type of reference in `ctlColor` (if it is not NIL). The color table for a TextEdit control (`TEColorTable`) is described in Chapter 49, "TextEdit," later in this book 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Style reference	bits 0–1	Defines type of style reference in `styleRef`. The format for a TextEdit style descriptor is described in Chapter 49, "TextEdit," later in this book 00 - style reference is pointer 01 - style reference is handle 10 - style reference is resource ID 11 - invalid value

△ **Important** Do not set `fTellAboutSize` to 1 unless the control also has a vertical scroll bar. △

`viewRect` Boundary rectangle for the text, within the rectangle defined in `boundsRect`, which surrounds the entire record, including its associated scroll bars and outline.

`totalHeight` Total height of the text in the TextEdit record, in pixels.

`lineSuper` Cached link into the linked list of `SuperBlock` structures that define the text lines in the record.

styleSuper Cached link into the linked list of SuperBlock structures that define the styles for the record.

styleList Handle to array of TEStyle structures, containing style information for the record. The array is terminated with a long set to $FFFFFFFF.

rulerList Handle to array of TERuler structures, defining the format rulers for the record. Note that only the first ruler is currently used by TextEdit. The array is terminated with a long set to $FFFFFFFF.

lineAtEndFlag Indicates whether the last character was a line break. If so, this field is set to $FFFF.

selectionStart
Starting text offset for the current selection. Must always be less than selectionEnd.

selectionEnd Ending text offset for the current selection.Must always be greater than selectionStart.

selectionActive
Indicates whether the current selection (defined by selectionStart and selectionEnd) is active:

$0000	Active
$FFFF	Inactive

selectionState Contains state information about the current selection range:

$0000	Off screen
$FFFF	On screen

caretTime Blink interrval for caret, expressed in system ticks.

nullStyleActive
Indicates whether the null style is active for the current selection:

$0000	Do not use null style when inserting text
$FFFF	Use null style when inserting text

nullStyle TEStyle structure defining the null style. This may be the default style for newly inserted text, depending upon the value of nullStyleActive.

topTextOffset Text offset into the record corresponding to the top line displayed on the screen.

`topTextVPos` Difference,in pixels, between the topmost vertical scroll position (corresponding to the top of the vertical scroll bar) and the top line currently displayed on the screen. This is, essentially, the vertical position of the display window in the text for the record.

`vertScrollBar` Handle to the vertical scroll bar control record.

`vertScrollPos` Current position of the vertical scroll bar, in units defined by `vertScrollAmount`.

◆ *Note:* that while TextEdit supports `vertScrollPos` as a long, standard Apple IIGS scroll bars support only the low-order word. This leads to unpredictable scroll bar behavior when editing large documents.

`vertScrollMax` Maximum allowable vertical scroll, in units defined by `vertScrollAmount`.

`vertScrollAmount`
Number of pixels to scroll on each vertical arrow click.

`horzScrollBar` Handle to the horizontal scroll bar control record. Currently not supported

`horzScrollPos` Current position of the horizontal scroll bar, in units defined by `horzScrollAmount`. Currently not supported

`horzScrollMax` Maximum allowable horizontal scroll, in units defined by `horzScrollAmount`. Currently not supported

`horzScrollAmount`
Number of pixels to scroll on each horizontal arrow click. Currently not supported.

`growBoxHandle` Handle of size box control record.

`maximumChars` Maximum number of characters allowed in the text.

`maximumLines` Maximum number of lines allowed in the text. Currently not supported.

`maxCharsPerLine`
Currently not supported.

`maximumHeight` Maximum text height, in pixels. This value allows applications to easily constrain text to a display window of a known height. Currently not supported.

`textDrawMode` QuickDraw II drawing mode for the text. See Chapter 16, "QuickDraw II," in the *Toolbox Reference* for more information on QuickDraw II drawing modes.

`wordBreakHook` Pointer to the routine that handles word breaks. See Chapter 49, "TextEdit," for information about word break routines. Your program may modify this field.

`wordWrapHook` Pointer to the routine that handles word wrap. See Chapter 49, "TextEdit," for information about word wrap routines. Your program may modify this field.

`keyFilter` Pointer to the keystroke filter routine. See Chapter 49, "TextEdit," for information about keystroke filter routines. Your program may modify this field.

`theFilterRect` Defines a rectangle used by the generic filter procedure for some of its routines. See Chapter 49, "TextEdit," for information about generic filter procedures and their routines. Your program may modify this field.

`theBufferVPos` Vertical component of the current position of the buffer within the port for the TextEdit record, expressed in the local coordinates appropriate for that port. This value is used by some generic filter procedure routines. See Chapter 49, "TextEdit," for information about generic filter procedures and their routines. Your program may modify this field.

`theBufferHPos` Horizontal component of the current position of the buffer within the port for the TextEdit record, expressed in the local coordinates appropriate for that port. This value is used by some generic filter procedure routines. See Chapter 49, "TextEdit," for information about generic filter procedures and their routines. Your program may modify this field.

`theKeyRecord` Parameter block, in `KeyRecord` format, for the keystroke filter routine. Your program may modify this field.

`cachedSelcOffset`
Cached selection text offset. If this field is set to $FFFFFFFF, then the cache is invalid and will be recalculated when appropriate.

`cachedSelcVPos`
Vertical component of the cached buffer position, expressed in local coordinates for the output port.

`cachedSelcHPos`
Horizontal component of the cached buffer position, expressed in local coordinates for the output port.

`mouseRect` Boundary rectangle for multiclick mouse commands. If the user clicks the mouse more than once within the region defined by this rectangle within the time period defined for multiclicks, then TextEdit interprets those clicks as multiclick sequences (double- or triple-clicks). The user sets the time period with the Control Panel.

`mouseTime` System tickcount when the user last released the mouse button.

`mouseKind` Type of last mouse click:

0	Single click
1	Double-click
2	Triple-click

`lastClick` Location of last user mouse click.

`savedHPos` Cached horizontal character position. TextEdit uses this value to manage where it should display the caret on a line when the user presses the up or down arrow.

`anchorPoint` Defines the character from which the user began to select text for the current selection. When TextEdit expands the current selection (as a result of user keyboard or mouse commands, or at the direction of a custom keystroke filter procedure), it always does so from the `anchorPoint`, not `selectionStart` or `selectionEnd`.

Chapter 29 **Desk Manager Update**

This chapter documents new features of the Desk Manager. The complete reference to the Desk Manager is in Volume 1, Chapter 5 of the *Apple IIGS Toolbox Reference.*

New features in the Desk Manager

It is now possible for a new desk accessory (NDA) to be a modal dialog box. When an NDA is opened, it returns a pointer to its window. The Desk Manager saves this pointer and marks the NDA open. The current version of the Desk Manager checks the returned window pointer, and if its value is 0 (if it is a null pointer) then the Desk Manager does not mark the NDA open. Subsequent attempts to open the NDA simply select the open window until the NDA is closed. A programmer can therefore write an NDA that opens a modal dialog box when chosen. When the dialog box is dismissed, the NDA can be chosen again without having been explicitly closed.

Scrollable CDA menu

The classic desk accessory (CDA) menu is now scrollable. Previously, the menu held a maximum of 13 entries in a fixed display. Now, up to 249 desk accessories can be installed and displayed.

Scrolling takes place only on systems with 14 or more CDAs installed. When the menu is scrollable, the system displays a more message at each scrollable end of the menu. That is, if there are additional items above those currently visible, the more message appears at the top of the menu. Similarly, if there are more items below those currently visible, a more message appears at the bottom of the menu. Messages may be placed at both the top and bottom of the menu, if appropriate.

The new menu behaves somewhat differently from the old one. When the user returns to the CDA menu from an accessory, the name of that accessory is highlighted (previously, the Control Panel entry was highlighted). In addition, the user can no longer wrap from the bottom of the menu to the top, or vice versa.

The valid keystrokes for the CDA menu are:

Keystroke	Effect
Up Arrow	Moves selection box up one entry in the menu; no effect if at the top of the menu
Command–Up Arrow	Moves selection box up one page in the menu; no effect if at the top of the menu
Down Arrow	Moves selection box down one entry in the menu; no effect if at the bottom of the menu
Command–Down Arrow	Moves selection box down one page in the menu; no effect if at the bottom of the menu
Enter or Return	Selects the highlighted item
Esc	Selects Quit

Run queue

The **run queue** allows you to install tasks (**run items**) that need to be called periodically. You establish the periodicity of the call by managing a field in the run item header. The Desk Manager has two new system calls, `AddToRunQ` and `RemoveFromRunQ`, that allow you to install and remove run items from the queue.

The system examines the run queue at system task time, when the system is guaranteed to be free and all tools are available. For each run item in the queue, the system adjusts the `period` header field. If the specified time period has elapsed, the system then calls the run item.

The run queue is quite similar to the heartbeat queue, and should be used in its place.

Each run item must be preceded by a header formatted as follows:

$00	`Reserved`	Long—Used by system as link to next run queue item
$04	`period`	Word (unsigned)—Period to wait, in ticks
$06	`signature`	Word—Header signature, to ensure integrity—set to $A55A
$08	`Reserved`	Long—Used by system to know when item was last executed

`period` Specifies the minimum number of system ticks that are to elapse between run item executions. Each system tick represents 1/60th of a second. A value of 0 indicates that the item is to be called as often as possible. A value of $FFFF indicates that the item should never be called. While the run queue supports call frequencies up to approximately 60 per second, the timing is less accurate for periods shorter than one second.

△ **Important** Run item code must reset the `period` field before returning control to the system. Failure to do so will result in a `period` of 0, which will cause the item to be called constantly. △

`signature` Used by the system to ensure that the header is well formed. The value of this field must be $A55A.

The entry point must immediately follow the header. Run items need not check the busy flag, since the system is guaranteed to be free before any run item is invoked. However, run items must be careful to save and restore the operating environment, since they may be invoked from TaskMaster, as well as from an application. You should also be careful to either unload your run items at application termination, or ensure that remaining items are not purgeable.

While the run queue and heartbeat queue (see Chapter 14, "Miscellaneous Tool Set," in Volume 1 of the *Toolbox Reference* for information about the heartbeat queue) are somewhat similar, there are some significant differences. First, the run item header has an additional field (the second `Reserved` field). Second, the system does not remove items from the run queue when their `period` reaches 0.

Run queue example

Following is an example run item, which beeps the speaker every 15 minutes.

```
;
; RunQ example task that beeps every 15 minutes.
; It is provided in MPWIIgs assembler format. The first portion is the
; task header.
;
BeepHdr     Record
            ds.L 1      ; reserve 1 long for link to next runQ entry
period      dc.W $D2F0  ; number of 60th of a sec (54000=15 minutes)
            dc.W $A55A  ; signature used to test for queue integrity
            dc.L 0      ; used by desk mgr to keep track of the time
            EndR
;
; Now the actual code of the task goes here.
;
BeepTask    Proc
            with BeepHdr

            _SysBeep    ; beep the speaker once

            lda #$D2F0  ; and now recharge the period for next call
            sta >period ; NOTE:Use long addressing: DataBank unknown
            rtl         ; and to exit use an RTL
            EndP
```

The following code installs the above item into the run queue

```
            PushLong #BeepHdr
            ldx #$1F05
            jsl >$E10000
```

New Desk Manager calls

The following new Desk Manager calls support the run queue and desk accessory removal.

`AddToRunQ` $1F05

Adds the specified routine to the head of the run queue.

Parameters

Stack before call

Previous contents	
– *runItemPtr* –	Long—Pointer to run item to add
	<—SP

Stack after call

Previous contents	
	<—SP

Errors None

C

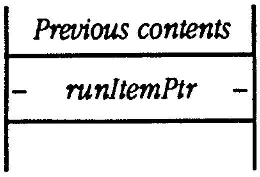
```
extern pascal void AddToRunQ(runItemPtr);

Pointer   runItemPtr;
```

`RemoveFromRunQ` $2005

Removes the specified run item from the run queue.

Parameters

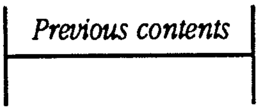
Stack before call



Long—Pointer to run item to remove

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void RemoveFromRunQ(runItemPtr);

Pointer   runItemPtr;
```

RemoveCDA $2105

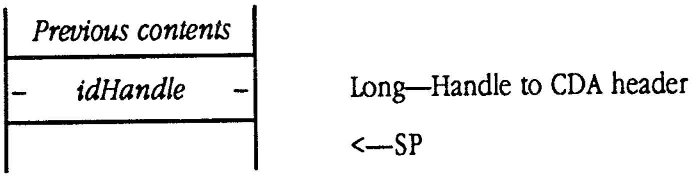
Removes the specified CDA from the Desk Manager CDA list. This routine does *not* dispose of the memory used by the DA.

This routine is the complement of `InstallCDA` (which is described in Chapter 5, "Desk Manager," in the *Toolbox Reference*).

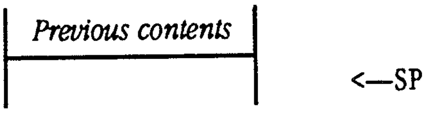
You should be very careful before issuing this call. Users generally install desk accessories for their own use; you should not spontaneously remove them from the system. Also, note that many desk accessories install other custom code (in the run queue, for example); you should not remove them unless you know that the other code has been removed, as well.

Parameters

Stack before call



Stack after call



Errors	$0510	`DaNotFound`	Specified desk accessory not found.

C

```
extern pascal void RemoveCDA(idHandle);

Handle    idHandle;
```

`RemoveNDA` $2205

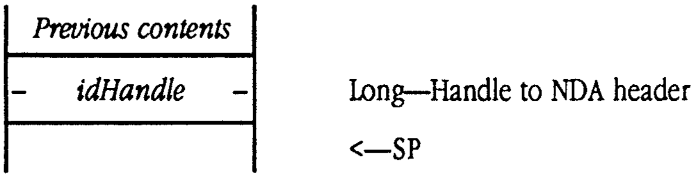
Removes the specified NDA from the Desk Manager NDA list. This routine does *not* dispose of the memory used by the DA.

This routine is the complement of `InstallNDA` (which is described in Chapter 5, "Desk Manager," in the *Toolbox Reference*).

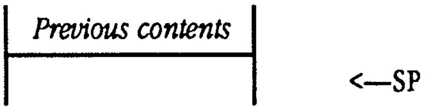
This call does not rebuild the Apple menu. Your application must rebuild the menu by issuing the `FixAppleMenu` tool call.

Parameters

Stack before call



Stack after call



Errors	$0510	`DaNotFound`	Specified desk accessory not found.

C

```
extern pascal void RemoveNDA(idHandle);

Handle    idHandle;
```

Chapter 30 **Dialog Manager Update**

This chapter documents new features of the Dialog Manager. The complete reference to the Dialog Manager is in Volume 1, Chapter 6 of the *Apple IIGS Toolbox Reference.*

Error corrections

This section explains changes that have been made to the Dialog Manager's documentation in the *Apple IIGS Toolbox Reference.*

- The documentation for `SetDItemType` on page 6-82 of the *Toolbox Reference* says that the call is used to change a dialog item to a different type. In fact, `SetDItemType` should be used only to change the *state* of an item from enabled to disabled or vice versa.
- The Dialog Manager does not support dialog item type values of `picItem` or `iconItem`, contrary to what the *Toolbox Reference* states in Table 6-3 on page 6-12.

Chapter 31 **Event Manager Update**

This chapter documents new features of the Event Manager. The complete reference to the Event Manager is in Volume 1, Chapter 7 of the *Apple IIGS Toolbox Reference.*

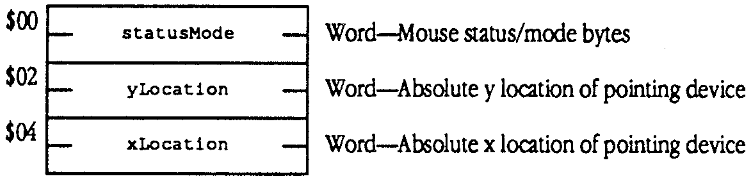
New features in Event Manager

The following sections discuss new features of the Event Manager.

Journaling changes

Previously, journaling did not capture operations that involved the `ReadMouse` Miscellaneous Tool Set call, because that call did not support journaling. As discussed in Chapter 39, "Miscellaneous Tool Set Update," in this book, `ReadMouse` has been changed to support journaling. As a result, journaling routines must now handle a new journal code.

When an application calls `ReadMouse`, and journaling is on, your journaling routine will be called with a journal code of 6 and *resultPtr* will point to a 6-byte record containing `ReadMouse` data. This record has the following format:


$00
statusMode
Word—Mouse status/mode bytes
$02
yLocation
Word—Absolute y location of pointing device
$04
xLocation
Word—Absolute x location of pointing device

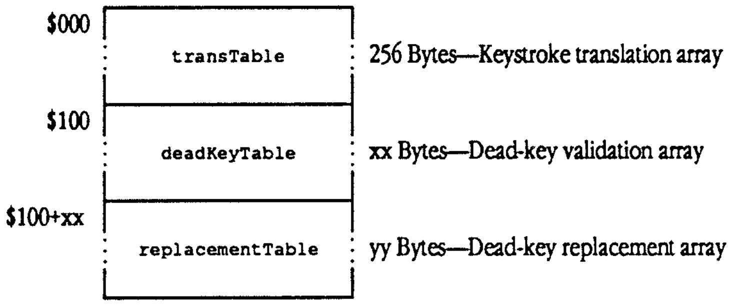
`statusMode` Mouse status and mode bytes, as described on pages 14-35 and 14-36 of the *Toolbox Reference.*

Keyboard input changes

The system now processes keyboard input through a translation routine, allowing Apple IIGS and Macintosh® keystrokes to match. The translation routine uses a resource-based keystroke translation table, which is identified by a unique resource ID. You can assign other tables to suit the needs of a particular language or keyboard. There are new Event Manager calls to read or write the current keyboard translation table resource ID.

Note that the system translates keystrokes before performing dead key replacements. To modify dead-key sequences you may find it easier to modify the appropriate `transTable` entry, since that table is more straightforward than the `deadKeyTable` and `replacementTable`.

The keystroke translation table must be formatted as follows:


$000
transTable
256 Bytes—Keystroke translation array
$100
deadKeyTable
xx Bytes—Dead-key validation array
$100+xx
replacementTable
yy Bytes—Dead-key replacement array

`transTable` This is a packed array of bytes used to map the ASCII codes produced by the keyboard into the character value to be generated. Each cell in the array directly corresponds to the ASCII code that is equivalent to the cell offset. For example, the `transTable` cell at offset $0D (13 decimal) contains the character replacement value for keyboard code $0D, which, for a straight ASCII translation table, is a Return character (CR). The `transTable` cells from 128 to 255 ($80 to $FF) contain values for Option-key sequences (such as Option-S).

`deadKeyTable` This table contains entries used to validate dead keys. Dead key refers to keystrokes used to introduce multikey sequences that result in single characters. For example, pressing Option-u followed by e yields an e with an umlaut. There is one entry in `deadKeyTable` for each defined dead key. The last entry_must be set to $0000. Each entry must be formatted as follows:

`deadKey`	Byte—Character code for dead key
`offset`	Byte—Offset from `deadKeyTable` into `replacementTable`

`deadKey` Contains the character code for the dead key. The system uses this value to check for user input of a dead key. The system compares this value with the first user keystroke.

`offset` Byte offset from beginning of `deadKeyTable` into relevant subarray in `replacementTable`, divided by 2. The system uses this value to access the valid replacement values for the dead key in question.

`replacementTable`
This table contains the valid replacement values for each dead key combination. This table is made up of a series of variable-length subarrays, each relevant to a particular dead key. The last entry in each sub-array must be set to $0000. Each entry in the `replacementTable` must be formatted as follows:

`scanKey`	Byte—Character code for dead key combination
`replaceValue`	Byte—Result character code for dead key combination

`scanKey` Contains a valid character code for dead key replacement. The system uses this field to determine whether the user entered a valid dead key combination. The system compares this value with the second user keystroke.

`replaceValue` Contains the replacement value for the character specified in `scanKey` for this entry. The system delivers this value as the replacement for a valid dead key combination.

New Event Manager calls

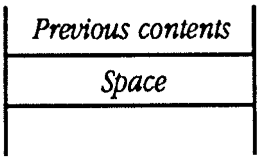
There are several new Event Manager calls, many concerning the new keyboard translation feature.

`GetKeyTranslation` $1B06

Returns the identifier for the currently selected keystroke translation table. Before setting a new translation table, your application should read and save the current identifier. When your application terminates, it should restore the previous keystroke translation table. Use the `SetKeyTranslation` call to modify the current identifier.

Parameters

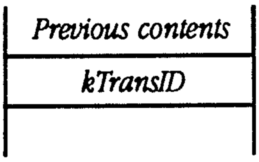
Stack before call



Word—Space for result

<—SP

Stack after call



Word—Keyboard translation identifier ($0000 to $00FF)

<—SP

Errors None

C `extern pascal Word GetKeyTranslation();`

`SetAutoKeyLimit` $1A06

Controls how repeated keystrokes are inserted into the event queue. The default value for the limit is 0, which specifies that auto-key events are inserted only if no other events are already in the queue. The *newLimit* parameter determines how many auto-key events must be in the event queue before `PostEvent` ceases to add them. For example, if *newLimit* is 0, then the default condition is maintained: `PostEvent` will not add auto-key events unless the queue is empty. However, if *newLimit* is 5, then `PostEvent` will add five auto-key events to the queue before it reverts to the rule that no more auto-key events are to be posted.

Parameters

Stack before call

Previous contents	
newLimit	Word—Limit for inserted auto-key events
	<—SP

Stack after call

Previous contents	
	<—SP

Errors None

C

```
extern pascal void SetAutoKeyLimit(newLimit);

Word      newLimit;
```

`SetKeyTranslation` $1C06

Sets a new keystroke translation table. Once set, the selected keystroke translation table stays in effect until this call is issued again, irrespective of application termination, system resets, or system power off. Before setting a new value for the keystroke translation table, your application should read and save the current value, using the `GetKeyTranslation` tool call. Your application should then restore that previous value when it is finished.

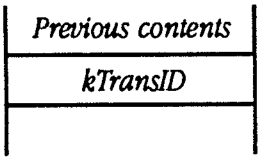
The system reads keystroke translation tables from resources of type $8021 and ID $0FFF06xx, where xx derives from the low-order byte of the *kTransID* parameter.

This call uses the current resource search path to find the specified resource. If you want your translation to stay in effect after your application has terminated, you must place the translation table resource in the system resource file.

If the system cannot find a resource corresponding to the value specified in *kTransID*, the keyboard defaults to the standard keystroke translation table ($00FF).

Parameters

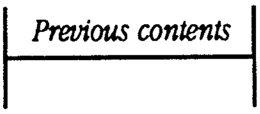
Stack before call



Word—Keystroke translation table identifier (low-order byte)

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void SetKeyTranslation(kTransID);

Word        kTransID;
```

kTransID The following are standard values for *kTransID*:

$0000	Use old-style Apple IIGS keyboard mapping
$00FF	Use standard keyboard remapping (makes Apple IIGS key sequences match Macintosh)

Chapter 32 **Font Manager Update**

This chapter documents new features of the Font Manager. The complete reference to the Font Manager is in Volume 1, Chapter 8 of the *Apple IIGS Toolbox Reference.*

New features in the Font Manager

- The current version of the Font Manager incorporates several changes. In previous versions, `FMStartUp` opened each font file in the FONTS folder, and constructed lists of information for all available fonts. These lists contained font IDs, font names, and so forth for every font in the FONTS folder. The present version of the Font Manager does this same work the first time it starts up, but caches all the information it compiles in a file called FONT.LISTS in the FONTS folder.

  The next time the Font Manager starts up, it checks all the creation and modification dates and times in font files against the information in FONT.LISTS. It compiles new FONT.LISTS information only if it finds new font files or other evidence of change. Otherwise, it simply starts up with the information stored in the FONT.LISTS file. In most cases, because it doesn't have to open every font file, the Font Manager can start up much more quickly.
- A bug has been fixed in the `ChooseFont` call. Previously, `ChooseFont` would hang the system if any update events were pending when the call was made. Now, `ChooseFont` will not hang the system under these circumstances; the system leaves update events in the Event Queue for processing by the application.
- In addition, the `ChooseFont` dialog now uses `NewWindow2`, with a control template that can be kept in a resource file. As a result, this dialog can be internationalized more easily.
- Scaled fonts may now contain more than 65,535 bytes of data. See Chapter 43, "QuickDraw II Update," later in this book for the layout of the new font record.
- A bug that corrupted the font family list has been fixed. This bug had varied symptoms, including incorrect font name displays in the Choose Font dialog and in the Font menu, and Font Manager crashes, among others.

New call

The new call `InstallWithStats` is provided to simplify the process of installing fonts. It allows an application to preserve certain information that is normally lost during font installation.

`InstallWithStats` $1C1B

Installs a font and returns information about that font. When an application requests the installation of a font, the Font Manager attempts to install the requested font, but it may not be available. In such cases, the Font Manager will install the closest match it can find to the requested font.

`InstallWithStats` installs a font just as if the application had called `InstallFont`, but it returns a `FontStatRec` in the buffer pointed to by *resultPtr*. This record contains the ID of the installed font, which may be different from the font requested. It also contains the purge status that the font had before it was installed. Since purge status can be changed by installation, this information can make it easier to restore a font's purge status. If you need to know an installed font's purge status, use `FindFontStats`.

Parameters

Stack before call

Previous contents	
– *desiredID* –	Long—Font ID of desired font
scaleWord	Word—Desired font size
– *resultPtr* –	Long—Pointer to buffer to receive `RontStatRec`
	<—SP

Stack after call

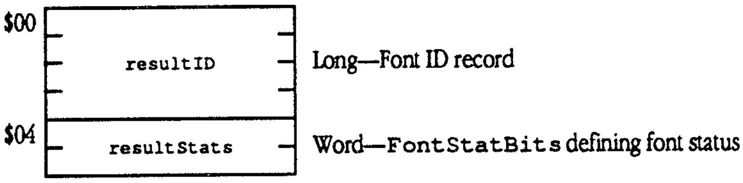
Previous contents	
	<—SP

Errors None

C

```
extern pascal void InstallWithStats(desiredID,
          scaleWord, resultPtr);

Long      desiredID;
Word      scaleWord;
Pointer   resultPtr;
```

resultPtr On return from `InstallWithStats`, the buffer pointed to by *resultPtr* will contain a `FontStatRec` formatted as follows



Chapter 33 **Integer Math Tool Set Update**

This chapter documents changes to the Integer Math Tool Set. The complete reference to Integer Math is in Volume 1, Chapter 9 of the *Apple IIGS Toolbox Reference.*

Clarifications

- The Long2Dec Integer Math tool call now correctly handles input long values that have the low-order three bytes set to zero. Previously, if the input long had its low-order three bytes set to zero, Long2Dec would always return a zero value, even if the high-order byte was non-zero.

Chapter 34 **LineEdit Tool Set Update**

This chapter documents new features of the LineEdit Tool Set. The complete reference to LineEdit is in Volume 1, Chapter 10 of the *Apple IIGS Toolbox Reference.*

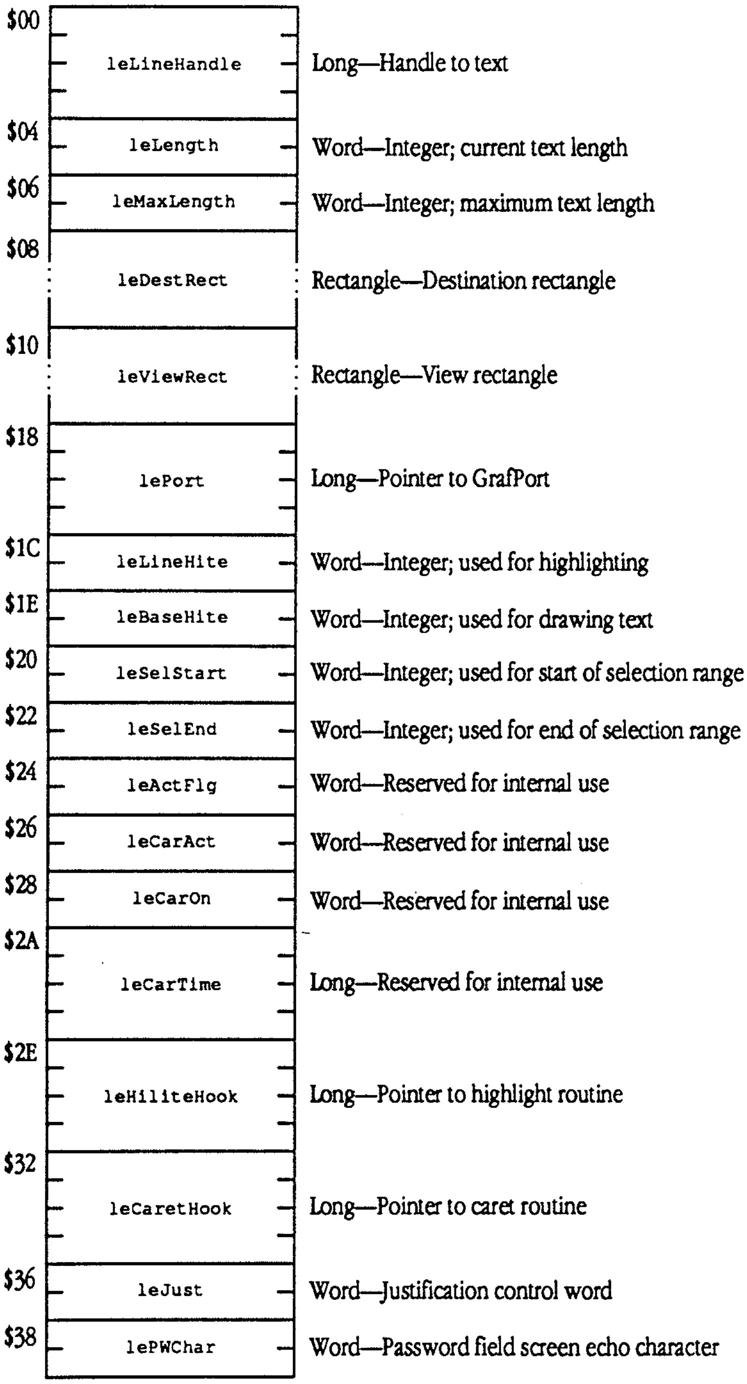
New features in LineEdit

- LineEdit now supports **password fields**. Password fields do not echo user input as typed. Instead, each input character is echoed with a special character. Your application can set the echo character; the default is asterisk (*).

  The LineEdit edit record has a new field, `lePWChar`, that supports the password feature. This field defines the screen echo character for password fields. It is located at the end of the edit record. The LineEdit record is now formatted as shown in Figure 34-1.

  To indicate that a LineEdit field is a password field, set the high-order bit of the maxSize field in the LineEdit control template to 1 (see "LineEdit control template" in Chapter 28, "Control Manager Update," earlier in this book for more information).
- Figure 34-1 shows the layout of the new LineEdit record.

■ **Figure 34-1** LineEdit Edit record (new layout)



$00	leLineHandle	Long—Handle to text
$04	leLength	Word—Integer; current text length
$06	leMaxLength	Word—Integer; maximum text length
$08	leDestRect	Rectangle—Destination rectangle
$10	leViewRect	Rectangle—View rectangle
$18	lePort	Long—Pointer to GrafPort
$1C	leLineHite	Word—Integer; used for highlighting
$1E	leBaseHite	Word—Integer; used for drawing text
$20	leSelStart	Word—Integer; used for start of selection range
$22	leSelEnd	Word—Integer; used for end of selection range
$24	leActFlg	Word—Reserved for internal use
$26	leCarAct	Word—Reserved for internal use
$28	leCarOn	Word—Reserved for internal use
$2A	leCarTime	Long—Reserved for internal use
$2E	leHiliteHook	Long—Pointer to highlight routine
$32	leCaretHook	Long—Pointer to caret routine
$36	leJust	Word—Justification control word
$38	lePWChar	Word—Password field screen echo character

`leMaxLength` Indicates the maximum text length allowed in the LineEdit field. Valid values range from 1 to 255. The high-order bit governs whether the field is a password field. If the bit is set to 1, then the field is a password field, and user input is echoed with character values specified by the contents of the `lePWChar` field.

`lePWChar` Defines the character to be echoed in password fields. This field contains the ASCII code for the echo character in its low-order byte. Default system value is asterisk (*).

New call

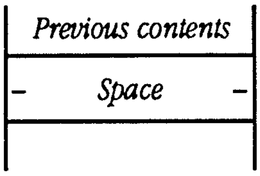
This new LineEdit tool call returns the address of the current LineEdit control definition procedure.

`GetLEDefProc` $2414

Returns the address of the current LineEdit control definition procedure. The system issues this call when the Control Manager starts up in order to obtain the address of the LineEdit control definition procedure. This call is not intended for application use.

Parameters

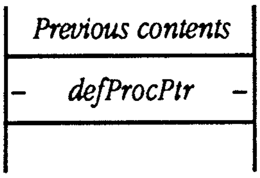
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to LineEdit control definition procedure

<—SP

Errors None

C
```
extern pascal Pointer GetLEDefProc();
```

Chapter 35 **List Manager Update**

This chapter documents new features of the List Manager. The complete reference to the List Manager is in Volume 1, Chapter 11 of the *Apple IIGS Toolbox Reference.*

Clarifications

- The *Apple IIGS Toolbox Reference* states that a disabled item of a list cannot be selected. In fact, a disabled item can be selected, but it cannot be highlighted. The List Manager provides the ability to select disabled (dimmed) items so that it is possible, for instance, for a user to select a disabled menu choice as part of a help dialog. To make an item unselectable, set it inactive (see "List Manager definitions" later in this chapter).
- Any List Manager tool call that draws will change fields in the GrafPort. If you are using List Manager tool calls you must set up the GrafPort correctly and save any valuable GrafPort data before issuing the call.
- Member text is now drawn in 16 colors in both 320 and 640 mode.
- Previous versions of List Manager documentation do not clearly define the relationship between the `listView`, `listMemHeight`, and `listRect` fields in the list record. To clarify this point, note that the following formula must be true for values in any list record:

  ```
  (listView *listMemHeight) +2=listRect.v2 -listRect.v1
  ```

  If you set `listView` to 0, the List Manager will automatically adjust the `listRect.v2` field and set the `listView` field so that this formula holds. Note that if you pass a 0 value for `listView` the bottom boundary of `listRect` may change slightly.

List Manager definitions

The following terms define the valid states for a list item.

inactive — Bit 5 of the list item's `memFlag` field is set to 1. Inactive items appear dimmed and cannot be highlighted or selected.

disabled — Bit 6 of the list item's `memFlag` field is set to 1. Disabled items appear dimmed and cannot be highlighted.

enabled — Bit 6 of the list item's `memFlag` field is set to 0. Enabled items appear normal and can be highlighted.

selected — Bit 7 of the list item's `memFlag` field is set to 1. This bit is set when a user clicks on the list item, or the item is within a range of selected items. A selected item appears highlighted only if it is also enabled.

highlighted	A member of a list appears highlighted only when it is both selected and enabled. This means that bit 7 of the `memFlag` field is set to 1 and bit 6 is set to 0. A highlighted member is drawn in the highlight colors.

New features in the List Manager

- The latest revision of the List Manager includes new versions of the tool calls that provide more flexible interfaces for application programmers in two ways. First, these new List Manager routines allow your application to pass an item number, rather than a list record pointer, to identify an item to process. This frees you from tracking pointer values, and allows you to focus on the more useful item number. Second, your application need no longer maintain the list record. All new tool calls allow you to identify the list by a handle to the list control record. List Manager returns this handle at `CreateList` or, preferably, `NewControl2` time.
- The `listType` field now supports a flag that governs where the scroll bar is to be created. Bit 2 of `listType` determines whether the scroll bar is created inside or outside of `listRect`. If the bit is set to 1, the List Manager adjusts the right side of `listRect` to accommodate the scroll bar, creates the scroll bar inside of the adjusted `listRect`, and then sets the flag to 0. If the bit is set to 0, the scroll bar resides outside `listRect`. This works the same way with old-style control records.

△ **Important** When using resources with the List Manager, be careful to define the memory referenced by `listRef` (see "NewList2" later in this chapter) as unpurgeable if you plan to use the `SortList` call. Otherwise, in response to a memory allocation request, the sorted list may be purged from memory. Then, when your application next issues a List Manager call, the system will reload the unsorted list. △

New List Manager calls

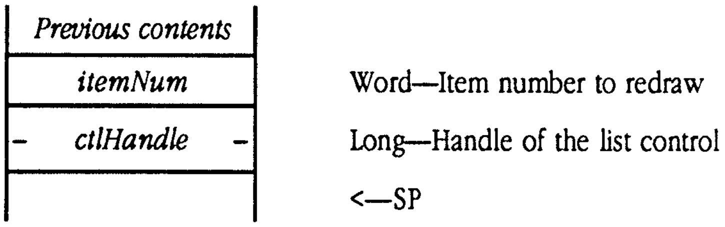
The following new List Manager calls support a new, more flexible programming interface. In general, these calls provide the same functionality as the old versions.

`DrawMember2` $111C

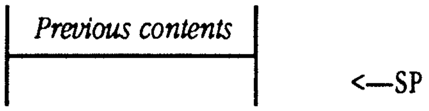
Draws one or all members of a specified list. If your application goes directly to the member record to change the state of a member, the application should then call `DrawMember` or `DrawMember2`. Unlike `DrawMember`, this call accepts an item number specification for the member to draw. Passing an item number of 0 causes the List Manager to redraw the entire list.

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal void DrawMember2(itemNum, ctlHandle);

Word      itemNum;
Handle    ctlHandle;
```

`NewList2` $161C

Resets the list control according to a specified list record. Your application passes the parameters controlling the creation of the list on the stack, rather than in a list record (as with `NewList`).

Parameters

Stack before call

Previous contents	
– *drawPtr* –	Long—Pointer to member draw routine; NIL for default routine
listStart	Word—Item number of first displayed list member
– *listRef* –	Long—Reference to list
listRefDesc	Word—Descriptor for listRef
listSize	Word—Number of items in the list
– *ctlHandle* –	Long—Handle of the list control returned by `NewControl2`
	<—SP

Stack after call

Previous contents	
	<—SP

Errors None

C

```
extern pascal void NewList2(drawPtr, listStart,
          listRef, listRefDesc, listSize,
          ctlHandle);

Pointer   drawPtr;
Word      listStart, listRefDesc, listSize;
Long      listRef;
Handle    ctlHandle;
```

drawPtr Pointer to custom list member drawing routine. NIL value causes the List Manager to use its standard routine.

listStart Item number of first list item to display. A value of $FFFF tells the List Manager to use the value currently stored in the list control record. Never set this parameter to 0.

listRef Reference (pointer, handle, or resource ID) to the list. The value of *listRefDesc* governs how the List Manager interprets this field. A value of $FFFFFFFF tells the List Manager to use the value currently stored in the list control record.

listRefDesc Defines the type of reference stored in *listRef*.

0	*listRef* reference is a pointer
1	*listRef* reference is a handle
2	*listRef* reference is a resource ID
$FFFF	no change

◆ *Note:* If you set either *listRef* or *listRefDesc* to –1, then you must set the other field to the same value.

listSize Number of entries in the list. A value of $FFFF tells the List Manager to use the value currently stored in the list control record.

NextMember2 $121C

Searches a specified list record, starting with a specfied item, and returns the item number corresponding to the next selected item. This call accepts an item number and control handle as input. If you pass an item number of 0, the List Manager starts its search from the beginning of the list.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
itemNum	Word—Item number of starting point for search
– *ctlHandle* –	Long—Handle of the list control
	<—SP

Stack after call

Previous contents	
itemNum	Word—Item number of selected member; 0 if no more
	<—SP

Errors None

C

```
extern pascal Word NextMember2(itemNum, ctlHandle);

Word      itemNum;
Handle    ctlHandle;
```

ResetMember2 $131C

Searches a specified list control, starting with the first list member, and returns the item number of the first selected member in the list. If the user has not selected a member, then the returned item number is 0. This call accepts a control handle as input.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
– *ctlHandle* –	Long—Handle of the list control
	<—SP

Stack after call

Previous contents	
itemNum	Word—Item number of selected member; 0 if no more
	<—SP

Errors None

C

```
extern pascal Word ResetMember2(ctlHandle);

Handle    ctlHandle;
```

`SelectMember2` $141C

Selects a specified member, deselects any other selected members of the list, and scrolls the list display so that the specified member is at the top of the display. This call accepts a control handle and an item number as input.

Parameters

Stack before call

Previous contents	
itemNum	Word—Item number of member to select
– *ctlHandle* –	Long—Handle of the list control
	<—SP

Stack after call

Previous contents	
	<—SP

Errors None

C

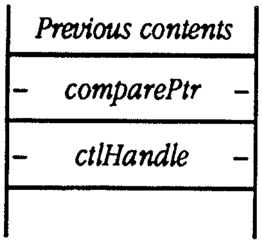
```
extern pascal void SelectMember2(itemNum,
          ctlHandle);

Word      itemNum;
Handle    ctlHandle;
```

SortList2 $151C

Alphabetizes a specified list by rearranging the array of member records. This call accepts a control handle and a pointer to a custom comparison routine as input.

Parameters

Stack before call

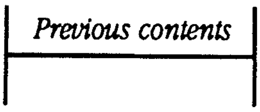


Long—Pointer to comparison routine; NIL for standard compare

Long—Handle of the list control

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void SortList2(comparePtr, ctlHandle);

Pointer   comparePtr;
Handle    ctlHandle;
```

Chapter 36 **Memory Manager Update**

This chapter documents new features of the Memory Manager. The complete reference to the Memory Manager is in Volume 1, Chapter 12 of the *Apple IIGS Toolbox Reference.*

Error correction

On page 12-10 of the *Toolbox Reference*, Figure 12-7 shows the low-order bit of the User ID is reserved. This is not correct. The figure should show that the `mainID` field comprises bits 0–7, and that the `mainID` value of $00 is reserved.

Clarification

The *Toolbox Reference* documentation of the `SetHandleSize` call ($1902) states "If you need more room to lengthen a block, you may compact memory or purge blocks." This is misleading. In fact, to satisfy a request the Memory Manager will compact memory or purge blocks in order to free sufficient contiguous memory. Therefore, the sentence should read "If your request requires more memory than is available, the Memory Manager may compact memory or purge blocks, as needed."

New features in the Memory Manager

The Memory Manager allocates handles much faster than before. The Memory Manager remembers the last handle allocated, and starts its search for new memory from that location, resulting in improved allocation time.

Out-of-memory queue

The **out-of-memory queue** allows application code to gracefully recover from low-memory conditions in the system. The out-of-memory queue consists of a series of **out-of-memory routines**, which are created and installed by application programs. When the Memory Manager cannot create a handle from memory currently available, it calls each of the out-of-memory routines. These routines can then either free up memory that is not crucial to the function of an application, or notify the application that it is time to tell the user to save and exit.

When the Memory Manager encounters a low-memory condition, it performs the following steps:

1. Invokes each out-of-memory routine until a routine reports that it has freed enough memory to satisfy the request. If a routine does free enough memory, the Memory Manager then allocates the handle and returns control to the calling application.
2. Compacts memory and retries the allocation. If the allocation is successful, the Memory Manager returns control to the calling application.
3. Purges Level 3 handles. If this frees enough memory, the Memory Manager compacts memory, allocates the handle, and returns to the calling application.
4. Purges Level 2 handles. If this frees enough memory, the Memory Manager compacts memory, allocates the handle, and returns to the calling application.
5. PurgesLevel 1 handles. If this frees enough memory, the Memory Manager compacts memory, allocates the handle, and returns to the calling application.
6. Again invokes each out-of-memory routine. If a routine frees enough memory, the Memory Manager allocates the handle and returns to the application. Otherwise, the Memory Manager reports an out-of-memory condition to the application.

Note that the Memory Manager may invoke an out-of-memory routine twice during the same low-memory condition. In the invokation parameter block for an out-of-memory routine, the Memory Manager passes a flag indicating whether this is the first or second time through the out-of-memory queue. By examining this flag, routines can react differently based upon the urgency of the low-memory condition.

Any application, desk accessory, or init that installs an out-of-memory routine must also remove that routine from the out-of-memory queue. Add routines to the queue with the `AddToOOMQueue` tool call; remove them with the `RemoveFromOOMQueue` tool call.

Out-of-memory routines may use any Memory Manager tool call. However, routines that must issue calls that allocate memory (such as `NewHandle`) should reserve the needed memory at initialization, so that the space will be available during a low-memory condition. For example, if you want your out-of-memory routine to save some user data to disk before purging a memory block, your application should reserve enough memory for the file open before installing the routine. When the routine gains control, it can then free the reserved memory, issue the file system calls, and purge the unneeded application memory without creating a recursive low-memory condition. See the code example for sample application and out-of-memory routine code.

An out-of-memory routine must be preceded by a header formatted as follows:

$00	Reserved	Long—Used by system as link to next queue item
$04	version	Word—Must be set to 0
$06	signature	Word—Header signature, to ensure integrity—set to $A55A

`version` Allows system to discriminate between current and future types of out-of-memory routines. Must be set to 0.

`signature` Used by the system to ensure that the header is well-formed. The value of this field must be $A55A.

The out-of-memory routine code must immediately follow the `signature` word. If the Memory Manager finds an invalid header for any out-of-memory routine, it terminates with a system death error code of $0209.

When the out-of-memory routine gets control, the Memory Manager will have formatted the input stack as follows:

Previous contents	
Space	Long—Space for result
bytesNeeded	Long—Number of bytes the Memory Manager needs
stage	Word—Flagword indicating stage of low-memory condition
RTLAddr	3 Bytes—Return address
	<—SP

stage Indicates the stage of the low-memory condition. This flag allows the routine to determine whether this is the first or second invokation for this condition. If the field is set to 0, then this is the first invokation, and the Memory Manager has not done anything else. If the field is set to 1, then this is the second invokation for this low-memory condition, and the Memory Manager will report an out-of-memory condition to the calling application if it cannot find enough memory to satisfy the request.

The out-of-memory routine must strip off the input parameters and return the number of bytes freed in the space provided. On exit, therefore, the routine should format the stack as follows:

Previous contents	
– *amountFreed* –	Long—Number of bytes of memory freed by routine
– *RTLAddr* –	3 Bytes—Return address
	<—SP

Out-of-memory Routine example

The following code example has two parts: the first shows how your application can install a routine in the out-of-memory queue; the second is a sample out-of-memory routine.

```
;
; first allocate a handle with enough memory for our low memory exit
; this example will use a 16k handle

            pha                 ; room for result
            pha
            PushLong #$4000     ; size of handle
            PushWord MyID       ; my applications ID
            PushWord #0         ; no bits set, unlocked and moveable
            PushLong #0         ; address (Not used)
            _NewHandle
            PullLong ResvHand ; and pull off the reserve handle

            PushLong #MyOOMRtn        ; address of the OOM header
            _AddToOOMQueue

            stz OOMFlag         ; zero our low memory indicator
```

Note that this application maintains the `OOMFlag` field in its global storage area.

The following is the actual out-of-memory queue entry itself. It has been written for the MPW™ Apple IIGS assembler.

```
;
; This is the OOMQueue header for our routine
;
MyOOMRtn     Record
             dc.L 0             ; used by queue manager
             dc.W 0             ; OOMEntry version
             dc.W $A55A         ; queue entry signature
             EndR
;
; Now for my out-of-memory routine
;
MyOOM        proc
;
; first set up the equates for the stack frame passed to us by the
;            memory mgr
;
RTLAdr       equ 1              ; return address we will go back to
Stage        equ RTLAdr+3       ; indicates when called
BytesNeeded equ Stage+2         ; number of bytes the mem mgr needs
Result       equ BytesNeeded+4 ; return number of bytes freed
;
; before we start we should zero out the result
;
             lda #0
             sta Result,s       ; zero the result on the stack
             sta Result+2,s
;
; Since this routine can be called before and after purging data
; we want to wait till the memory manager has purged everything it can
; before we panic so the first thing we do is test the Stage
;
             lda Stage,s        ; get the passed stage
             beq OOMEnd         ; if 0 then don't free anything
;
; Now that we know that the memory manager has tried everything else,
;            we test to see if we have done this before by testing
;            the OOMFlag
;
             lda >OOMFlag       ; must use long address DB=unknown
             bne OOMEnd         ; if non-zero then memory already free
```

```
;
; since we know that we have not freed the reserve memory yet
;            we will do so now and set the flag.
;
            PushLong >ResvHand      ; handle to our reserve space
            _DisposeHandle     ; and dispose of it

            lda #$FFFF         ; now set our flag to true
            sta >OOMFlag       ; so that the event loop knows low mem

            lda #$4000         ; and signal the memory manager how
            sta Result,s       ; much mem we freed
;
; Now return to the memory manager first adjusting the stack to remove
the
;            passed params
;
OOMEnd
            LongA Off          ; turn on 8 bit accumulator
            SEP #$20

            pla                ; load the return address for safe
            ply                ; keeping for a sec

            plx                ; now pull off 6 bytes of parameters
            plx
            plx

            phy                ; put the return addr back
            pha
            LongA On           ; turn on 16 bit accumulator
            REP #$20

            RTL                ; and return
```

New Memory Manager calls

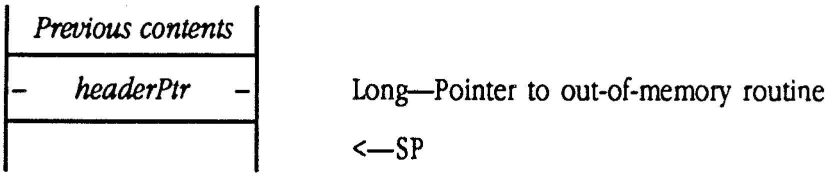
`RealFreeMem` is a new Memory Manager call designed to provide accurate information about available memory. Other new Memory Manager calls support the out-of-memory queue.

`AddToOOMQueue` $0C02

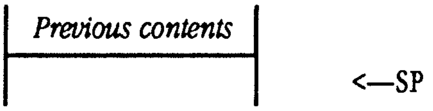
Adds the specified out-of-memory routine to the head of the out-of-memory queue. The input routine pointer should contain the address of the routine header block.

Parameters

Stack before call



Stack after call



Errors	$0381	`invalidTag`	Correct signature value not found in header

C

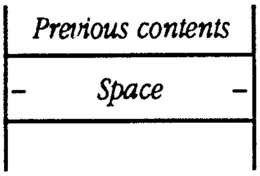
```
extern pascal void AddToOOMQueue(headerPtr);

Pointer    headerPtr;
```

RealFreeMem $2F02

Returns the number of bytes in memory that are free, plus the number that could be made free by purging. `FreeMem` only returns the number of bytes that are actually free, ignoring memory that is occupied by unlocked purgeable blocks. Since unlocked blocks of allocated memory can be freed by purging, `FreeMem` does not provide an accurate picture of the memory that is actually available. `RealFreeMem` provides a more accurate value.

Parameters

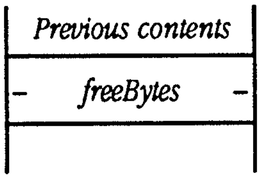
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Number of available bytes in memory

<—SP

Errors None

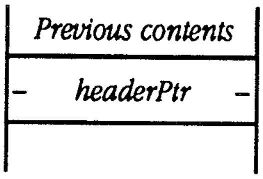
C `extern pascal Long RealFreeMem();`

`RemoveFromOOMQueue` $0D02

Removes the specified out-of-memory routine from the queue. The input routine pointer should contain the address of the routine header block.

Parameters

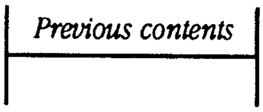
Stack before call



Long—Pointer to out-of-memory routine

<—SP

Stack after call



<—SP

Errors	$0381	`invalidTag`	Correct signature value not found in header
	$0380	`notInList`	Specified routine not found in queue

C

```
extern pascal void RemoveFromOOMQueue(headerPtr);

Pointer   headerPtr;
```

Chapter 37 **Menu Manager Update**

This chapter documents new features of the Menu Manager. The complete reference to the Menu Manager is in Volume 1, Chapter 13 of the *Apple IIGS Toolbox Reference.*

Error corrections

- In the description of the `SetSysBar` tool call (pages 13-86 and 13-3), the *Toolbox Reference* states that, after an application issues this call, the new system menu bar becomes the current menu bar. This is incorrect. Your application must issue the `SetMenuBar` tool call to make the new menu bar the current menu bar.
- In the definition of the menu bar record (pages 13-17–18), the *Toolbox Reference* shows that bits 0–5 of the `ctlFlag` field are used to indicate the starting position for the first title in the menu bar. This is incorrect. The `ctlHilite` field defines the starting position for the first title. Note further that the entire `ctlHilite` field is used in this manner. The documented purpose of the `ctlHilite` field (number of highlighted titles) is not supported by the Menu Bar record.

Clarifications

- The `SetBarColors` tool call changes the color table for all menu bars in a window. If you want to use separate color tables for different menu bars, your application must build a menu bar color table and modify the `ctlColor` field of the appropriate control record to point to this custom color table. See "SetBarColor" in Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference* for the format and contents of a menu bar color table.
- The description of the `InsertMenu` tool call should also note that your application must call `FixMenuBar` before calling `DrawMenuBar` in order to display the modified menu bar.
- The description of the `InitPalette` tool call in the *Toolbox Reference* should also note that the call changes color tables 1 through 6 to correspond to the colors needed for drawing the Apple logo in its standard colors.
- The `CalcMenuSize` call uses the *newWidth* and *newHeight* parameters to compute a menu's size. These parameters may contain the width and height of the menu, or may contain the values $0000 or $FFFF. A value of $0000 tells `CalcMenuSize` to calculate the parameter automatically. A value of $FFFF tells it to caiculate the parameter only if the current setting is 0.

The effect of all three uses:

- □ **Pass the new value.** The value passed will become the menu's size. Use this method when a specific menu size is needed.
- □ **Pass $0000.** The size value will be automatically computed. This option is useful if menu items are added or deleted, rendering the menu's size incorrect. The menu's height and width can be automatically adjusted by calling `CalcMenuSize` with *newWidth* and *newHeight* equal to $0000.
- □ **Pass $FFFF.** The width and height of a menu is 0 when it is created. `FixMenuBar` calls `CalcMenuSize` with *newWidth* and *newHeight* equal to $FFFF to calculate the sizes of those menus with heights and widths of 0.

New features in the Menu Manager

This section lists several new features of the Menu Manager, and some information that was not previously clear.

- Menus in windows can now display the Apple character (ASCII $14), though it will not be multicolored.
- Menus now use their outline color for lines that separate menu items.
- The `NewMenuBar` call automatically sets bit 31 of the `ctlOwner` field in the menu bar record to 1, if the designated menu bar is a window menu bar (the value passed for the window is not 0).
- The default position for the first menu title in a menu bar is 10 pixels indented from the left edge of the screen, in 640 mode; in 320 mode the item is indented 5 pixels.
- The Menu Manager's justification procedures adjust for menu bars in windows. Menus will be moved to the left if they would otherwise appear to the right of the menu bar's right end.
- The default menu bar has the following coordinates: top = 0; left = 0; height = 13; width = the width of the screen.
- `MenuShutDown` does not return an error if the Menu Manager has already been shut down.
- Your application can now create empty menus. To create an empty menu, set the first byte in the first menu line item to either null ($00) or return ($0D), signifying the end of the menu definition. For example:

```
          dc.b   '$$ Empty Menu \N1',$00  ; menu title and ID
          dc.b   $00                      ; first character in first
                                          ;  item to null (or return)
                                          ;  indicates end of menu def.
```

  Or, using a menu template:

```
     EmptyMenu
          dc.W   0                        ; version
          dc.W   1                        ; menu id
          dc.W   0                        ; menu flag
          dc.L   Title                    ; menu title
          dc.L   $00000000                ; indicates end of item list
     Title str   'Empty Menu'
```

- The Menu Manager now correctly supports outline and shadow text styles. As a result, the existing *Toolbox Reference* description of the `SetMItemStyle` tool call, and the menu text style word defined in that description, is now correct.

  In addition, the Menu Manager now supports two new special characters for menu definition:

  O Outline the text
  S Shadow the text

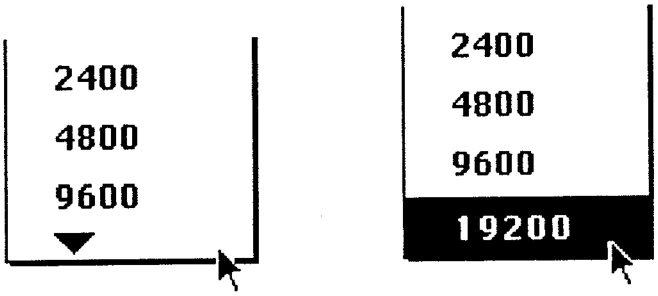
  Other special characters are listed on page 13-14 of Volume 1 of the *Toolbox Reference.* Note that this feature requires the QuickDraw II Auxiliary Tool Set.

- Menus now scroll up or down if their items will not fit on the screen. When a menu is scrollable in a direction, an arrow indicator appears at the appropriate end of the menu, signifying that there are more items available. See Figure 37-1.

  The indicator does not highlight, but the menu contents scroll when the user drags over it. When the last item is displayable, the indicator disappears. Indicators may appear at both the top and bottom of a menu, if appropriate.

  Menus scroll at two speeds, depending upon where the user drags in the indicator. If the user drags within the first five pixels of an indicator, scrolling runs at its slow speed. Dragging anywhere beyond this point results in fast scrolling.

- **Figure 37-1** Scrolling menus with indicator at bottom



◆ *Note:* If your application defines menus within a moveable window, dragging that window close to the bottom of the screen may force some of the menus to be scrollable. If there is not room for three visible items (up and down indicators and one menu item), then the menu will drop below the visible screen area.

- The menu record has been slightly modified. The `firstItem` and `numOfItems` byte fields have been combined into a single word field, `numOfItems`, at offset $0C into the record. This field specifies the number of items in the menu.
- Bit 8 of the flag field in the menu record is now defined as the `alwaysCallmChoose` flag. When this flag is set to 1, the Menu Manager calls the mChoose routine in the defProc for a custom menu even when the mouse is not within the menu rectangle. This feature supports tear-off menus.
- Keyboard equivalents and check marks now appear in plain text regardless of the style of the associated menu item.
- The Menu Manager can now handle large fonts in menus.

Menu caching

The current version of the Menu Manager introduces new menu caching features. Menu caching is designed to provide faster display of menus under certain circumstances. When a menu is drawn on the screen, the area of the screen that it covers is copied into a buffer. When the menu goes away, the contents of the buffer are copied back to the screen.

With the menu caching feature, when the saved screen image is copied back to the screen, the menu that goes away is copied into the buffer. In other words, the Menu Manager swaps the menu image with the screen image. Therefore, the next time that menu is pulled down, the Menu Manager can copy it from the buffer instead of drawing a new image.

If the menu image changes—for example, an item is disabled or the items on the menu change—then the cached image is inaccurate, and the Menu Manager must redraw the menu. In those cases where a menu image does not change, the menu bar can respond to the user more quickly.

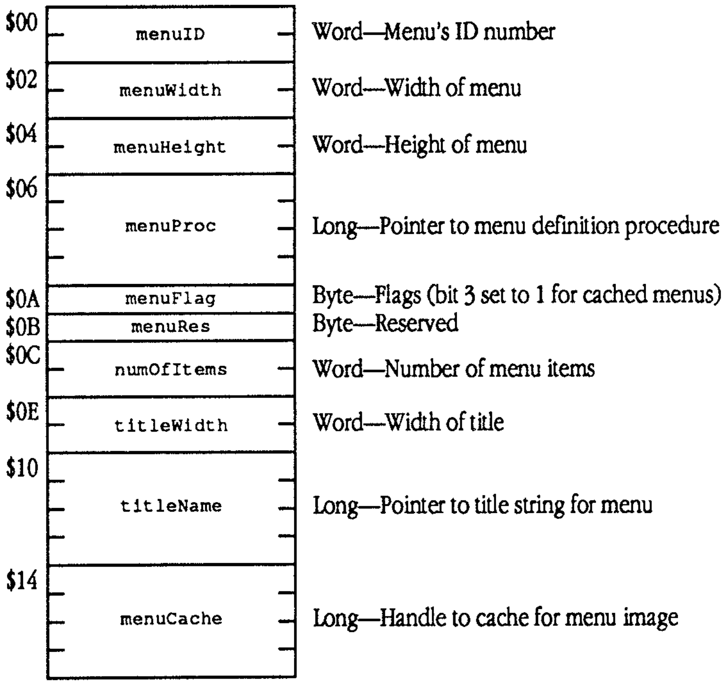
Menu caching should not increase memory requirements, because menu images are purgeable when not displayed on the screen.

This menu caching scheme should work properly with all existing standard menus. You will have to alter custom menus, however, so that they can take advantage of menu caching. Custom menus will still function normally, as long as they do not change the menu record directly, but they will not be able to take advantage of the menu caching scheme to speed up display.

Caching does not work with menus in windows, so the `InsertMenu` call automatically disables caching for such menus.

Caching with custom menus

Bit 3 of the `MenuFlag` field in a menu record indicates whether a menu's definition procedure knows about caching. A value of 1 indicates that the menu in question is cacheable. A custom menu that uses caching must definea menu record that sets this flag and allocates an extra field, a handle to the cache in which the menu image will be stored, as shown in the following figure.



$00	menuID	Word—Menu's ID number
$02	menuWidth	Word—Width of menu
$04	menuHeight	Word—Height of menu
$06	menuProc	Long—Pointer to menu definition procedure
$0A	menuFlag	Byte—Flags (bit 3 set to 1 for cached menus)
$0B	menuRes	Byte—Reserved
$0C	numOfItems	Word—Number of menu items
$0E	titleWidth	Word—Width of title
$10	titleName	Long—Pointer to title string for menu
$14	menuCache	Long—Handle to cache for menu image

Pop-up menus

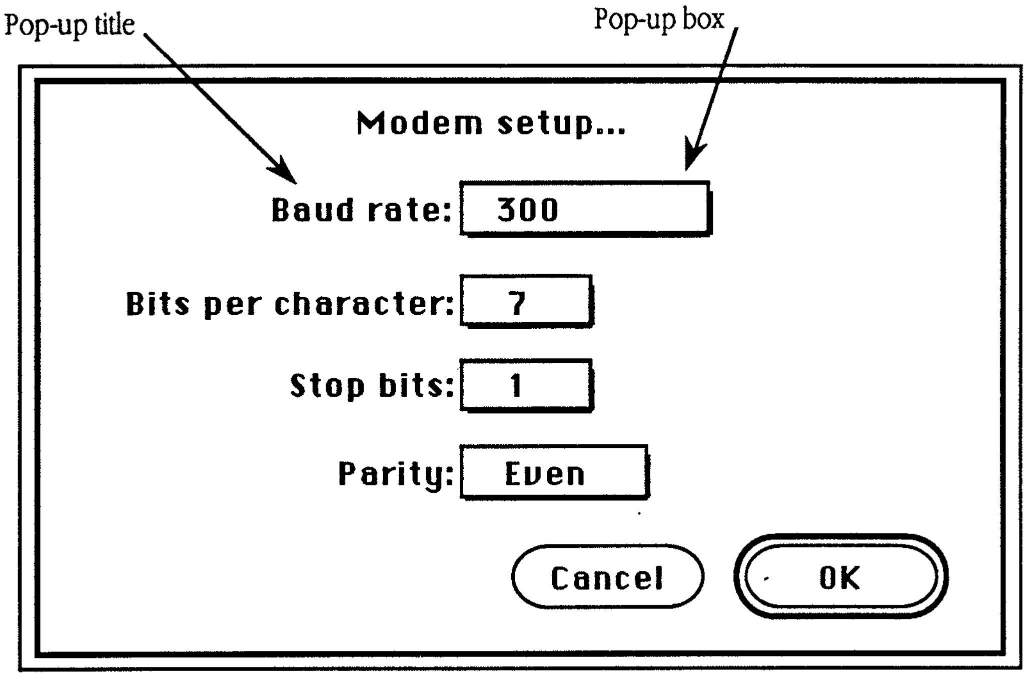
Menu Manager now supports **pop-up menus**. Pop-up menus exist in a window, not in the menu bar. Figure 37–2 shows a window with pop-up menus. The screen representation of a pop-up is a box with a one-pixel-thick drop shadow. When the user clicks the mouse inside the pop-up box, the menu appears, with the current value highlighted under the arrow, as shown in Figure 37–3. If the menu has a title, the title is highlighted whenever the menu is visible.

Pop-up menus work in the same way as other menus: the user can move around within the menu, select an item by positioning over it, or not select any item by dragging outside the menu. Pop-up menus support scrolling, if it is needed to view all the menu items. Pop-up menus are useful for setting values or choosing from lists of related values.

Pop-up menus support most of the standard features and calls available with standard menus:

- Pop-up menu items support keystroke equivalents. Pop-up menus will display the equivalent (Apple logo with character). Note that if a pop-up item's keystroke equivalent conflicts with a standard menu item equivalent, the pop-up menu may not receive the keystroke. TaskMaster passes the keystroke to the system first, unless the `tmControlKey` flag in the `wmTaskMask` field of the task record is set to 0 (do not pass keys to controls in the active window).
- Pop-up menu items can be dimmed to indicate that they are disabled and cannot be chosen.
- Each item in a pop-up menu can have its own text styles.

■ **Figure 37-2** Window with pop-up menus



Modem setup...

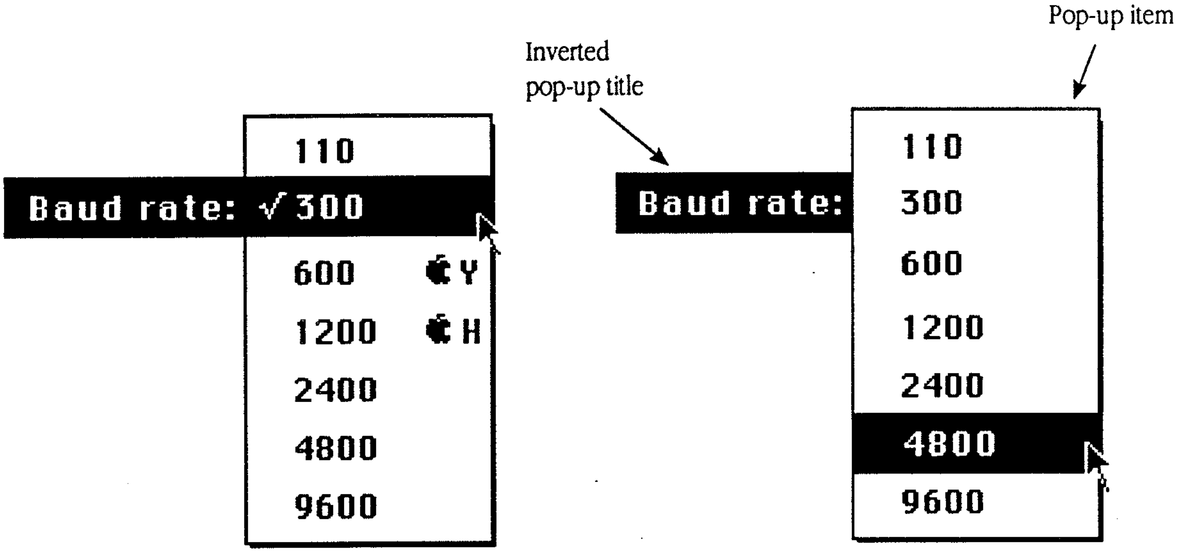
Baud rate: 300

Bits per character: 7

Stop bits: 1

Parity: Even

- **Figure 37-3** Dragging through a pop-up menu

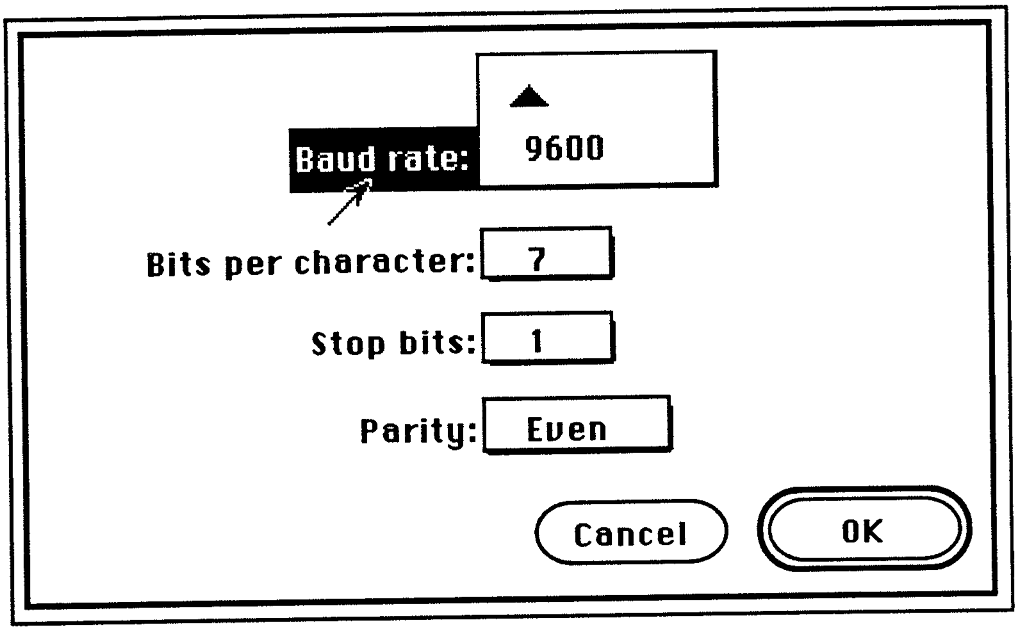


Pop-up menu scrolling options

There are two types of pop-up menus, which are distinguished by their support for scrolling: **type 1** pop-up menus and **type 2** pop-up menus.

The Menu Manager determines the size rectangle into which to draw a **type 1** pop-up menu based upon the relative position of the current item within the menu and the window constraints of the pop-up menu (see Figure 37-4). The Menu Manager draws the pop-up menu with the current item highlighted and positioned adjacent to the menu title. The menu extends up and down only as far as is necessary to display the remaining items in each direction, and indicators as appropriate, within the boundary rectangle for the window. Therefore, with type 1 pop-up menus, it is possible to obtain a display such as that shown in Figure 37-4, where the user can display only a single item.

- **Figure 37-4** Type 1 pop-up menu



Baud rate: 9600

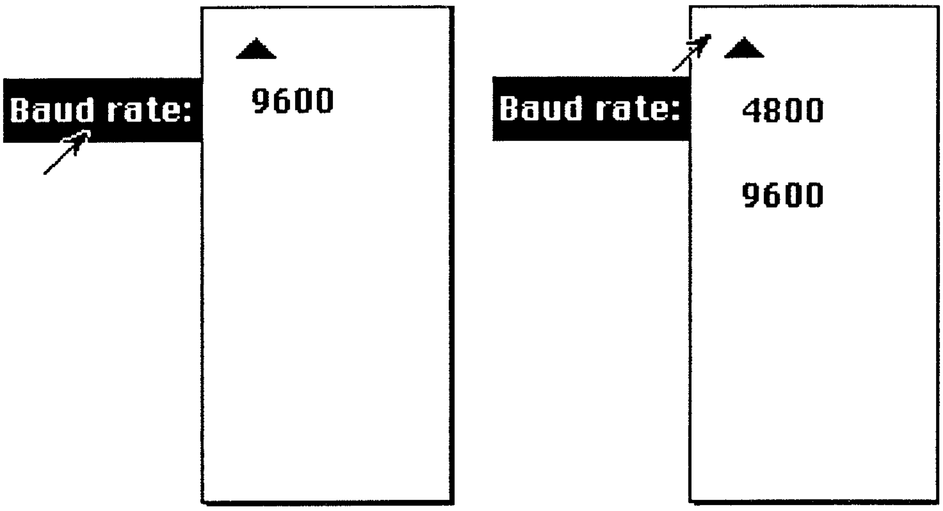
Bits per character: 7

Stop bits: 1

Parity: Even

When the Menu Manager needs to make a **type 2** pop-up menu scrollable, it creates a menu that is long enough to receive all the menu items, within the bounds of the screen. In this manner, the user never sees a menu with too few item lines to be useful. Figure 37–5 shows how the Baud rate pop-up menu from Figure 37–4 would appear if it had been defined as a type2 pop-up menu.

- **Figure 37-5** Type 2 pop-up menu



By dragging over the scroll indicator, the user can eventually scroll all menu items that will fit on the screen into view, regardless of menu proximity to top or bottom of screen.

How to use pop-up menus

Your application can define pop-ups in two ways: either as controls or menus.

If your application defines its pop-ups as controls, using the `NewControl2` Control Manager tool call, then drawing, updating, resizing, and tracking will all be handled by TaskMaster and `TrackControl` automatically. TaskMaster will also deal with any keystroke equivalents you have defined. See Chapter 28, "Control Manager Update," for details on how to create a pop-up control template and invoke `NewControl2`.

If, on the other hand, your application defines its pop-ups as menus, it gains flexibility but has more responsibility. Your application must draw the pop-up box and title, highlight the title, recognize mouse-down events in the pop-up box or title, and change the current entry in response to user choices. Your application must also deal with keystroke equivalents. Once your program detects a mouse-down event inside the Pop–up box or title, it must call `PopUpMenuSelect` to display the menu and track the mouse. This call returns the item ID of the selected item (0 if none selected). Your program can use this item ID to determine which item was selected. Your program must pass this item ID to `PopUpMenuSelect` the next time the user clicks in the pop-up.

◆ *Note:* When you create a pop-up control with `NewControl2`, calling `SetMItem`, `SetMItem2`, `SetMItemName`, `SetMItemName2`, `SetMItemStyle`, `SetMenuTitle` or `SetMenuTitle2` does not change the appearance of the pop-up until the pop-up is redrawn. If your application changes the pop-up title, the system does not change the control rectangle to account for a length change. To resize the control rectangle, your program must dispose of the existing control and create a new one with `NewControl2`.

Table 37-1 lists the Menu Manager routines that work with pop-up menus. Refer to the call descriptions in either the *Toolbox Reference* or in this chapter for details on each call.

■ **Table 37-1** Menu Manager calls that work with pop-up menus

CalcMenuSize
`CheckMItem`
`CountMItems`
`DeleteMItem`
`DisableMItem`
`EnableMItem`
`GetMenuFlag`
`GetMenuTitle`
`GetMHandle`
`GetMItem`
`GetMItemFlag`
`GetMItemMark`
`GetMItemStyle`
`GetMTitleWidth`
`InsertMItem`
`SetMenuBar`
`SetMenuFlag`
`SetMenuID`
`SetMenuTitle`
`SetMenuTitle2`
`SetMItem`
`SetMItem2`
`SetMItemBlink`

```
SetMItemFlag
SetMItemMark
SetMItemName
SetMItemName2
SetMItemStyle
SetMTitleWidth
```

Each of the routines listed in Table 37-1 operate on the current menu bar. If your application defined its pop-ups using `NewControl2`, then it must set the pop-up to be the current menu, by issuing the `SetMenuBar` call and specifying the control handle for the pop-up as input.

If your application uses `PopUpMenuSelect`, rather than `NewControl2`, then it must insert the pop-up menu into the current menu bar by calling `InsertMenu`, issue the desired Menu Manager tool calls, then remove the pop-up menu from the menu bar by calling `DeleteMenu`. Your program passes the handle to the pop-up menu to each of these routines.

New Menu Manager data structures

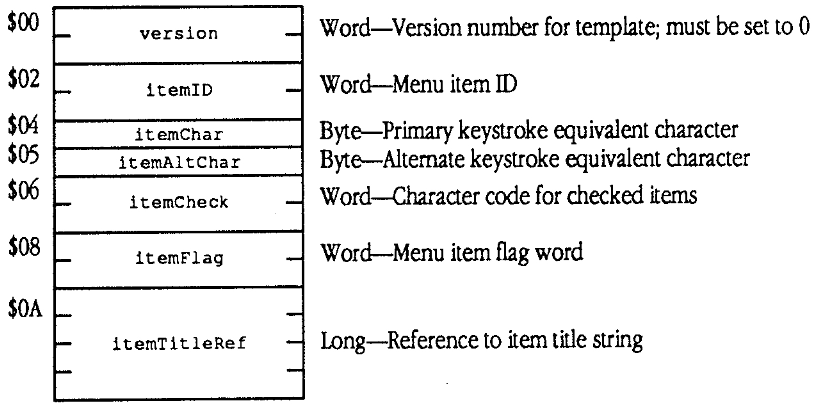
The new Menu Manager calls allow you to define menus using **templates**, analogous to the templates used by the `NewControl2` Control Manager tool call, which can then be stored in fixed memory, in allocated memory referenced by handle, or in resources. When using any of these new calls, your program must specify the input data according to these templates.

◆ *Note:* Any strings referenced in these data structure descriptions are Pascal strings. Note as well that all flag bit definitions are backward compatible. That is, no existing bits have been redefined. In addition, note that the `menuFlag` field is now defined as a word, rather than a byte. The byte following the old `menuFlag` byte, `menuRes`, was never used, and has been collapsed into `menuFlag`.

Menu item template

Figure 37-'6 shows the template that defines the characteristics of a menu item. Use it with new Menu Manager calls that require menu item templates.

■ **Figure 37-6** Menu item template


$00
version
Word—Version number for template; must be set to 0
$02
itemID
Word—Menu item ID
$04
itemChar
Byte—Primary keystroke equivalent character
$05
itemAltChar
Byte—Alternate keystroke equivalent character
$06
itemCheck
Word—Character code for checked items
$08
itemFlag
Word—Menu item flag word
$0A
itemTitleRef
Long—Reference to item title string

`version` Identifies the version of the menu item template. The Menu Manager uses this field to distinguish between different revisions of the menu item template. Must be set to 0.

`itemID` Unique identifier for the menu item. See Chapter 13, "Menu Manager," in the *Toolbox Reference* for information on valid values for `itemID`.

`itemChar, itemAltChar`
These fields define the keystroke equivalents for the menu item. The user can select the menu item by pressing the Command key along with the key corresponding to one of these fields. Typically, these fields contain the upper and lower case ASCII codes for a particular character. If you only have a single key equivalence, set both fields with that value.

`itemCheck` Defines the character to be displayed next to the item when it is checked.

`itemFlag` Bit flags controlling the display attributes of the menu item. Valid values for `itemFlag` are:

`titleRefType`	bits 14–15	Defines the type of reference in `itemTitleRef`: 00 - Reference is pointer 01 - Reference is handle 10 - Reference is resource ID 11 - Invalid value
Reserved	bit 13	Must be set to 0
`shadow`	bit 12	Indicates item shadowing: 0 - No shadow 1 - Shadow
`outline`	bit 11	Indicates item outlining 0 - Not outlined 1 - Outlined
Reserved	bits 8–10	Must be set to 0
`disabled`	bit 7	Enables or disables the menu item: 0 - Item enabled 1 - Item disabled
`divider`	bit 6	Controls drawing divider below item: 0 - No divider bar 1 - Divider bar
`XOR`	bit 5	Controls how highlighting is performed: 0 - Do not use XOR to highlight 1 - Use XOR to highlight item
Reserved	bits 3–4	Must be set to 0
`underline`	bit 2	Controls item underlining: 0 - Do not underline item 1 - Underline item

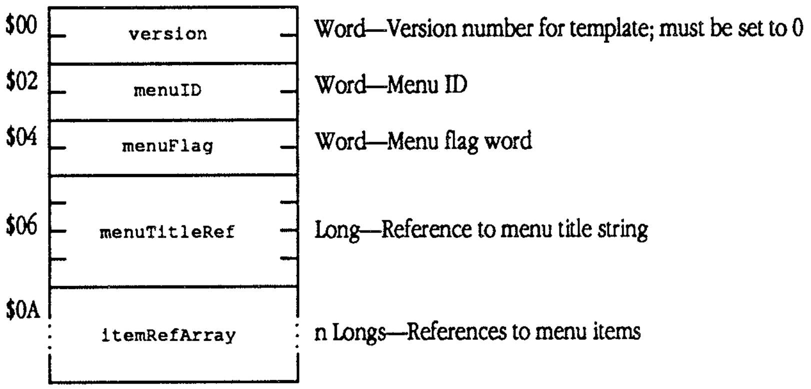
`italic`	bit 1	Indicates whether item is italicized 0 - Not italicized 1 - Italicized
`bold`	bit 0	Indicates whether item is drawn bold: 0 - Not bold 1 - Bold

`itemTitleRef` Reference to title string for menu item. The `titleRefType` bits in `itemFlag` indicate whether `itemTitleRef` contains a pointer, a handle, or a resource ID. If `itemTitleRef` is a pointer, then the title string must be a Pascal string. Otherwise, the Menu Manager can retrieve the string length from control information in the handle.

Menu template

Figure 37-7 shows the menu template, which defines the characteristics of a menu, including its menu item references. Use it with new Menu Manager calls that require menu templates.

- **Figure 37-7** Menu template


$00
version
Word—Version number for template; must be set to 0
$02
menuID
Word—Menu ID
$04
menuFlag
Word—Menu flag word
$06
menuTitleRef
Long—Reference to menu title string
$0A
itemRefArray
n Longs—References to menu items

`version` Identifies the version of the menu template. The Menu Manager uses this field to distinguish between different revisions of the template. Must be set to 0.

`menuID` Unique identifier for the menu. See Chapter 13, "Menu Manager," in the *Toolbox Reference* for information on valid values for `menuID`.

`menuFlag` Bit flags controlling the display and processing attributes of the menu. Valid values for `menuFlag` are:

`titleRefType` bits 14–15 Defines the type of reference in `menuTitleRef`:
00 - Reference is pointer
01 - Reference is handle
10 - Reference is resource ID
11 - Invalid value

`itemRefType`	bits 12–13	Defines the type of reference in each entry of `itemRefArray` (all array entries must be of the same type): 00 - References are pointers 01 - References are handles 10 - References are resource IDs 11 - Invalid value
Reserved	bits 9–11	Must be set to 0
`alwaysCallmChoose`	bit 8	Causes the Menu Manager to call a custom menu defProc mChoose routine even when the mouse is not in the menu rectangle (supports tear-off menus): 0 - Do not always call mChoose routine 1 - Always call mChoose routine
`disabled`	bit 7	Enables or disables the menu: 0 - Menu enabled 1 - Menu disabled
Reserved	bit 6	Must be set to 0
`XOR`	bit 5	Controls how selection highlighting is performed: 0 - Do not use XOR to highlight 1 - Use XOR to highlight item
`custom`	bit 4	Indicates whether custom or standard menu: 0 - Standard menu 1 - Custom menu
`allowCache`	bit 3	Controls menu caching: 0 - Do not cache menu 1 - Menu caching allowed
Reserved	bits 0–2	Must be set to 0

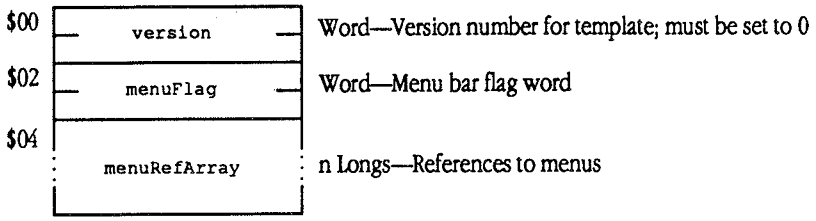
`menuTitleRef` Reference to title string for menu. The `titleRefType` bits in `menuFlag` indicate whether `menuTitleRef` contains a pointer, a handle, or a resource ID. If `menuTitleRef` is a pointer, then the title string must be a Pascal string. Otherwise, the Menu Manager can retrieve the string length from control information in the handle.

`itemRefArray` Array of references to the menu items for the menu. The `itemRefType` bits in `menuFlag` indicate whether the entries in the array are pointers, handles, or resource IDs. Note that all array entries must contain the same reference type. The last entry in the array must be set to $00000000.

Menu bar template

Figure 37-8 shows the menu bar template, which defines the characteristics of a menu bar, including its menu references. Use it with new Menu Manager calls that require menu bar templates.

■ **Figure 37-8** Menu bar template


$00
version
Word—Version number for template; must be set to 0
$02
menuFlag
Word—Menu bar flag word
$04
menuRefArray
n Longs—References to menus

`version` Identifies the version of the menu bar template. The Menu Manager uses this field to distinguish between different revisions of the template. Must be set to 0.

`menuBarFlag` Bit flags controlling the display and processing attributes of the menu bar. Valid values for `menuBarFlag` are:

`menuRefType` bits 14–15 Defines the type of reference in each entry of `menuRefArray` (all array entries must be of the same type):
00 - References are pointers
01 - References are handles
10 - References are resource IDs
11 - Invalid value

Reserved bits 0–13 Must be set to 0

`menuRefArray` Array of references to the menus for the menu bar. The `menuRefType` bits in `menuBarFlag` indicate whether the entries in the array are pointers, handles, or resource IDs. Note that all array entries must contain the same reference type. The last entry in the array must be set to $00000000.

New Menu Manager calls

The following sections discuss the various new Menu Manager tool calls in alphabetical order by call name.

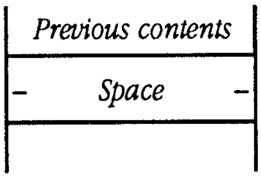
`GetPopUpDefProc` $3B0F

Returns a pointer to the control definition procedure for pop-up menus. Your application should not issue this call.

The system issues this call during Control Manager start up processing in order to obtain the address of the pop-up menu definition procedure.

Parameters

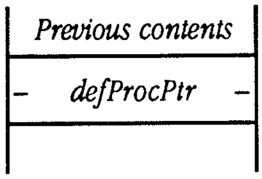
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to control procedure

<—SP

Errors None

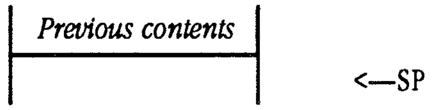
C `extern pascal Pointer GetPopUpDefProc();`

`HideMenuBar` $450F

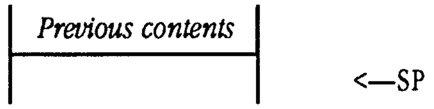
Hides the system menu bar, by adding the menu bar to the desktop region. This call sets the invisible flag for the menu bar, resets scan lines 2 through 9 (which had been changed to correctly display the colors of the Apple logo), and refreshes the desktop. The system ignores all subsequent calls to `DrawMenuBar` or `FlashMenuBar`, since the menu bar is invisible. Use the `ShowMenuBar` call to make the menu bar visible again.

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal void HideMenuBar();
```

`InsertMItem2` $3F0F

Inserts a menu item into a menu after a specified menu item or at the top of the menu. This call accepts a menu item template for its input specification.

Parameters

Stack before call

Previous contents	
refDescriptor	Word—Defines type of reference in *menuItemTRef*
– *menuItemTRef* –	Long—Reference to menu item template
insertAfter	Word—ID of item after which to insert this item
menuNum	Word—ID of menu into which to insert item
	<—SP

Stack after call

Previous contents	
	<—SP

Errors None

C

```
extern pascal void InsertMItem2(refDescriptor,
              menuItemTRef, insertAfter, menuNum);

Word        refDescriptor, insertAfter, menuNum;
Long        menuItemTRef;
```

refDescriptor Indicates the type of reference stored in *menuTRef.* Valid values are:

0	Reference is a pointer
1	Reference is a handle
2	Reference is a resource ID

insertAfter Specifies ID of item after which the new item is to be inserted. In order to insert the new item at the top of the menu, set this field to 0. To insert at the end, set this field to $FFFF.

NewMenu2 $3E0F

Allocates space for a menu list and its items. This call accepts a menu template for its input specification.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
refDescriptor	Word—Defines type of reference in *menuTRef*
– *menuTRef* –	Long—Reference to menu template
	<—SP

Stack after call

Previous contents	
– *menuBarHandle* –	Long—Handle for new menu
	<—SP

Errors None

C

```
extern pascal Long NewMenu2(refDescriptor,
                    menuTRef);

Word        refDescriptor;
Long        menuTRef;
```

refDescriptor Indicates the type of reference stored in *menuTRef*. Valid values are:

0 Reference is a pointer
1 Reference is a handle
2 Reference is a resource ID

`NewMenuBar2` $430F

Creates a default menu bar with no menus. This call accepts a menu bar template for its input specification.

The upper-left corner of the default menu bar matches the port, and is as wide as the screen. The bar is 13 pixels high.

Note that passing a NIL value for the windowPtr parameter creates a menu bar that is not inside a window, but does not automatically replace the current menu bar. In order to create a new system menu bar and make it current, you must issue the following tool calls:

```
NewMenuBar2()
SetSysBar                 /* use menuBarHandle from NewMenuBar2 */
SetMenuBar(NIL)
```

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
refDescriptor	Word—Defines type of reference in *menuBarTRef*
– *menuBarTRef* –	Long—Reference to menu bar template
– *windowPtr* –	Long—Pointer to port for window; NIL for system menu bar
	<—SP

Stack after call

Previous contents	
– *menuBarHandle*	Long—Handle for-new menu bar
	<—SP

Errors None

C

```
extern pascal Long NewMenuBar2(refDescriptor,
                    menuBarTRef, windowPtr);
```

```
Word       refDescriptor;
Long       menuBarTRef;
Pointer    windowPtr;
```

refDescriptor Indicates the type of reference stored in *menuBarTRef.* Valid values are:

0	Reference is a pointer
1	Reference is a handle
2	Reference is a resource ID

`PopUpMenuSelect` $3C0F

Draws highlighted titles and handles user interaction when the user clicks on a pop-up menu.

You specify the pop-up with the handle returned by `NewMenu` or `NewMenu2`.

◆ *Note:* The system draws the pop-up menu into the port that is active at the time you issue the `PopUpMenuSelect` call. The menu is constrained by the intersection of the port rectangle, the visible region, and the clip region. By altering any of these you can change the constraints on the menu.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result (Item ID)
selection	Word—Item ID of current menu selection
currentLeft	Word—Global coordinate value of left edge of pop-up menu
currentTop	Word—Global coordinate value of top of current selection
flag	Word—Flag word for call
– *menuHandle* –	Long—Menu handle
	<—SP

Stack after call

Previous contents	
itemID	Word—Item ID of new selection (0 if none)
	<—SP

Errors None

C

```
extern pascal Word PopUpMenuSelect(selection,
          currentLeft, currentTop, flag,
          menuHandle);

Word      selection, currentLeft, currentTop, flag;
Long      menuHandle;
```

selection Defines the current selection in the menu. Set to 0 if no item is currently selected. The initial value is the default value for the menu, and is displayed in the pop-up rectangle of "unpopped" menus. You specify an item by its ID, that is, its relative position within the array of items for the menu (see Chapter 37, "Menu Manager Update," for information on the layout and content of the pop-up menu template). If you pass an invalid item ID then no item is displayed in the pop-up rectangle.

currentLeft, currentTop
Define the left edge of the pop-up and the top of the current selection, global coordinates.

flag Flag word for the tool call. Bits are defined as follows:

Reserved	bits 7–15	Must be set to 0
`type2`	bit 6	Indicates whether pop-up is type 1 or type2: 0 - Type 1 menu (no white space added) 1 - Type 2 menu (white space added)
Reserved	bits 0–5	Must be set to 0

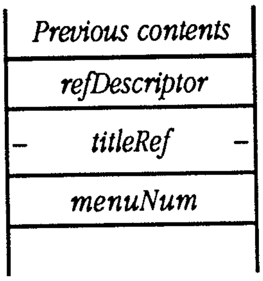
menuHandle The handle of the pop-up menu. The Menu Manager returned this value to your application from `NewMenu` or `NewMenu2`.

SetMenuTitle2 $400F

Specifies the title for a menu. This call accepts a reference to the title string that can be a pointer, handle, or resource ID.

Parameters

Stack before call



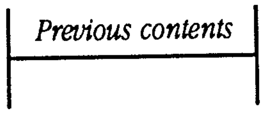
Word—Defines type of reference in *titleRef*

Long—Reference to title string for menu

Word—ID of menu to receive title

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void SetMenuTitle2(refDescriptor,
                    titleRef, menuNum);

Word        refDescriptor, menuNum;
Long        menuItemTRef;
```

refDescriptor Indicates the type of reference stored in *titleRef.* Valid values are:

0	Reference is a pointer
1	Reference is a handle
2	Reference is a resource ID

`SetMItem2` $410F

Specifies the name for a menu item. This call accepts a menu item template for its input specification.

Parameters

Stack before call

Previous contents	
refDescriptor	Word—Defines type of reference in *menuItemTRef*
– *menuItemTRef* –	Long—Reference to menu item template
menuItem	Word—ID of item to be changed
	<—SP

Stack after call

Previous contents	
	<—SP

Errors None

C

```
extern pascal void SetMItem2(refDescriptor,
                    menuItemTRef, menuItem);

Word        refDescriptor, menuItem;
Long        menuItemTRef;
```

refDescriptor Indicates the type of reference stored in *menuItemTRef*. Valid values are:

0	Reference is a pointer
1	Reference is a handle
2	Reference is a resource ID

menuItem Specifies the menu item to be changed. Note that you can change the item ID by specifying a different item number in the menu item template. The Menu Manager will apply the item ID from the template to the item to be changed.

`SetMItemName2` $420F

Specifies the name of a menu item . This call accepts a reference to a title string that can be a pointer, handle, or resource ID.

Parameters

Stack before call

Previous contents	
refDescriptor	Word—Defines type of reference in *titleRef*
– *titleRef* –	Long—Reference to menu item title
menuItem	Word—ID of item to be changed
	<—SP

Stack after call

Previous contents	
	<—SP

Errors None

C

```
extern pascal void SetMItemName2(refDescriptor,
          titleRef, menuItem);

Word      refDescriptor, menuNum;
Long      titleRef;
```

refDescriptor Indicates the type of reference stored in *titleRef.* Valid values are:

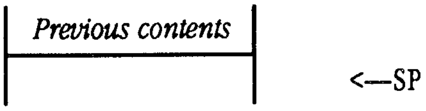
0	Reference is a pointer
1	Reference is a handle
2	Reference is a resource ID

`ShowMenuBar` $460F

Reveals the system menu bar, by subtracting the menu bar from the desktop region. This call also resets the `invisible` flag for the menu bar, resets scan lines 2 through 9 (to correctly display the colors of the Apple logo), and draws the menu. Use the `HideMenuBar` call to make the menu bar invisible.

Parameters

Stack before call



Stack after call

Previous contents
<—SP

Errors None

C `extern pascal void ShowMenuBar();`

Chapter 38 **MIDI Tool Set**

This chapter documents the MIDI Tool Set. This is a new tool set; it was not documented in the *Apple IIGS Toolbox Reference.*

About the MIDI Tool Set

The Apple IIGS MIDI Tool Set provides a software interface between the Apple IIGS and external synthesizers and other musical equipment that accepts the **Musical Instrument Digital Interface (MIDI)** protocol. The MIDI Tool Set has the following key attributes:

- **Hardware independence**

  The MIDI Tool Set is hardware-independent. It uses a separately loaded device driver to communicate with the hardware interface that connects the Apple IIGS to an external MIDI device. This driver-based design frees applications from referencing the specifics of the MIDI hardware interface. Applications that use the MIDI Tool Set can therefore run on Apple IIGS systems with different MIDI interfaces.

- **Interrupt-driven operation**

  The MIDI Tool Set is interrupt-driven and can transfer MIDI data in the background while other tasks take place in the foreground. For example, it is possible to write an application that enables a user to edit MIDI data while simultaneously playing a sequence. MIDI applications that use the tool set need not provide interrupt handlers since they are provided by the MIDI Tool Set.

- **Accurate clock**

  The MIDI Tool Set provides a high-speed, high-resolution clock. If an application needs precise timing, it can use the MIDI Tool Set clock to provide time-stamps accurate to within 76 microseconds. The clock uses one of the **Digital Oscillator Chip (DOC)** generators and the first 256 bytes of DOC RAM. When the clock is not in use, the MIDI Tool Set releases the DOC generator and RAM. See Chapter 47, "Sound Tool Set Update," for more information about the Digital Oscillator Chip.

- **Fast response**

  The tool set automatically polls incoming MIDI data, and receives the data without loss at speeds up to one byte per 320 microseconds—as long as interrupts are never disabled for more than 270 microseconds. If your application must disable interrupts for longer than this interval, you can use the `MidiInputPoll` vector to retrieve incoming data explicitly.

- **Multiple formats**

  The tool set supports two input and output formats. When the application retrieves MIDI data in raw mode, it receives the data bytes exactly as they appear in the input stream, but with length and time-stamp data added. In packet mode, the MIDI Tool Set expects to receive MIDI data packets, but will perform some additional cleanup to make those packets complete.

- **Error checks**

  The MIDI Tool Set provides error-checking and reports a variety of error conditions, including reception of MIDI packets with an incorrect number of data bytes.

- **Real-time and background commands**

  The MIDI Tool Set can report real-time commands to an application immediately. This feature enables the application to process real-time commands as they occur, for interactive control of musical instruments.

- **Intelligent NoteOff commands**

  The tool set's NoteOff commands can turn off all notes that are playing or only those it has turned on. They can do this on all channels or only on specified channels.

- **Variable clock frequency**

  You can change the time base for MIDI timestamps, thereby varying the tempo of played data (see the description of the `miSetFreq` function of the `MidiClock` tool call later in this chapter for more information).

- **User definable service routines**

  You can enhance the functionality provided by the MIDI Tool Set by providing your own service routines. The MIDI Tool Set calls these routines under a variety of defined circumstances. See "MIDI Tool Set service routines" later in this chapter for more information.

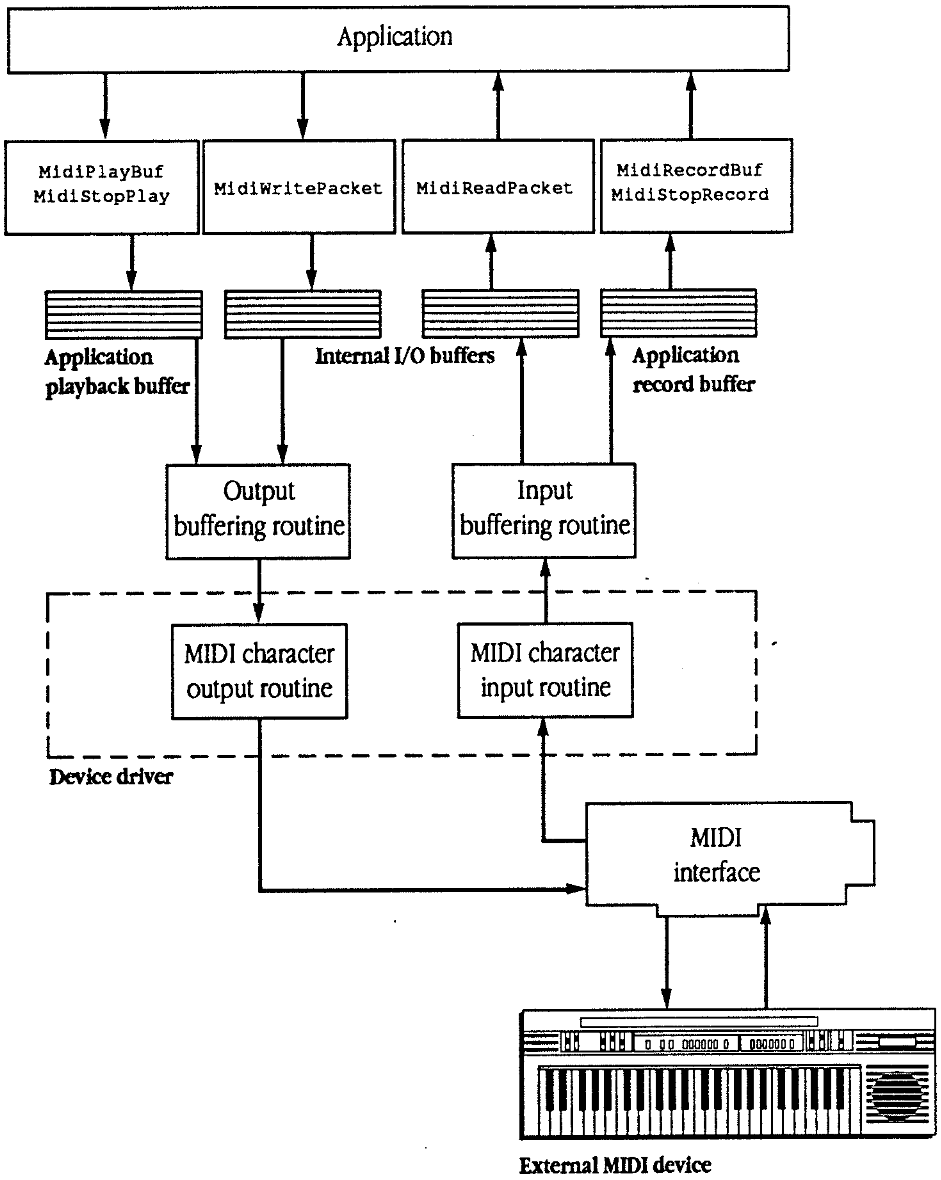
◆ *Note:* The Note Synthesizer, the Note Sequencer, and the MIDI Tool Set refer to the software tools provided with the Apple IIGS, not to any separate instrument or device. The MIDI tools are software tools for use in controlling external instruments, which may be connected through a MIDI interface device.

Using the MIDI Tool Set

This section describes the basic steps involved in using the MIDI Tool Set to interact with external musical instruments. Following the initial overview discussion are several code examples, demonstrating techniques for performing many key MIDI Tools functions.

Figure 38-1 illustrates some of the relationships between a typical MIDI application, the MIDI Tool Set, MIDI device drivers, and a MIDI interface card.

- **Figure 38-1** MIDI application environment



Before using the MIDI Tool Set, you must install the tool set and its associated drivers using the Installer utility.

To use the MIDI Tool Set, you must first start it up with the `MidiStartUp` call. Then you must load a MIDI device driver by using the `MidiDevice` call. The tool set loads the driver separately so that its operation is independent of the particular MIDI interface that connects the Apple IIGS to the external MIDI instrument.

MIDI device drivers are normally found in the */SYSTEM/DRIVERS directory, and end with the suffix .MIDI. Apple currently supplies the APPLE.MIDI and CARD6850.MIDI drivers; the first driver supports the Apple MIDI Interface, and the second supports plug-in 6850-based **Asynchronous Communications Interface Adapter (ACIA)** cards.

After the application loads the MIDI device driver, it must make the `MidiControl` call to allocate input and output buffers for `MidiReadPacket` and `MidiWritePacket` calls. Note that if the application never calls `MidiReadPacket`, it need not allocate an input buffer, and if it never calls `MidiWritePacket`, it need not allocate an output buffer.

The MIDI Tool Set is now ready to send or receive MIDI data. However, the application must explicitly start the MIDI input and output processes, using the appropriate options of the `MidiControl` tool call.

The application can start or stop MIDI data transfer at any time. Once started, the input and output processes continue without intervention until stopped by the application. They run in the background so that other processes, such as interaction with the user, can run unimpeded in the foreground. The tool set enables the programmer to switch the processes on or off at any time because MIDI data transfer incurs considerable processor overhead, and a programmer might want to disable it under some circumstances to improve the application's performance on other tasks.

The MIDI input process fills the MIDI Tool Set's input buffer with data packets as they arrive. The application must periodically retrieve the data from the buffer by making calls to `MidiReadPacket`. Similarly, the MIDI output process transmits the data placed in the tool set's output buffer by application with calls to `MidiWritePacket`. The Apple IIGS can simultaneously send and receive MIDI data packets.

When you use the `MidiClock` call to start the MIDI Tool Set's clock, the tool set begins stamping each data packet with a time value it retrieves from its clock process. This clock is actually a DOC generator that the MIDI Tool Set allocates with the Note Synthesizer `AllocGen` call. Start the MIDI clock before starting the input process, because the `MidiClock` function disables interrupts long enough to interfere with correct reception of MIDI data.

The clock is very fast; a tick occurs every 76 microseconds at the default settings. MIDI data packets receive a time-stamp consisting of the value of the clock when they are received. `MidiWritePacket` receives a packet with a time-stamp attached and writes it to the output buffer, and the MIDI Tool Set transfers the packet only when the current value of the clock is greater than the output data's time-stamp.

If the clock is stopped, MIDI input data receive time-stamps equal to the value of the stopped clock, and only MIDI data with time-stamps less than the value of the stopped clock can be sent.

If you want to read and write MIDI packets in real time, in response to user events, you do not need the MIDI clock.

You can start or stop the MIDI clock or the input and output processes at any time, so that you can budget processor resources intelligently. The input, output, and clock processes consume a great deal of processor time, and so limit the processing power available to tasks that execute during their operation.

Tool dependencies

The MIDI Tool Set uses Note Synthesizer calls to allocate a DOC generator for its clock. If your application does not use the MIDI Tool Set clock, you need not start up the Note Synthesizer to use the MIDI Tool Set. If your application is not using the MIDI Tool Set clock or `MidiInputPoll`, then it can start up and shut down the Note Synthesizer as needed, but the Note Synthesizer must be started up if you use the MIDI clock or the `MidiInputPoll` vector.

The Sound Tools must be started to use the MIDI tools.

Refer to Chapter 51, "Tool Locator Update," for information about the specific version requirements the MIDI Tool Set has for other tool sets.

MIDI packet format

MIDI data sent and received using the MIDI Tool Set must always be formatted into valid MIDI Tool Set packets. The tool set handles this for incoming data; your application must format outgoing data according to the packet layout described in this section.

The first 2 bytes of a packet contain a byte count of the MIDI data in the packet, plus the 4-byte time-stamp. The next 4 bytes are the time-stamp, and are equal to the value of the MIDI clock at the time the packet was received. The remaining bytes are the actual MIDI data:

$00	`length`	Word—Packet length (excluding `length`)
$02	`timestamp`	4 Bytes—MIDI time-stamp
$06	`MIDIData`	Array—MIDI data (variable length)

A NoteOn command might look like this (in hex notation):

07 00 24 63 03 00 90 40 5C

The first 2 bytes are the length in bytes of the MIDI data packet plus the 4-byte time-stamp. In this case the MIDI packet is 3 bytes, so the length value is 7. The next 4 bytes contain the time-stamp, and the MIDI data follow.

The result of a `MidiReadPacket` call on this packet is 9—the 7 bytes counted in the length word plus the 2 bytes of the length word itself.

If the current input mode is MIDI packet mode, the first byte of the MIDI data is always a MIDI status byte. If a received MIDI packet does not contain a valid status byte, the MIDI Tool Set inserts the current status at the beginning of the packet. `MidiReadPacket` never returns real-time commands in packet mode; they are always passed to the real-time command routine installed by `MidiControl`. See "MIDI Tool Set service routines" later in this chapter for more information.

In raw mode the MIDI data are returned to the application just as they are received from the MIDI interface. The MIDI protocol allows MIDI devices to omit the status byte unless it has changed from its last value. The status byte may appear anywhere in the stream because it may be received only when it changes. MIDI devices may also omit the $F7 value at the end of a MIDI System Exclusive command; the $F7 value always appears at the end of a System Exclusive command in packet mode, but not necessarily in raw mode.

In raw mode, the maximum number of MIDI data bytes that `MidiReadPacket` passes to the application is 4. Therefore, the longest packet it can pass is 10 bytes in length—2 length bytes, 4 time-stamp bytes, and 4 MIDI data bytes. In packet mode, System Exclusive packets may be of any length.

`MidiReadPacket` also returns real-time commands in raw mode unless a real-time vector is installed See "`MidiControl`" later in this chapter for more information.

MIDI Tool Set service routines

Your program can contain service routines that the MIDI Tool Set will invoke under certain defined circumstances. By providing these service routines, you can tailor the functionality of the MIDI Tool Set to fit your particular needs. The MIDI Tool Set calls these routines under the following circumstances

Real-time command routine
Called when the MIDI Tool Set receives a MIDI real-time command. You set the vector to this routine with the `miSetRTVec` function of the `MidiControl` tool call.

Real-time error routine
Called when the MIDI Tool Set encounters an error during real-time processing. You set the vector to this routine with the `miSetErrVec` function of the `MidiControl` tool call.

Input data routine
Called by the MIDI Tool Set to handle MIDI data received during processing of an `miStartInput` function request. You set the vector to this routine when you issue the `miStartInput` function of the `MidiControl` tool call.

Output data routine
Called by the MIDI Tool Set to obtain data to send during processing of an `miStartOutput` function request. You set the vector to this routine when you issue the `miStartOutput` function of the `MidiControl` tool call.

The following sections discuss each of these service routines in more detail.

Real-time command routine

When the MIDI Tool Set receives MIDI real-time commands, it calls this service routine. The service routine must not enable interrupts, and if it runs for longer than 300 microseconds, it must call the MIDI polling vector at least every 270 microseconds. The only MIDI calls that the service routine should make are `MidiReadPacket` and `MidiWritePacket`.

Real-time MIDI data are passed to the service routine in the low-order byte of a word on the stack above the `RTL` address. This word must remain on the stack. When the service routine is called, the data bank register is set to the value it had when `MidiStartUp` was called, but the direct-page register points to one of the MIDI Tool Set's direct pages and must be preserved.

You set the vector to this routine with the `miSetRTVec` function of the `MidiControl` tool call.

Parameters

Stack before call

Previous contents	
MIDIData	Word—Low-order byte contains MIDI real-time data
– *returnAddress* –	3 Bytes—`RTL` address
	<—SP

Stack after call

Previous contents	
MIDIData	Word—Low-order byte contains MIDI real-time data
– *returnAddress* –	3 Bytes—`RTL` address
	<—SP

Real-time error routine

The MIDI Tool Set calls this routine in the event of a MIDI real-time error. This service routine must not enable interrupts. If it executes for longer than 300 microseconds, it must call the MIDI polling vector at least every 270 microseconds. It can call `MidiWritePacket` and `MidiReadPacket`, but no other MIDI tool calls.

The error is passed to the service routine in a word on the stack above the RTL address. This word must remain on the stack. When the service routine is called, the data bank register is set to the value it had when `MidiStartUp` was called, but the direct page register points to one of the MIDI Tool Set's direct pages and must be preserved. When the MIDI Tool Set invokes this routine, there is very little space left on the stack.

You set the vector to this routine with the `miSetErrVec` function of the `MidiControl` tool call.

The service routine may receive the following error codes:

$200A	`miClockErr`	MIDI clock wrapped to 0.
$2084	`miDevNoConnect`	No connection to MIDI interface.

Parameters

Stack before call

Previous contents	
MIDIError	Word—Error code
– *returnAddress* –	3 Bytes—RTL address
	<—SP

Stack after call

Previous contents	
MIDIError	Word—Error code
– *returnAddress* –	3 Bytes—RTL address
	<—SP

Input data routine

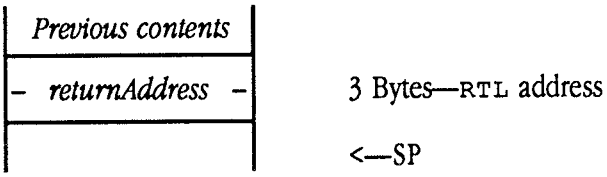
The MIDI Tool Set calls this routine during processing of the `miStartInput` function of the `MidiControl` tool call when the first packet is available in a previously empty input buffer. The service routine must not enable interrupts, and if it runs for longer than 300 microseconds, it must call `MidiInputPoll` at least every 270 microseconds. The only MIDI calls that the service routine should make are `MidiReadPacket` and `MidiWritePacket`.

When the service routine is called, the data bank register is set to the value it had when `MidiStartUp` was called, but the direct page register points to one of the MIDI Tool Set's direct pages and must be preserved. The system will call the service routine immediately if a complete MIDI packet is available in the input buffer when the `miStartInput` function of the `MidiControl` tool call is made.

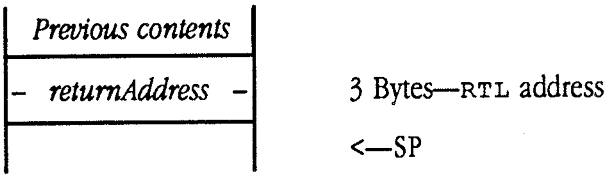
You set the vector to this routine when you issue the `miStartInput` function of the `MidiControl` tool call.

Parameters

Stack before call



Stack after call



Output data routine

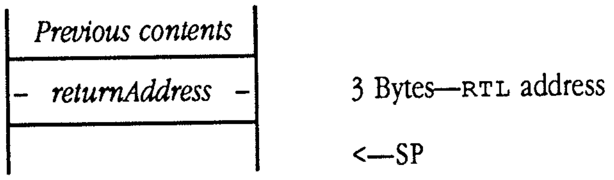
The MIDI Tool Set calls this routine during processing of the `miStartOutput` function of the `MidiControl` tool call when the output buffer becomes completely empty. The service routine must not enable interrupts, and if it runs for longer than 300 microseconds, it must call the MIDI polling vector at least every 270 microseconds. The only MIDI calls that the service routine should make are `MidiReadPacket` and `MidiWritePacket`.

When the service routine is called, the data bank register is set to the value it had when `MidiStartUp` was called, but the direct page register points to one of the MIDI Tool Set's direct pages and must be preserved.

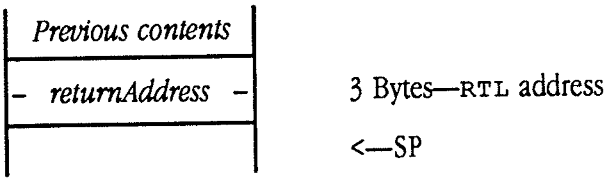
You set the vector to this routine when you issue the `miStartOutput` function of the `MidiControl` tool call.

Parameters

Stack before call



Stack after call



Starting up the MIDI Tool Set

The `MidiStartUp` call takes as arguments a word containing the Memory Manager ID number of the application that is starting up the tools, and a word containing the address of a three-page memory block in bank zero. The three-page block is used as the MIDI Tool Set's direct-page area, and it must be aligned on a page boundary.

```
/*
 *  StartupTools()
 *
 *  Starts up the MIDI Tool Set and all of the tools it
 *    requires.  For readability, this subroutine is presented
 *    without the error-checking that would normally be performed
 *    after each tool is started (call?).
 */

/* direct page use */
#define DPForSound       0x0000             /* needs 1 */
#define DPForMidi        0x0100             /* needs 3 */
#define DPForEventMgr    0x0400             /* needs 1 */
#define TotalDP          0x0500             /* total direct page use */

static word AppID;                          /* Apps Memory Manager ID */

void
StartupTools()
{
      static struct {
            word NumberOfTools;
            word Table[5*2];
      } ToolTable = {
            5,                              /* number of tools in list */
            1,  0x0101,                     /* Tool Locator */
            2,  0x0101,                     /* Memory Manager */
            8,  miSTVer,                    /* Sound Tools */
            25, miNSVer,                    /* Note Synthesizer */
            miToolNum, 0x0000               /* Midi Tool Set */
      };
      MiDriverInfo DriverInfo;              /* device driver info */
      MiBufInfo InBufInfo,OutBufInfo;       /* I/O buffer information */
```

```
handle ZeroPageHandle;
ptr ZeroPagePtr;


TLStartUp();                        /* Tool Locator startup */
AppID = MMStartUp();                /* Memory Manager startup */

/* allocate direct pages for tools */
ZeroPageHandle = NewHandle((long) TotalDP,
                        (word) AppID,
                        (word) attrBank | attrPage | attrFixed
                              | attrLocked,
                        (long) 0);
ZeroPagePtr = *ZeroPageHandle;

EMStartUp((word)(ZeroPagePtr + DPForEventMgr), (word) 0,
            (word) 0,
            (word) 640,
            (word) 0,
            (word) 200,
            (word) AppID );

LoadTools(&ToolTable);              /* load RAM-based tools */

SoundStartUp((word)(ZeroPagePtr + DPForSound));
NSStartUp(0, 0L);
MidiStartUp(AppID, (word)(ZeroPagePtr + DPForMidi));
                                    /* load device driver */
DriverInfo.slot = 2;                /* use the modem port */
DriverInfo.external = 0;            /* internal slot */
strcpy(DriverInfo.file, "\p*/system/drivers/apple.midi");
MidiDevice(miLoadDrvr, &DriverInfo);
```

```
      /* allocate input and output buffers */
      InBufInfo.bufSize = 0;              /* default size */
      InBufInfo.address = 0;              /* MIDI Tool Set will
                                             allocate the buffer and
                                             set its actual address */
      MidiControl(miSetInBuf, &InBufInfo);
      OutBufInfo.bufSize = 0;             /* default size */
      OutBufInfo.address = 0;             /* MIDI Tool Set will
                                             allocate the buffer and
                                             set its actual address */
      MidiControl(miSetOutBuf, &OutBufInfo);
}     /* end of StartupTools() */
```

Reading time-stamped MIDI data

This example shows a simple method of recording time-stamped MIDI data as it is received. The example records incoming data until any key is depressed or until the MIDI Tool Set's internal data buffer is full, whichever comes first. The routine's data buffer should not be confused with the MIDI Tool Set's input buffer, which you allocate for MIDI data by using the `MidiControl` call.

```
/*
 *   RecordMIDI()
 *
 *   Record incoming MIDI data with time stamps into the
 *   global buffer "AppMIDIBuffer" until the buffer is
 *   full or the user presses the mouse button.
 */

#define BufSize         (20 * 1024)
char SeqBuffer[BufSize];
int BufIndex = 0;
```

```
void
RecordMIDI()
{
      int PacketSize;                     /* size of packet read */


      MidiControl(miFlushInput, 0L);      /* discard contents
                                             of input buffer */
      MidiClock(miSetFreq,0L);            /* set clock to
                                             default frequency */
      MidiClock(miSetClock, 0L);          /* clear the clock */
      MidiClock(miStartClock, 0L);        /* start the clock */
      MidiControl(miSetInMode,
            (long)miPacketMode);          /* set MIDI input mode */
      MidiControl(miStartInput, 0L);      /* start MIDI input */

      BufIndex = 0;

      while ( Button(0) == 0 )            /* until presses mouse */
      {
            PacketSize = MidiReadPacket(SeqBuffer+BufIndex,
                              BufSize-BufIndex);
            if (_toolErr)
            {
                  if (_toolErr == miArrayErr)
                  {
                        break;            /* our buffer is full */
                  }
                  else
                  {
                        printf("MIDI error $%4.4X\n",_toolErr);
                  }
            }
            else
            {
                  BufIndex += PacketSize;
            }
      }
```

```
        /* stop recording */
        MidiControl(miStopInput, 0L);        /* stop MIDI input */
        MidiClock(miStopClock, 0L);          /* stop the clock */

        /* show user recording statistics */

        printf("Bytes recorded: %d\n",BufIndex);
        printf("Maximum bytes buffered: %ld\n",
              MidiInfo(miMaxInChars));
}       /* end of RecordMIDI() */
```

This example is a simple subroutine that continuously plays previously recorded time-stamped MIDI data until the user presses any key.

```
/*
 *  PlayMIDI()
 *
 *  This routine repeatedly plays the MIDI data that was
 *  previously recorded and stored into the global buffer
 *  "SeqBuffer" until the user presses the mouse button.
 */

void
PlayMIDI()
{
        long FirstTime;
        int PlayIndex;

        if (BufIndex == 0)
        {
              printf("You must record or load MIDI data first\n");
              return;
        }

        /* find the first time stamp in the sequence and subtract
        a little */
        FirstTime = *((long *) (SeqBuffer+2));
        if (FirstTime > 0x200)
              FirstTime -= 0x200;
        else
              FirstTime = 0;
        MidiControl(miFlushOutput,(long)(0xFFFF << 16));
                                            /* empty output buffer */
        MidiClock(miSetClock,FirstTime);    /* set clock before
                                               first time stamp*/
        MidiClock(miStartClock,0L);         /* start clock */
        MidiControl(miSetOutMode, (long)miPacketMode);
                                            /* set output mode */
        MidiControl(miStartOutput,0L);      /* start output */
        PlayIndex = 0;

        /* Repeatedly play song */
        while ( Button(0) == 0 )            /* Until presses mouse */
        {
              PlayIndex += MidiWritePacket(SeqBuffer + PlayIndex);
                                            /* write next packet */
              if (PlayIndex == BufIndex)    /* Time to repeat? */
              {
                    while ( Button(0) == 0 && MidiInfo(miOutputChars))
                          ;                 /* wait for the song to end */

                    if (Button(0) || !LoopPlayback)
                          break;
                    MidiClock(miSetClock,FirstTime);
                                            /* restart clock */
                    PlayIndex = 0;
              }
        }

        MidiControl(miFlushOutput,0x10L);
                                            /* flush output buffer
                                               & turn all notes off */
        MidiClock(miStopClock,0L);          /* stop the clock */
        MidiControl(miStopOutput,0L);       /* stop output */
}       /* end of PlayMIDI() */
```

Fast access to MIDI Tool Set routines

Because of the tight timing requirements of MIDI processing, there are many time-critical situations in which the overhead of a tool call can be problematic. When you need to save as much time as possible, you may want to call MIDI Tool Set routines directly and avoid the time needed to make a tool call. The following example demonstrates how to do this in 65816 assembly language. This example can save approximately 85 microseconds per call. This time saving can be very helpful in an application that makes numerous calls to `MidiReadPacket` and `MidiWritePacket`.

```
;
; look up the address of MidiWritePacket (as an example)
;
               pushlong #0          ; space for result
               pushword #0          ; system tool
               pushword #$0E20      ; tool and function number
               _GetFuncPtr
               pla
               sta MidiWriteAddr    ; save the address
               pla
               sta MidiWriteAddr+2
                     .
                     .
                     .
;
```

```
; do this instead of _MidiWritePacket
;
                jsl MidiWriteGlue
                      .
                      .
                      .
;
; IMPORTANT NOTE: The variable "MidiDP2" must contain the
; address of the second page of bank zero memory allocated for
; the MIDI Tool Set's direct page. If MidiStartUp is given
; a starting address of X, then MidiDP2 = X + $100.
;
MidiWriteGlue   jsl MidiWriteGlue1    ; push an extra RTL
;                                       address
                 rtl
 MidiWriteGlue1 lda MidiWriteAddr+1    ; simulate a Tool call
                pha                    ; to MidiWritePacket
                phb
                lda MidiWriteAddr
                sta 1,s
                lda MidiDP2            ; the A register must
;                                        contain the address of
;                                        the MIDI Tool
;                                        Set's direct page address
GlueReturn      rtl
MidiWriteAddr   ds 4
```

MIDI application considerations

This section contains advice on a number of topics, and is intended to help you create more satisfying MIDI applications.

MIDI and AppleTalk

The MIDI Tool Set is not designed to operate with AppleTalk® enabled. The Apple IIGS is not fast enough to process both AppleTalk interrupts and MIDI interrupts simultaneously. If an application that uses the MIDI Tool Set runs with AppleTalk enabled, you should expect occasional MIDI input errors and output delays. For most programs, even one MIDI error is difficult to handle, so you should probably recommend that applications that use the MIDI Tool Set not be used with AppleTalk enabled.

Disabling interrupts

Several tool calls that disable interrupts can cause loss of MIDI data. These include calls that access the disk drives, Event Manager calls, and Dialog Manager calls.

The rate of MIDI data transfer leaves little margin for error in the MIDI Tool Set's operation. The rate at which the tool set must retrieve MIDI data places great demands on the system's computational resources. If possible, an application should avoid disabling interrupts while reading MIDI data. If a program must disable interrupts while reading MIDI data, it should not do so for longer than 270 microseconds.

In cases where compliance with these restrictions is impossible, you can use the `MidiInputPoll` vector. This vector is provided for those applications that must disable interrupts for dangerously long periods. To call `MidiInputPoll`, execute a `JSL` to $E101B2. If the MIDI Tool Set has not been started up, or if the MIDI input process has not been started, the vector will return immediately. Any MIDI data that were present on the call to the vector will appear in the input buffer that you allocated with `MidiControl`.

◆ *Note:* If you need the values of the A, X, and Y registers, you must save them yourself before calling the vector. The direct-page and data bank registers are preserved. `MidiInputPoll` must be called only in full native mode.

▲ **Warning** Do not call `MidiInputPoll` before loading the MIDI Tool Set in a system with a Sound Tool Set version earlier than 2.3 or system software earlier than 4.0. Doing so will cause a system failure. ▲

If you use the `MidiInputPoll` vector, you must ensure that it is called at least every 270 microseconds, or MIDI data may be lost. A call to the vector when no data are present returns in from 8 to 30 microseconds, and when data are present the vector can take up to 450 microseconds, at 150 microseconds per character read.

You can call `MidiReadPacket` and `MidiWritePacket` inside interrupt-service routines, because they perform polling automatically. Other tool sets do not perform MIDI polling, so MIDI applications should not make calls to other tool sets in interrupt-service routines.

▲ **Warning** Do not make MIDI Tool Set calls other than `MidiReadPacket` and `MidiWritePacket` from interrupt-service routines. Doing so can cause unpredictable system failure. ▲

Whenever possible, you should use MIDI interface cards that support MIDI data buffering. By storing some received data, these cards relieve the time constraints on your application.

MIDI and other sound-related tool sets

If you use the recommended versions of the Note Synthesizer, Note Sequencer, and Sound Tool Set (see Chapter 51, "Tool Locator Update," for details), these tool sets are fully compatible with the MIDI Tool Set and do not cause MIDI data losses. It is possible to write programs that use the Note Sequencer to play notes on the internal voices of the Apple IIGS and on an external MIDI synthesizer while simultaneously accepting MIDI input from an external keyboard and translating it to Note Synthesizer commands to play the notes.

The MIDI clock

This section discusses the technique currently used to generate MIDI time-stamps. Note that this technique may not be used on future Apple IIGS machines. Any application that employs a similar technique to implement timing may be incompatible with future systems.

Properly time-stamping MIDI input data requires a clock with resolution better than one millisecond. When a long stream of MIDI data is received in a short time period (such as when the user plays a large chord on a MIDI keyboard), each note must be accurately time-stamped. However, the Apple IIGS cannot process interrupts quickly enough to satisfy this requirement.

To provide a reasonable clock resolution, the Apple IIGS MIDI time-stamp is implemented using one of the system's DOC generators. The MIDI tool set loads the DOC with a 256-byte waveform consisting of consecutive values from $01 to $FF (followed by an additional byte of $FF), and sets the DOC to play this waveform at zero volume. When a MIDI character is received, the time-stamping routine uses the value from this DOC for the low-order byte of the time-stamp. The system obtains the high-order 3 bytes from a counter that is incremented each time the DOC cycles through its waveform (once every 19.45 milliseconds at the default clock rate). This technique reduces the system interrupt load to a manageable level while also providing sufficiently fine clock resolution to correctly process MIDI data.

Because the MIDI clock is actually a DOC generator, you cannot use that generator while the clock is running; under these circumstances, only 13 generators are available for general use. The clock also uses the first 256 bytes of DOC RAM for its waveform, so running the clock reduces the memory available for application waveforms. While the clock is running you must not use the Sound Tool Set's free-form synthesizer (the `FFStartSound` call). The frequency and duration of Sound Tool Set interrupts also interferes with the MIDI Tool Set's ability to perform its services often enough to prevent data loss.

Input and output buffer sizing

You should adjust the MIDI input buffer size for the amount of data you can expect to receive before the application processes it. Any process that competes with the application for processor time, such as Note Synthesizer calls to play complex envelopes, reduces the frequency at which the application can call `MidiReadPacket` and process the data in the input buffer. If the input buffer fills before it can be processed, data will be lost. Complex applications that use time-consuming tool calls therefore require large input buffers.

You can estimate the size of the needed input buffer from the size of the largest MIDI System Exclusive command you intend to receive. The default size of the input and output buffers is 8 KB. This is the size of two very large System Exclusive packets. You should choose a size that is large enough to accommodate two of the largest System Exclusive packets you expect to receive so that the MIDI tools can receive one packet and still have room for another. In packet mode, the MIDI Tool Set does not return a packet until it has received all of it, and MIDI data may continue to arrive while the tool set is returning the first packet.

The maximum buffer size is 32K, so your application may have to run the MIDI interface in raw mode (rather than packet mode) in order to support System Exclusive messages longer than 16KB.

You might want to keep statistics on the maximium number of data bytes in the input buffer in order to allow your application toadjust the input buffer size intelligently. Several MIDI Tool Set calls return information you can use for this purpose; see "MIDI Tool Calls" in this chapter for more detailed information on data returned by MIDI tool calls (especially the `miMaxInChars` and `miMaxOutChars` functions of the `MidiInfo` call).

Loss of MIDI data

The Apple 6850 driver was designed to work with non-buffered interface cards. When you use this driver and the desktop interface you may lose MIDI data. To avoid this data loss, you can

- Use a different, buffered 6850-based MIDI card along with a driver that supports the card
- Prevent the user from moving the cursor or making menu selections when your program is recording MIDI data

Number of MIDI interfaces

Note that the Apple IIGS can support only a single MIDI interface at a time. If you try to support more than one MIDI interface at the same time, you will lose MIDI data.

MIDI Housekeeping calls

The following MIDI calls perform common tool set functions.

`MidiBootInit` $0120

Initializes the MIDI Tool Set; called only by the Tool Locator.

▲ **Warning** An application must never make this call.▲

Parameters This call has no input or output parameters. The stack is unaffected.

Errors None

C

```
extern pascal void MidiBootInit();
```

`MidiStartUp` $0220

Starts up the MIDI tools for use by an application. Applications should make this call before any other calls to the MIDI tools. Normally an application must next call `MidiDevice` to load a MIDI device driver, and then `MIDIControl` to allocate any necessary input or output buffers.

Parameters

Stack before call

Previous contents	
userID	Word—Application user ID (for the Memory Manager)
dPageAddr	Word—Beginning of MIDI direct-page space
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$0812	`noSAppInitErr`	The Sound Tool Set has not been started up.

C

```
extern pascal void MidiStartUp(userID, dPageAddr);

Word      userID, dPageAddr;
```

dPageAddr Must specify 3 pages of page-aligned direct-page space for the MIDI tools.

`MidiShutDown` $0320

Shuts down the MIDI Tool Set. An application that uses the MIDI tools should make this call before it quits. `MidiShutDown` deallocates the input and output buffers, stops the MIDI clock and deallocates its generator, and shuts down the hardware interface. The call's actions take place immediately, so the application should take any necessary steps to see that all recent MIDI output has been sent before shutting down the tools (see the `MidiControl` call).

Parameters This call has no input or output parameters. The stack is unaffected.

Errors None

C

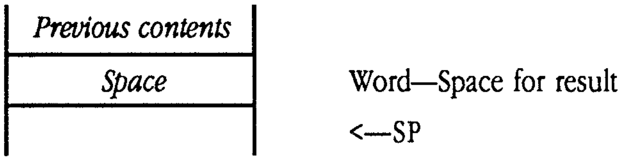
```
extern pascal void MidiShutDown();
```

`MidiVersion` $0420

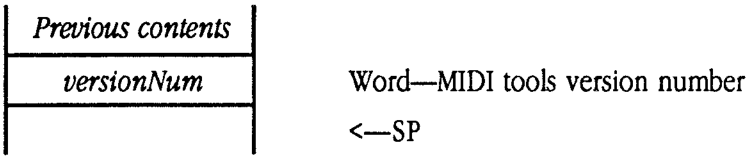
Returns the version number of the currently loaded MIDI tools. For information on the format of the returned *versionNum*, see Appendix A, "Writing Your Own Tool Set," in Volume 2 of the *Toolbox Reference*.

Parameters

Stack before call



Stack after call



Errors None

C `extern pascal Word MidiVersion();`

`MidiReset` $0520

Resets the MIDI tools; called by system reset.

This tool call causes the MIDI device driver reset routine to be invoked, in order to allow for reset-specific processing that may differ from shutdown processing.

▲ **Warning** An application must never make this call.▲

Parameters This call has no input or output parameters. The stack is unaffected.

Errors None

C

```
extern pascal void MidiReset();
```

`MidiStatus` $0620

Returns a Boolean value of TRUE if the MIDI tools are active and FALSE if they are not.

◆ *Note:* If your program issues this call in assembly language, initialize the result space on the stack to NIL. Upon return from `MidiStatus`, your program need only check the value of the returned flag. If the MIDI Tool Set is not active, the returned value will be FALSE (NIL).

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
	<—SP

Stack after call

Previous contents	
activeFlag	Word—Boolean; TRUE if the tool set is active
	<—SP

Errors None

C `extern pascal Word MidiStatus();`

MIDI tool calls

All the MIDI Tool Set calls are new calls, added to the toolbox since publication of the first two volumes of the *Apple IIGS Toolbox Reference.*

The routines used to work with the MIDI Tool Set are `MidiControl`, `MidiDevice`, `MidiClock`, `MidiInfo`, `MidiReadPacket`, and `MidiWritePacket`. Four of these calls are multifunction calls, which perform different actions depending on a control parameter passed to them. The workhorse of the group is `MidiControl`, which performs 18 different functions, depending on the control function parameter. The other multipurpose calls are `MidiDevice`, `MidiClock`, and `MidiInfo`.

`MidiClock` $0B20

Controls operation of the optional time-stamp clock. The clock ticks once every 76 microseconds with default settings, and allows MIDI data to be input and output with precise timing. The *funcNum* parameter specifies which clock function to perform, and the *arg* parameter provides the argument to the selected function.

Parameters

Stack before call

Previous contents	
funcNum	Word—Specifies `MidiClock` function number
– *arg* –	Long—Argument passed to `MidiClock` function
	<—SP

Stack after call

Previous contents	
	<—SP

Errors See the `MidiClock` function descriptions below.

C

```
extern pascal void MidiClock(funcNum, arg);

Word      funcNum;
Long      arg;
```

funcNum Specifies the `MidiClock` function to be performed. Four different functions are provided for clock control. They are

0 `miSetClock` The value of *arg* becomes the new value of the time-stamp clock. The most significant bit of the *arg* parameter must be set to 0. There is a limit to the precision with which the clock can be set. The least significant byte of the time-stamp clock will always be 0 if the clock is stopped. If the clock is running, the value of the least significant byte will be undefined for the purposes of this call. The result is that an application can set the clock only to within 20 milliseconds of a particular value with the clock frequency at its default value.

Errors None

1 miStartClock

Allocates a DOC generator, writes consecutive values from $01 through $FF into the first page of the DOC RAM, and starts the clock. By default, the clock starts counting at 0. If the application stops the clock and restarts it, the clock starts with the same value it had when it stopped, unless the value is changed with an miSetClock call. Note that only the high-order 3 bytes are preserved; the low-order byte always starts at $01. You should call miStartClock before miStartInput if you are using time-stamps.

Start the MIDI clock before starting to receive or transmit MIDI data. The process of starting the clock is time-consuming and disables interrupts, so it could cause MIDI data to be lost if it is done while the application is receiving a MIDI transmission. The Sound Tool Set and the Note Synthesizer *must* be loaded and started up before this call is issued.

Errors

$0810	noDOCFndErr	No DOC or DOC RAM was found.
$1921	nsNotAvail	No DOC generator was available.
$1923	nsNotInit	The Note Synthesizer was not started.

2 miStopClock

Stops the MIDI time-stamp clock and releases the DOC generator and its associated RAM for use by the Note Synthesizer. The MIDI tools time-stamp MIDI data received while the clock is stopped with the value of the stopped clock in the high-order 3 bytes, and the low-order byte set to $00. The MIDI tools will not send any output packets with time-stamps greater than the value of the stopped clock until the clock is restarted or reset.

Errors None

3 `miSetFreq` Sets the frequency for the MIDI time-stamp clock. The *arg* parameter contains the number of clock ticks to be processed per second. Valid values lie in the range from 1 to 65,535; a 0 value specifies the default setting (13,160 ticks per second).

Since the clock frequency affects the rate of playback, be careful to set the clock frequency at playback to the same value that was used when the sequence was recorded, unless you intend to vary the tempo.

See the `MidiInfo` call for information about how to read the current clock frequency and value.

Errors

$2009	`miBadFreqErr`	Unable to set MIDI clock to the specified frequency (use the `MidiInfo` tool call to get the current value).

`MidiControl` $0920

Performs 18 different control functions required by the MIDI Tool Set.

The *funcNum* parameter selects which function is to be performed, and the *arg* parameter passes any argument required by that function.

Parameters

Stack before call

Previous contents	
funcNum	Word—Specifies `MidiControl` function number
– *arg* –	Long—Argument passed to `MidiControl` function
	<—SP

Stack after call

Previous contents	
	<—SP

Errors See the `MidiControl` function descriptions.

C

```
extern pascal void MidiControl(funcNum, arg);

Word      funcNum;
Long      arg;
```

funcNum Specifies the `MidiControl` function to be performed:

0 `miSetRTVec`

Sets the real-time vector. The *arg* parameter contains the address of a service routine in the application. When the MIDI Tool Set receives MIDI real-time commands, it calls this service routine. A value of 0 in this parameter disables the service routine. See "MIDI Tool Set service routines" earlier in this chapter for more information on the real-time command routine.

Errors None

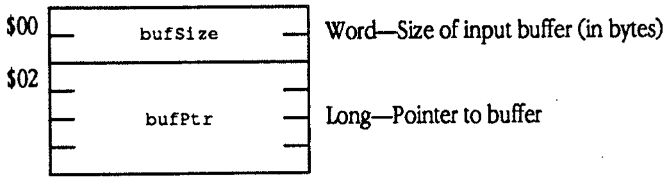
1 `miSetErrVec`

Sets the real-time error vector. The *arg* parameter contains the address of a service routine in the application. The MIDI Tool Set calls this routine in the event of a MIDI real-time error. A value of 0 in the parameter disables the service routine. See "MIDI Tool Set service routines" earlier in this chapter for more information on the real-time error routine.

Errors None

2 `miSetInBuf`

Sets the MIDI input buffer. The *arg* parameter contains a pointer to a 6-byte record. The fields of this record are



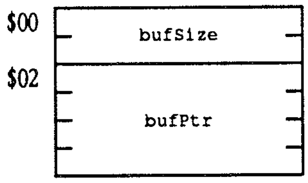
If the `bufPtr` parameter is set to 0, the MIDI Tool Set will allocate the input buffer. If the `bufSize` parameter is set to 0, the MIDI tools will allocate a buffer 8 KB in size. Note that these parameters are independent; your program may set either one of them to 0. If the application allocates the buffer, it must be nonpurgeable, in a fixed location, and must not cross bank boundaries. The size must be greater than or equal to 32 bytes and less than or equal to 32 KB.

Errors

$2002	`miArrayErr`	Array was an invalid size.
Memory Manager errors		Returned unchanged.

3 `miSetOutBuf`

Sets the MIDI output buffer. The *arg* parameter contains a pointer to a 6-byte record. The fields of this record are



Word—Size of output buffer (in bytes)

Long—Pointer to buffer

If the `bufPtr` parameter is set to 0, the MIDI Tool Set will allocate the output buffer. If the `bufSize` parameter is set to 0, the MIDI Tool Set will allocate a buffer 8 KB in size. Note that these parameters are independent; your program may set either one of them to 0. If the application allocates the buffer, it must be nonpurgeable, in a fixed location, and must not cross bank boundaries. The size must be greater than or equal to 32 bytes and less than or equal to 32KB.

Errors

$2002	`miArrayErr`	Array was an invalid size.
Memory Manager errors		Returned unchanged

4 `miStartInput`

Starts an interrupt-driven process that reads MIDI data into the MIDI Tool Set's input buffer. Any data being received when this call is made are discarded until the first MIDI status byte is received. An application can retrieve these data with a `MidiReadPacket` call. The *arg* parameter contains the address of a service routine to be called when the first packet is available in a previously empty input buffer. The system will call the service routine immediately if a complete MIDI packet is available in the input buffer when this function is called. A value of 0 disables this service routine.

Errors

$2007	`miNoBufErr`	No buffer allocated.
$200C	`miNoDevErr`	No device driver loaded.

5 `miStartOutput`

Starts an interrupt-driven process that writes application MIDI data to the MIDI Tool Set's output buffer. Your application uses `MidiWritePacket`calls to queue data to this process. The *arg* parameter contains the address of a service routine called when the output buffer becomes completely empty. A value of 0 disables this service routine.

Errors

$2007	`miNoBufErr`	No buffer allocated.
$200C	`miNoDevErr`	No device driver loaded.

6 `miStopInput`

Causes the MIDI Tool Set to ignore MIDI data until the next `miStartInput` call.

Errors None

7 `miStopOutput`

Halts MIDI output until the next `miStartOutput` call.

Errors None

8 `miFlushInput`

Discards the contents of the current input buffer.

Errors

$2007	`miNoBufErr`	No buffer allocated.

9 `miFlushOutput`

Discards the contents of the current output buffer. The *arg* parameter selects the method:

***arg* value**	**Action**
$0000 00XX	Wait for the current packet to finish transmission, then turn off all notes that have not been turned off in channel XX. If XX = $10, turn off notes in all channels.
$0001 00XX	Wait for current packet to finish transmission, then turn off all possible notes (pitch $00 through $7F) in channel XX. If XX = $10, turn off notes in all channels. Note that this option may take several seconds to complete.
$FFFF XXXX	Discard the contents of the output buffer immediately without turning off any notes.

Some synthesizers may require a short delay between the high-speed NoteOff commands generated by this function. In such cases, use the `miSetDelay` function of this tool call to control that delay. The NoteOff side-effect can be useful for shutting off notes.

Errors

$2005	`miOutOffErr`	MIDI output disabled.
$2007	`miNoBufErr`	No buffer allocated.

10 `miFlushPacket`

If there is a complete packet in the input buffer, this call discards that packet. If there is no complete packet available, this call does nothing. This call is especially useful for discarding large System Exclusive packets that are of no interest to your application.

Errors

$2007	`miNoBufErr`	No buffer allocated.

11 `miWaitOutput`

Ceases execution until the output buffer becomes empty. This function may never return if output is disabled.

Errors

$2007	`miNoBufErr`	No buffer allocated.

12 `miSetInMode`

Set input mode. The *arg* parameter selects the input mode.

***arg* value**	**Input mode**
0	Raw mode. MIDI data is converted to packets, with length-of-packet and time-stamp bytes added to the front of each packet.
1	Packet mode. Packet mode is the default mode. MIDI data is converted to packets, with length-of-packet and time-stamp bytes added to the front of each packet. Running status bytes, which MIDI may discard to abbreviate transmitted data, are restored.

The input buffer is cleared when this call is made because the input buffer cannot contain data in more than one format at a time.

Errors None

13 `miSetOutMode`

The *arg* parameter selects the output mode.

***arg* value**	**Input mode**
0	Raw mode. This mode is very similar to packet mode, but no attempt is made to keep track of which notes are on. Running status optimization is still performed unless explicitly disabled by `miOutputStat`. Because no record is kept of which notes are on, all notes that are turned on must be explicitly turned off.
1	Packet mode. Packet mode is the default mode. Your application must format output data into valid MIDI packets (see "MIDI packet format" in this chapter for details). The MIDI tools track NoteOn and NoteOff commands.

Your program should wait for a clear output buffer before switching modes. If the output buffer contains mixed-mode data, the MIDI tools may not track NoteOn and NoteOff commands correctly.

Errors None

14 `miClrNotePad`

Erases the MIDI Tool Set's record of which notes are on and which are off. This call causes the tool set's record to show that all notes are off.

Errors None

15 `miSetDelay`

Sets a delay value for use with MIDI synthesizers that cannot process MIDI data at the full MIDI transfer rate. The low word of *arg* specifies a minimum delay between packet sends in units of 76 microseconds. The delay mechanism is most effective when the MIDI Tool Set clock is running, because it can use the clock to time the delay. If the clock is not running, the tool set must use code loops to create the delay, which is inherently less accurate, and which uses more processor time. The default delay value is 0, or no delay.

Many synthesizers may need a delay value in order to correctly process the many high-speed NoteOff commands generated by the `miFlushOutput` function.

Errors None

16 `miOutputStat`

Enables or disables transmission of standard MIDI running status. When running status is enabled, MIDI status bytes are sent only when they change or are otherwise absolutely necessary. This optimization speeds transmission and reduces CPU overhead, but can cause malfunctions if the synthesizer and computer disagree on the current value of the status byte.

The low word of *arg* contains the enable/disable flag

$0000	Disable running status
$0001	Enable running status

Whatever the value of the parameter, the next MIDI packet after this call contains a status byte, so it can be useful to make this call periodically to ensure that the Apple IIGS and the external device agree about the current value of the status byte.

Errors None

17 `miIgnoreSysEx`

Specifies whether to ignore MIDI System Exclusive data. System Exclusive packets begin with the value $F0. If the application configures the MIDI Tool Set to ignore System Exclusive packets, the system will not buffer them, and the application will not receive them. The *arg* parameter contains a flag indicating how to process System Exclusive data:

$0000	Ignore System Exclusive data
$0001	Accept System Exclusive data (default)

Errors None

`MidiDevice` $0A20

Allows an application to select, load, and unload device drivers for use with the tools. `MidiDevice` loads and unloads MIDI device drivers, which allow the MIDI tools to drive a particular MIDI interface. The present version of the MIDI Tool Set supports the Apple MIDI Interface and ACIA 6850 MIDI Interface cards.

The call interprets the *driverInfo* parameter as the address of the driver to be loaded. The *funcNum* parameter specifies whether the driver is to be loaded or unloaded.

Parameters

Stack before call

Previous contents	
funcNum	Word—Specifies `MidiDevice` function number
– *drvrPtr* –	Long—Pointer to device driver information
	<—SP

Stack after call

Previous contents	
	<—SP

Errors See the `MidiDevice` function descriptions.

C

```
extern pascal void MidiDevice(funcNum, drvrPtr);

Word       funcNum;
Pointer    arg;
```

funcNum Specifies the `MidiDevice` function to be performed

0 **Not yet implemented**

1 `miLoadDrvr`

Loads the specified device driver into memory, after shutting down and unloading any previously loaded device drivers. It then initializes the newly loaded driver. The *drvrPtr* parameter points to a device driver record, which specifies a device driver to be loaded.

Errors

$2008	`miDriverErr`	Specified device driver invalid.
$2080	`miDevNotAvail`	MIDI interface not available.
$2081	`miDevSlotBusy`	Specified slot not selected in Control Panel.
$2082	`miDevBusy`	MIDI interface already in use.
$2084	`miDevNoConnect`	No connection to MIDI interface.
$2086	`miDevVersion`	ROM version or machine type incompatible with device driver.
$2087	`miDevIntHndlr`	Conflicting interrupt handler installed.

2 `miUnloadDrvr`

Shuts down and unloads the currently loaded device driver. Terminates MIDI transmission or reception if they are currently active. Releases memory occupied by the device driver.

Errors None

drvrPtr The record pointed to by the *drvrPtr* parameter contains device driver information:

$00	`slotNumber`	Word
$02	`slotFlag`	Word
$04	`driverPath`	Pascal string

`slotNumber` Specifies the system slot containing the MIDI interface to be supported by the driver being loaded. Valid values range from $0000 through $0007.

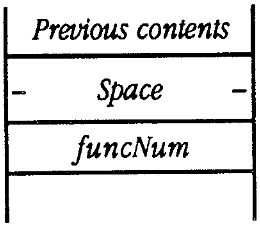
`slotFlag`	Indicates whether type of slot specified in `slotNumber`: $0000 Internal slot $0001 External slot
`driverPath`	Pascal string containing the GS/OS™ pathname to the file containing the device driver to be loaded. Pascal strings consist of data preceded by a length byte. The pathname cannot exceed 64 characters in length.

`MidiInfo` $0C20

Returns certain information about the state of the MIDI tools. The *funcNum* parameter can specify nine different functions, whose results are returned in *infoResult.*

Parameters

Stack before call

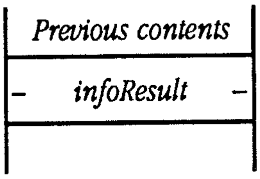


Long—Space for result

Word—Specifies `MidiInfo` function number

<—SP

Stack after call



Long—Result of `MidiInfo` function

<—SP

Errors See the `MidiInfo` function descriptions.

C

```
extern pascal Long MidiInfo(funcNum);

Word        funcNum;
```

funcNum Specifies the `MidiInfo` function to be performed

0 `miNextPktLen`

Returns the number of bytes in the next MIDI packet. On return, *infoResult* contains the length of the next complete MIDI packet in the input buffer, including the four-byte time-stamp at the beginning of the packet. Note that if there is no complete packet in the input buffer, this function returns a value of 0.

Errors

$2007	`miNoBufErr`	No buffer allocated.

1 `miInputChars`

Returns the number of bytes of MIDI data waiting in the input buffer. On return, *infoResult* contains the number of bytes of MIDI data currently stored in the input buffer, including any time-stamp and length data (6 bytes per packet), error codes, and up to 12 bytes of extra space at the end of the buffer due to call latency. It is therefore only a rough estimate of the number of bytes in the buffer. Your application can use this call to monitor whether the input buffer is large enough.

Errors

$2007	`miNoBufErr`	No buffer allocated.

2 `miOutputChars`

Returns the number of bytes of MIDI data waiting in the output buffer. On return, *infoResult* contains the number of bytes waiting to be transmitted from the MIDI output buffer, including time-stamp and length data (6 bytes per packet), error codes, and up to 12 bytes of extra space at the end of the buffer due to call latency. It is therefore only a rough estimate of the number of bytes in the buffer.Your application can use this call to monitor whether the output buffer is large enough.

Errors

$2007	`miNoBufErr`	No buffer allocated.

3 `miMaxInChars`

Returns the largest number of bytes that were stored in the input buffer since the last `miMaxInChars` call or since the buffer was last flushed. This call is especially useful for deriving statistics on buffer utilization.

Errors None

4 `miMaxOutChars`

Returns the largest number of bytes that were stored in the output buffer since the last `miMaxOutChars` call or since the output buffer was last flushed. This call is especially useful for deriving statistics on buffer utilization.

Errors None

5 **Not yet implemented**

6 **Not yet implemented**

7 `miClockValue`

Returns the current value of the MIDI Tool Set time-stamp clock. If the clock is stopped then the low-order byte of the result is 0.

Errors None

8 `miClockFreq`

Returns the current MIDI Tool Set clock frequency in ticks per second. The default value is 13,160 ticks per second.

Errors None

`MidiReadPacket` $0D20

Moves MIDI data from the MIDI Tool Set's input buffer to a specified location and returns the length of the packet in bytes. If no packet is available, the call returns a 0. For more information on MIDI packets, see "MIDI packet format" earlier in this chapter.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
– *bufPtr* –	Long—Pointer to buffer for received data
bufSize	Word—Length, in bytes, of the receive buffer
	<—SP

Stack after call

Previous contents	
result	Word—Number of bytes actually returned
	<—SP

Errors			
	$2001	`miPacketErr`	Incorrect packet length received.
	$2002	`miArrayErr`	Array was an invalid size. The array was too small to store the next packet. Use a larger array or discard the packet with the `MidiControl` call.
	$2003	`miFullBufErr`	MIDI data discarded because of buffer overflow.
	$2007	`miNoBufErr`	No buffer allocated.
	$2083	`miDevOverrun`	MIDI interface overrun by input data; interface not serviced quickly enough.
	$2084	`miDevNoConnect`	No connection to MIDI interface.
	$2085	`miReadErr`	Error reading MIDI data.

C

```
extern pascal Word MidiReadPacket(bufPtr, bufSize);

Pointer   bufPtr;
Word      bufSize;
```

MidiWritePacket $0E20

Queues the specified MIDI packet into the MIDI Tool Set's output buffer. If the packet is successfully written to the output buffer, this call returns the number of bytes written. If the buffer is too full to accommodate the packet, `MidiWritePacket` returns 0. For more information on MIDI packets, see "MIDI packet format" in this chapter.

`MidiWritePacket` returns within 1/50 of a second, but the output process waits until the MIDI clock value is equal to or greater than the output packet's time-stamp before sending it. Your program should issue this call before before starting the MIDI output process (with the `miStartOutput` function of the `MidiControl` tool call).

In packet mode, `MidiWritePacket` assumes that only complete MIDI commands are passed to it, and that the first byte of each packet is a MIDI status byte. The MIDI Tool Set uses these assumptions to track NoteOn and NoteOff commands. In raw mode the MIDI Tool Set makes no attempt to track NoteOn and NoteOff commands, so the intelligent NoteOff function provided in `MidiControl` will not work, and packets may contain complete, partial, or multiple MIDI commands. In either mode the MIDI Tool Set omits the MIDI status byte unless its value has changed since the last one was transmitted. You can, however, disable running status transmission entirely by using the `MidiControl` call.

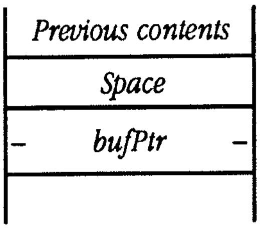
If the MIDI clock is stopped, then all packets with a time-stamp less than or equal to the value of the clock are immediately transmitted, and all packets with a value greater than the clock remain in the buffer unless the clock is restarted and its value becomes greater than the time-stamps.

Two special time-stamp values override normal output buffer processing, irrespective of MIDI clock state. Any packet with a zero time-stamp is written immediately upon reaching the head of the output buffer. Any packet with a negative time-stamp value is considered to be a real-time command and the packet is inserted at the head of the output queue for immediate transmission. Note that MIDI real-time messages may be transmitted in the middle of non–real-time MIDI messages.

The MIDI Tool Set routines do not sort the packets in the output buffer, therefore, a packet at the head of the output queue can delay transmission of any packets behind it that have earlier time-stamp values.

Parameters

Stack before call

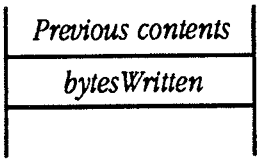


Word—Space for result

Long—Pointer to buffer containing output data

<—SP

Stack after call



Word—Number of bytes actually written

<—SP

Errors None

C

```
extern pascal Word MidiWritePacket(bufPtr);

Pointer   bufPtr;
```

MIDI Tool Set error codes

This section lists the error codes that may be returned by MIDI Tool Set calls.

Value	Name	Definition
$2000	`miStartUpErr`	MIDI Tool Set not started up.
$2001	`miPacketErr`	Incorrect packet length received.
$2002	`miArrayErr`	Array was an invalid size.
$2003	`miFullBufErr`	MIDI data discarded because of buffer overflow.
$2004	`miToolsErr`	Required tools inactive or incorrect version.
$2005	`miOutOffErr`	MIDI output disabled.
$2007	`miNoBufErr`	No buffer allocated.
$2008	`miDriverErr`	Specified device driver invalid.
$2009	`miBadFreqErr`	Unable to set MIDI clock to the specified frequency (use the `MidiInfo` tool call to get the current value).
$200A	`miClockErr`	MIDI clock wrapped to 0.
$200B	`miConflictErr`	Two processes competing for MIDI input.
$200C	`miNoDevErr`	No device driver loaded.
$2080	`miDevNotAvail`	MIDI interface not available.
$2081	`miDevSlotBusy`	Specified slot not selected in Control Panel.
$2082	`miDevBusy`	MIDI interface already in use.
$2083	`miDevOverrun`	MIDI interface overrun by input data; interface not serviced quickly enough.
$2084	`miDevNoConnect`	No connection to MIDI interface.
$2085	`miDevReadErr`	Error reading MIDI data.
$2086	`miDevVersion`	ROM version or machine type incompatible with device driver.
$2087	`miDevIntHndlr`	Conflicting interrupt handler installed.

Chapter 39 **Miscellaneous Tool Set Update**

This chapter documents new features of the Miscellaneous Tool Set. The complete reference to the Miscellaneous Tools is in Volume 1, Chapter 14 of the *Apple IIGS Toolbox Reference.*

Error corrections

- On page 14-58 of the *Toolbox Reference*, Figure 14-3 shows the low-order bit of the User ID is reserved. This is not correct. The figure should show that the `mainID` field comprises bits 0–7, and that the `mainID` value of $00 is reserved.
- The sample code on page 14-28 contains an error. In the code to clear the 1 second IRQ source, the second instruction reads

  ```
  TSB    $C032
  ```

  This instruction should read

  ```
  TRB    $C032
  ```
- The descriptions of the `PackBytes` and `UnPackBytes` tool calls are unclear with respect to the *startHandle* parameter to each call. The stack diagrams correctly describe the parameter as a pointer to a pointer. However, the C sample code for each call defines *startHandle* as a handle. In both cases, *startHandle* is not a Memory Manager handle, but is a pointer to a pointer. Creating *startHandle* as a handle will cause unpredictable system behavior.

New features in the Miscellaneous Tool Set

- `ClearHeartBeat` and `DeleteHeartBeat` will turn off the interrupts that occur every 60th of a second if the following conditions are satisfied:
  - There are no remaining heartbeat tasks.
  - The interrupt handler installed in IRQ.VBL is the standard system interrupt handler, that is, no other interrupt handlers have been installed.
  - The standard mouse is not running in VBL interrupt mode.
- `SetVector` and `GetVector` support several new vectors. The new vectors are:

$80	Vector to memory mover
$81	Vector to set system speed
$82	Vector to slot arbiter
$86	Hardware independent interrupt vector
$87	MIDI interrupt vector (IRQ-MIDI)

◆ *Note:* `SetVector` no longer validates the input vector number. Therefore, you must be extremely careful when using this call in order to avoid corrupting memory.

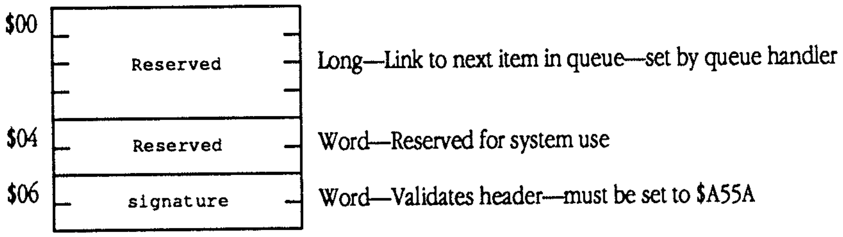
Queue handling

The Miscellaneous Tool Set now provides a generalized queue handler that can be used by other tools and applications. A queue is defined here as an ordered collection of variable-length data elements. Each data element must be preceded by a standard queue header. Your application must format the queue elements and format the correct header. The queue handler provides calls to add or remove elements from a queue (`AddToQueue` and `DeleteFromQueue`).

A queue is identified by its header pointer, a pointer to the first element in the queue. Your application establishes and maintains the header pointer. Do not add this first element to the queue with `AddToQueue`.

Figure 39-1 shows the format of the queue header.

- **Figure 39-1** Queue header layout


$00
Reserved
Long—Link to next item in queue—set by queue handler
$04
Reserved
Word—Reserved for system use
$06
signature
Word—Validates header—must be set to $A55A

Application data immediately follow the header.

See "New Miscellaneous Tool Set calls" later in this chapter for details on `AddToQueue` and `DeleteFromQueue`.

Interrupt state information

The Miscellaneous Tool Set now provides a set of calls that allow you to obtain interrupt-time system state information. These calls should be particularly useful to developers of debuggers or interrupt handlers. With these new calls, your program can get or set system interrupt state information.

All these new calls use a standard interrupt state record. Note that the tool calls have been designed to support an extensible state record. In the future, the record may grow in size, but existing program code should still work.

Figure 39-2 shows the format of the interrupt state record. For more information about any of these registers, see the *Apple IIGS Firmware Reference.*

- **Figure 39-2** Interrupt state record layout

$00	irq_A	Word—A register contents
$02	irq_X	Word—X index register contents
$04	irq_Y	Word—Y index register contents
$06	irq_S	Word—S (stack) register contents
$08	irq_D	Word—D (direct) register contents
$0A	irq_P	Byte—P (program status) register contents
$0B	irq_DB	Byte—DB (data bank) register contents
$0C	irq_e	Byte—Bit 0 is the emulation mode bit
$0D	irq_K	Byte—K (program bank) register contents
$0E	irq_PC	Word—PC (program counter) register contents
$10	irq_state	Byte—STATEREG byte value
$11	irq_shadow	Word—SHADOW byte (low byte) and CYAREG (high byte) values
$13	irq_mslot	Byte—SLTROMSEL byte

New Miscellaneous Tool Set calls

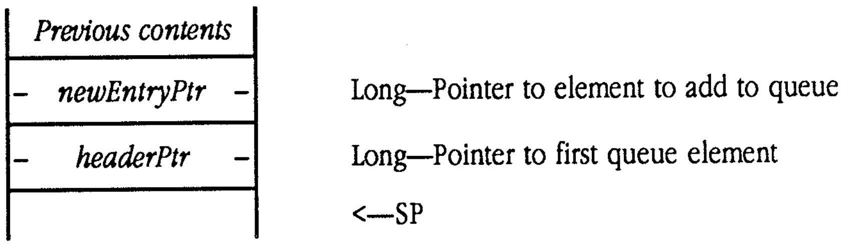
The following sections introduce several new Miscellaneous Tool Set calls.

AddToQueue $2E03

Adds the specified entry to a queue.

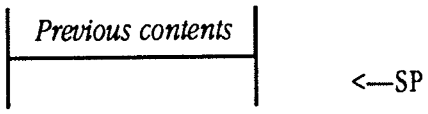
Parameters

Stack before call



Previous contents
– *newEntryPtr* –
– *headerPtr* –
Long—Pointer to element to add to queue
Long—Pointer to first queue element
<—SP

Stack after call



Errors	$0381	`invalidTag`	Signature value invalid in element header
	$0382	`alreadyInQueue`	Specified element already in queue

C

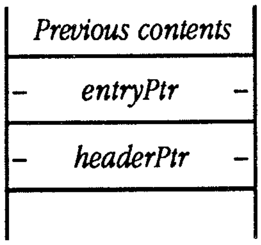
```
extern pascal void AddToQueue(newEntryPtr,
              headerPtr);

Pointer   newEntryPtr, headerPtr;
```

`DeleteFromQueue` $2F03

Deletes a specified element from a queue.

Parameters

Stack before call

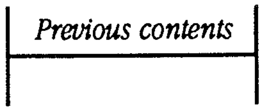


Long—Pointer to element to delete from queue

Long—Pointer to first queue element

<—SP

Stack after call



<—SP

Errors			
	$0380	`notInList`	Specified element not found in queue
	$0381	`invalidTag`	Signature value invalid in element header

C

```
extern pascal void DeleteFromQueue(entryPtr,
           headerPtr);

Pointer    entryPtr, headerPtr;
```

`GetCodeResConverter` $3403

Returns the address of a routine that loads code resources. This is a Miscellaneous tool call because the loader is not in directly accessible memory (it lives in the bank 1 language card, which may or may not be addressable at any given time).

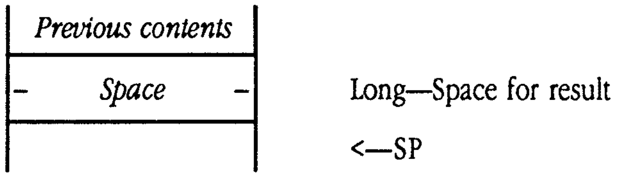
Your program would use this call in conjunction with the `ResourceConverter` Resource Manager tool call (see Chapter 45, "Resource Manager,"). For example, the Control Manager issues the following call during its startup processing:

```
ResourceConverter(GetCodeResConverter(),
                  rCtlDefProc,
                  LogConverterIn+SysConverterList);
```

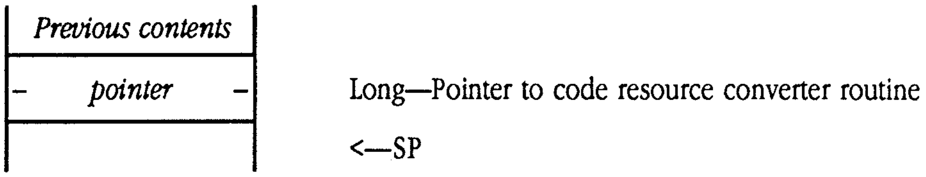
After issuing this call, all future calls to the Resource Manager to load resources of type `rCtlDefProc` will use the Miscellaneous tools routine to bring the resource into memory. Note that this routine does not preserve preserve the memory attributes of the converted resource (for more information on resource converters, see Chapter 45, "Resource Manager," later in this book).

Parameters

Stack before call



Stack after call



Errors None

C `extern pascal Pointer GetCodeResConverter();`

GetInterruptState $3103

Copies the specified number of bytes into a specified input interrupt state record from the system interrupt variables. The copy always starts from the beginning of the interrupt state record. Use the `SetInterruptState` call to set the contents of the system interrupt state record.

Parameters

Stack before call

Previous contents	
– *intStateRcdPtr* –	Long—Pointer to interrupt state record
bytesDesired	Word—Number of bytes to copy from system to record
	<—SP

Stack after call

Previous contents	
	<—SP

Errors None

C

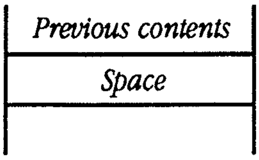
```
extern pascal void GetInterruptState(intStateRcdPtr,
                    bytesDesired);

Pointer    intStateRcdPtr;
Word       bytesDesired;
```

`GetIntStateRecSize` $3203

Returns the size (in bytes) of the interrupt state record. This call allows application programs to work with extended interrupt state records.

Parameters

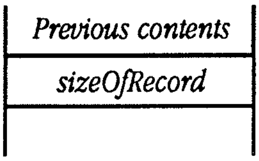
Stack before call



Word—Space for result

<—SP

Stack after call



Word—Length of interrupt state record, in bytes

<—SP

Errors None

C `extern pascal Word GetIntStateRecSize();`

GetROMResource $3503

This call is only for use by system firmware

ReadMouse2 $3303

Returns the mouse position, status, and mode. This call does not support journaling. Refer to Chapter 14, "Miscellaneous Tool Set," in Volume 1 of the *Toolbox Reference* for information about the `ReadMouse` tool call.

▲ **Warning** Applications should never make this call.▲

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
Space	Word—Space for result
Space	Word—Space for result
	<—SP

Stack after call

Previous contents	
xPosition	Word—X position of mouse
yPosition	Word—Y position of mouse
statusMode	Word—Status and mode bytes
	<—SP

Errors	$0309	`unCnctdDevErr`	Pointing device is not connected

C `extern pascal MouseRec ReadMouse2();`

ReleaseROMResource $3603

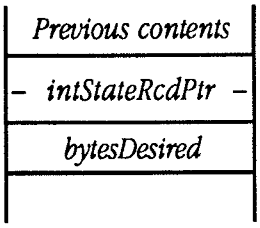
This call is only for use by system firmware

SetInterruptState $3003

Copies the specified number of bytes from the input interrupt state record into the system interrupt variables. The copy always starts from the beginning of the interrupt state record. Use the `GetInterruptState` call to read the system interrupt state record.

Parameters

Stack before call

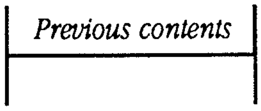


Long—Pointer to interrupt state record

Word—Number of bytes to copy from record to system

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void SetInterruptState(intStateRcdPtr,
            bytesDesired);

Pointer   intStateRcdPtr;
Word      bytesDesired;
```

Chapter 40 **Note Sequencer**

This chapter documents the Note Sequencer. This is new documentation, not previously presented in the *Apple IIGS Toolbox Reference.*

About the Note Sequencer

The Note Sequencer is a collection of routines that implement a sequencer in the Apple IIGS. The sequencer is an interpreter for a simple music programming language designed to play music in the background. It can be used to play music from a static file as long as any other active system tasks do not disable interrupts.

This sequencer plays melodies by using data stored in a specific format. It does not provide the means to create these data structures, and so an application must provide its own tools for building new sequences.

The Note Sequencer works with the Note Synthesizer, and it can work with the MIDI tools if you choose.

◆ *Note:* The Note Synthesizer, the Note Sequencer, and the MIDI Tool Set refer to the software tools provided with the Apple IIGS, not to any separate instrument or device. The MIDI tools are software tools for use in controlling external instruments, which may be connected through a MIDI interface device.

Using the Note Sequencer

To use the Note Sequencer, you must have loaded the following tool sets

- Tool Locator
- Memory Manager
- Sound Tool Set
- Note Synthesizer
- MIDI Tool Set (if MIDI is to be used)

All the required tool sets must be started up except the Sound Tool Set and the Note Synthesizer. The Note Sequencer makes the appropriate calls to start up these two tool sets. Refer to Chapter 51, "Tool Locator Update," for information on the specific version requirements of the Note Sequencer.

The Note Sequencer is interrupt-driven and can run in the background while other application tasks take place in the foreground. Therefore, interrupts must not be disabled while a sequence is being played. Any activity that disables interrupts interferes with execution of a sequence. Disk access, for example, disables interrupts, so an application cannot simultaneously gain access to a disk and play a sequence with the Note Sequencer. Note as well that any custom error and completion routines your application provides to the Note Sequencer (see "Error handlers and completion routines" in this chapter) also run with interrupts disabled and with a very low stack.

An application can normally rely on the Note Sequencer's built-in functions to synchronize a sequence correctly. For those applications that must directly control the timing of sequence execution, the `StepSeq` call has been provided. This call enables an application to explicitly control the execution of a sequence one step at a time.

Sequence timing

Normally you might think of a musical sequence as several independent tracks playing at the same time. For example, a musical passage might consist of a violin sound playing a melody accompanied by a viola and a flute. Musically, the three instruments will often play at once, sounding different notes. The Note Sequencer, however, always plays notes in sequence, one after another, however many instruments it is using to play them.

A chord, which is musically a group of different notes played at the same time, is executed by the Note Sequencer as a series of discrete notes played very quickly one after the other. For example, the Note Sequencer would play a chord consisting of F above middle C, A above middle C, and C one octave above middle C as a series of note commands:

Note	Duration
F4	4 counts
A4	4 counts
C5	4 counts

If the Note Sequencer waited for each note to finish before beginning the next one, the resulting passage would be three distinct notes of equal length, which is not what was intended. The Note Sequencer therefore provides a way to play the three notes with very little delay between them; so little, in fact, that they sound as though they were being played all at once.

If the `chord` bit is set to 1 in a note command, it indicates that the next note should be played to sound a chord with the current one. If, on the other hand, the `delay` bit is set to 1, it indicates that the current note must be completed before the next one is played.

Using MIDI with the Note Sequencer

The appropriate calls must be made to the MIDI Tool Set to use MIDI with the Note Sequencer. Specifically, the MIDI tools must be started up, a device driver must be selected, and a MIDI output buffer must be allocated (see Chapter 38, "MIDI Tool Set," in this book for details). In addition, you must start the MIDI output process by issuing the `miStartOutput` function of the `MidiControl` tool call.

You must specify whether MIDI is to be used when you start up the Note Sequencer. If the high bit of the *mode* parameter is set when the `SeqStartUp` call is made, then MIDI is enabled. If a particular track is to use MIDI, it must be enabled for that track, using the `SetTrkInfo` call. Finally, the Note Sequencer checks tool call-specific and seqItem-specific flags for MIDI information, so that individual tool calls or commands can enable or disable MIDI.

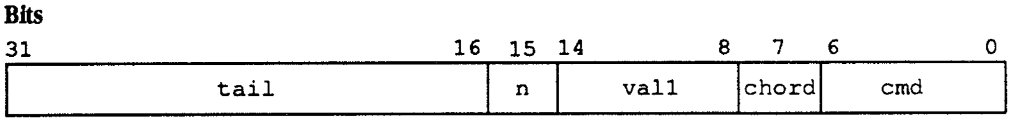
If all the appropriate flags—the *mode* flag, the *track* flag, and the command or tool call flag—are enabled, then MIDI commands are sent to external MIDI devices. This arrangement is designed to provide flexibility in execution. You could, for example, play only the drum parts of a sequence on external MIDI instruments, by enabling MIDI output only on the appropriate tracks, or you could play all parts on external MIDI instruments. Switching between the two modes of play would not require any modification of the sequence itself.

The Note Sequencer as a command interpreter

The Note Sequencer is actually a command interpreter. The commands it interprets are 32-bit data structures called seqItems, or sequence items. These 32-bit items contain information that the Note Sequencer needs to classify them as note commands, control commands, MIDI commands, or register commands, and to execute them properly.

The format of a seqItem is detailed in Figure 40-1.

- **Figure 40-1** Format of a seqItem



`cmd` For all commands except note commands, this is the command identifier, a 7-bit number that uniquely identifies the command. For example, the vibrato depth command has a `cmd` value of 4.

`chord` The `chord` bit is a Boolean value. If set, it specifies that the Note Sequencer should immediately execute the next seqItem with no delay.

`vall` The meaning of the `vall` field depends on the command being issued.

`note` The note bit identifies note commands. If bit `n` is set, the seqItem is a note command.

`tail` The format of the `tail` field depends upon the command type. It contains two or more fields with command-specific information in them.

There are four types of seqItems: note commands, control commands, MIDI commands, and register commands. Each type is organized in the same way, but the values in each part of the data structure have different meanings in the different commands.

Error handlers and completion routines

The Note Sequencer provides facilities allowing application programs to gain control at the end of a sequence, and when any errors are encountered during sequence processing. The Note Sequencer invokes completion routines when it has finished a sequence. The completion routine can then perform any necessary application-specific processing. Similarly, when an error occurs during sequence processing, the Note Sequencer calls a specified error handler, which can process the error in a manner appropriate to the current application.

When you start a sequence with the `StartSeq` tool call, you may specify a completion routine, an error handler, or both for the sequence. The *compRoutine* parameter points to the completion routine; the *errHndlrRoutine* parameter specifies the error Handler. Zero values for either parameter indicate to the Note Sequencer that there is no custom routine of the appropriate type available.

On entry to either type of routine, the Note Sequencer sets up the following conditions:

- Interrupts disabled
- Direct page set for Note Sequencer data area
- Data bank set to its value at the time of initial `SeqStartUp` tool call for the application; Note Sequencer restores this value when the routines return
- All registers saved
- Very little stack available

When a sequence started by `StartSeq` reaches its end, control will pass to the routine specified by *compRoutine.*

Whenever an error is encountered during sequence processing, the Note Sequencer will try to call the error handler for the sequence. A useful function of an error handler might be to place an error flag for the completion routine and make a `GetLoc` call to determine the location of the error.

The Note Sequencer passes error codes to the error handler in the A register. In step mode, the Note Sequencer will both report the error condition to the error handler and post it in the A register at the completion of the call to `StepSeq`. In interrupt mode, the Note Sequencer will only report the error to the application error handler.

◆ *Note:* The Note Synthesizer's timer oscillator is not forced on when an error occurs in the `StartSeq` call; neither the Note Synthesizer nor the Sound Tool Set will have been started.

Note commands

Note commands switch notes on and off. You can use note commands in two ways. You can issue a pair of NoteOn and NoteOff commands, turning a specified note on at a certain point and then explicitly turning it off, or you can issue a NoteOn command with a duration specified. In this case the Note Sequencer plays the note for a number of ticks equal to the value of the duration parameter, then turns the note off without the need for an explicit NoteOff command. Each tick occurs at an interval set by the Note Synthesizer's update rate (see Chapter 41, "Note Synthesizer," in this book for details).

■ **Figure 40-2** Note command format

Bits

31	30 27	26 16	15	14 8	7	6 0
`d`	`trk`	`duration`	`n`	`pitch`	`chord`	`volume`

`volume` Specifies note volume. Corresponds to MIDI velocity. A value of 0 indicates a NoteOff command.

`chord` Indicates that the seqItem is to be played simultaneously with the next seqItem. Do not set both the `chord` bit and the `delay` bit in the same item.

`pitch` Selects the pitch to be played. Values may range from 0 to 127. A value of 60 selects middle C (261.6 Hz). Adjacent values are one semitone apart. A value of 0 specifies a filler note (see "Filler notes" in this chapter for details).

`note` Always set to 1 for note commands. If this bit is not set to 1 in a seqItem, then the seqItem is not a note command.

`duration` Specifies the length of time the Note Sequencer is to play the note. Values may range from 0 to 2047, and specify the number of ticks the note is to be played.

A duration of 0 identifies the seqItem as a NoteOn command. A NoteOn seqItem is played continuously until the Note Sequencer finds a matching NoteOff.

`trk` Track number. Assigns notes to synthesizer voices and MIDI channels by specifying their track numbers. Values from $0 to $F are legal. Refer to the description of the `SetTrkInfo` call for more information.

`delay` If this bit is set to 1, the Note Sequencer must finish playing this seqItem before beginning to play the next one. The Note Sequencer cannot advance to the next seqItem until the `duration` is past. Do not set this bit to 1 if the `chord` bit is set to 1.

Filler notes

Filler notes are used to fill spaces in musical sequences. Intuitively, you might suppose that an application should use delays to create rests, but during a delay the Note Sequencer delays all its operations. Not only does it not play any notes until the delay period has elapsed, it also does not perform other services, such as turning notes off. You could cause strange behavior in a sequence if you used delays to create rests.

An alternative approach is to use filler notes. A filler note is simply a note command with a pitch value of 0. Such a note will be played by the Note Sequencer as though it were an ordinary note, but will not produce a tone. You can therefore use filler notes to fill out rests where you might have supposed a delay would be needed. For example, a passage may contain a chord consisting of notes with different duration, followed by a run of other notes. In this case, you would want to place a filler note at the end of the chord, so that you can easily vary the delay between the start of the chord and the start of the run.

Filler note command

`volume`	0
`chord`	1
`pitch`	Pitch value = 0
`note`	1
`duration`	Desired delay time
`trk`	0
`delay`	Set to 1 if a delay is desired

NoteOff command

`volume`	Note volume=0
`chord`	Set if the note is part of a chord
`pitch`	0 to 127; must be the same as matching NoteOn
`note`	1
`duration`	0
`trk`	0–15; must be the same as matching NoteOn
`delay`	0

NoteOn command

`volume`	Note volume; varies from 1 to 127
`chord`	Set if the note is part of a chord
`pitch`	Pitch value; varies from 0 to 127
`note`	1
`duration`	0
`trk`	0–15
`delay`	0

Control commands

Control commands are used to specify the characteristics of the Note Sequencer as it is playing the notes. They can control pitch bend, tempo, vibrato, and other note characteristics.

- **Figure 40-3** Control command format

Bits

31	30 27	26 24	23 16	15	14 8	7	6 0
d	trk	res	val2	n	val1	chord	cmd

`cmd` Command number.

`chord` The chord bit should be set to 1 in a control command. A 0 `chord` bit can sometimes cause unwanted delays in playing a sequence.

`val1` This field contains data specific to each command.

`note` Always set this bit to 0 for control commands. A `note` bit set to 1 causes the seqItem to be processed as a note command instead of a control command.

`val2` Contains data specific to each command.

`reserved` Reserved for control field. These bits should always be set to 0 unless otherwise specified.

`trk` Notes are assigned to synthesizer voices and to handlers by specifying their `trk` numbers. Legal values are $0 to $F.

`delay` The `delay` bit should always be set to 0 in control commands, since they have no duration.

CallRoutine command

`cmd`	30
`chord`	1
`val1`	0
`note`	0
bits 16-23	Low-order byte of routine address
bits 24-31	High-order byte of routine address

This command allows you to invoke program code from within a sequence being played by the Note Sequencer. This program code is then free to perform custom processing. The command specifies the low-order word of the routine address; the bank portion of the address matches the value of the data bank register at the time the Note Sequencer was started by your application.

On entry, interrupts are disabled and there is very little stack space remaining. The Note Sequencer saves its registers before issuing the call. However, the direct page and data bank registers are set for the Note Sequencer, so your routine code must change these in order to access application data. The routine should return with an `RTL` instruction.

If your application uses MIDI, this routine must be careful to poll MIDI every 270 microseconds to avoid losing MIDI data. See Chapter 38, "MIDI Tool Set," in this book for more information.

Jump command

`cmd`	3
`chord`	1
`val1`	`val1` is the high 7 bits of the destination
`note`	0
`val2`	`val2` is the low 8 bits of the Jump destination
`reserved`	0
`trk`	not used
`delay`	0

The Jump command is the Note Sequencer's equivalent of a jump or goto command in a conventional programming language. Execution of seqItems will continue with the item specified by `val1` and `val2`. The number given is a simple index into the series of seqItems (it is not a byte index into the seqItem array). The Jump command does not check the bounds of the sequence, and it is therefore possible to jump to an arbitrary area in memory that does not contain valid seqItems, which will produce unpredictable results.

Note that this command causes a jump in the sequence being processed. To jump to executable code from a sequence, use the CallRoutine command.

Pitch Bend command

`cmd`	0
`chord`	1
`val1`	Pitch wheel position. Values > 64 specify sharp pitch bend; values < 64 specify flat; intervals are expressed in fractions of the current pitch bend range
`note`	0
`val2`	No significance in the Pitch Bend command; the val2 field should always be set to 0 for Pitch Bend
`reserved`	Selects pitch bend assignment 0 selects both internal and MIDI pitch bend 1 selects internal pitch bend 2 selects MIDI pitch bend
`trk`	Track number
`delay`	0

The Pitch Bend command creates a bend effect in a played note. A control command expresses pitch bend as a value from 0 to 127. A value of 64 indicates no pitch bend, and the note is played at the pitch specified in its note command. The note is played at a pitch determined by its nominal pitch plus the pitch bend sharp or flat. The pitch changes immediately to the new value. As a result, the sequence must use a series of Pitch Bend commands to achieve the smooth portamento usually associated with a pitch bend.

The `reserved` field indicates whether the pitch bend is to affect the system's internal voices, external MIDI devices, or both. Note that your application must have specified MIDI support at `SeqStartUp` time in order for MIDI commands to be issued.

Program Change command

`cmd`	5
`chord`	1
`val1`	Instrument index from instrument table
`note`	0
`val2`	New MIDI program number, if the sequence is using MIDI
`reserved`	Specifies MIDI usage; legal values are 0 The Apple IIGS internal synthesizer and an external MIDI device 1 The Apple IIGS internal synthesizer only 2 External MIDI device only
`trk`	Track number; specifies which instrument program to change by specifying the track to which that instrument is assigned
`delay`	0

The Program Change command allows a sequence to change the instrument assigned to a track during play. The new instrument must be in the current instrument table for the new assignment to be possible.

If MIDI is enabled and the `reserved` field specifies that a MIDI command is to be issued, the Note Sequencer generates a MIDI Program Change command using `val2` for the program number.

Tempo command

`cmd`	1
`chord`	1
`val1`	New increment; he value may vary from 0 to 127
`note`	0
`val2`	0
`reserved`	0
`trk`	0
`delay`	0

This command sets the Note Sequencer's increment value. The increment value determines the number of ticks between updates in the execution cycle, so larger increments translate to slower tempos. The increment value is set to its initial value by the `SeqStartUp` tool call.

Turn Notes Off command

`cmd`	2
`chord`	1
`val1`	0
`note`	0
`val2`	0
`reserved`	0
`trk`	0
`delay`	0

This command turns off all notes currently being played, overriding any previous note commands. If MIDI support has been enabled, the system also turns off any active MIDI notes.

Vibrato Depth command

`cmd`	4
`chord`	1
`val1`	The new value for vibrato depth; the value may vary from 0 to 127
`note`	0
`val2`	Control number if a MIDI command is generated
`reserved`	0: internal and MIDI vibrato; 1: internal only
`trk`	Track number
`delay`	0

The Vibrato Depth command assigns a depth value to the vibrato effect used with the specified track. The vibrato effect is a modulation in the pitch of the voice assigned to the specified track. The `val1` value can range from 0 to 127, with larger values resulting in greater vibrato depth. A value of 0 disables vibrato, which conserves CPU cycles.

If MIDI support has been enabled and the `reserved` field indicates that a MIDI command is to be issued as well, `val2` specifies the MIDI control number, and `val1` specifies the new vibrato value for the MIDI Control Change command.

Register commands

Register commands provide the Note Sequencer with program control capabilities. The Note Sequencer maintains eight 8-bit pseudoregisters that can be used to implement looping and conditional branching structures. With register commands, an application can achieve the effect of control structures such as "if...then," "do...while," or "repeat...until" in sequences.

Each register occupies 8 bits of memory, but not all the commands use the full register. The IfGo Register and Set Register commands treat each register as if it were only 4 bits in size, using only the least significant 4 bits of the byte.

Bytes 2 through 9 of the Note Sequencer's direct page contain the pseudoregisters; these pseudoregisters are numbered 0 through 7. Note that Note Sequencer direct-page space starts $100 bytes beyond the location specified at `SeqStartUp` time. The intervening space is used by the Note Synthesizer and the Sound Tool Set.

■ **Figure 40-4** Register command format

Bits

31	30 27	26 24	23 16	15	14 8	7	6 0
d	trk	res	val2	n	val1	chord	cmd

`cmd` Command number.

`chord` The `chord` bit should be set to 1 in a register command. A 0 `chord` bit can sometimes cause unwanted delays in playing a sequence.

`val1` Contains data specific to each command. Generally specifies the register number for the command.

`note` Always set this bit to 0 for register commands. A `note` bit set to 1 causes the seqItem to be processed as a note command instead of a register command.

`val2` Contains data specific to each command.

`reserved` Reserved for control field. These bits should always be set to 0 unless otherwise specified.

`trk` Always set to 0 for register commands.

`delay`	The `delay` bit should always be set to 0 in register commands, since they have no duration.

Dec Register command

`cmd`	9
`chord`	1
`val1`	Low 3 bits contain the register number
`note`	0
`val2`	0
`reserved`	0
`trk`	0
`delay`	0

Decrements the value of the specified pseudoregister. If the value is 0 when the command is executed, the pseudoregister's value will wrap to $FF.

IfGo Register command

`cmd`	7
`chord`	1
`val1`	Low 3 bits contain the register number High 4 bits contain the value
`note`	0
`val2`	Offset: -128 to +127 seqItems
`reserved`	0
`trk`	0
`delay`	0

Tests the specified pseudoregister for the specified value. If the pseudoregister contains the supplied value, then execution continues with the seqItem at the offset specified in `val2`, calculated from the current seqItem. If the values do not match, execution continues with the next seqItem in sequence. The IfGo Register command does not check the bounds of the offset provided, so the value must be a valid one, or the effects will be unpredictable.

Inc Register command

cmd	8
chord	1
val1	Low 3 bits contain the register number
note	0
val2	0
reserved	0
trk	0
delay	0

Increments the value of the specified pseudoregister.

Set Register command

cmd	6
chord	1
val1	Low 3 bits contain the register number High 4 bits contain the value
note	0
val2	0
reserved	0
trk	0
d	0

Sets the specified pseudoregister to the specified value.

MIDI commands

MIDI commands enable an executing sequence to send data directly to MIDI devices that are connected to the Apple IIGS. All the standard MIDI commands are provided.

For MIDI commands to be enabled, the high bit of the *mode* parameter must be set when the `SeqStartUp` call is made. To produce MIDI output, your application must also have loaded and started up the MIDI Tool Set. For further information on the MIDI Tool Set, see Chapter 38, "MIDI Tool Set," in this book.

These commands are based on version 1.0 of the MIDI specification, version 1.0, which is not described in this documentation.

- **Figure 40-5** MIDI command format

Bits

31–24	23–16	15	14–8	7	6–0
`high`	`low`	`n`	`vall`	`chord`	`cmd`

`cmd` Command number.

`chord` The `chord` bit should be set to 1 in a MIDI command. A 0 chord bit can sometimes cause unwanted delays in playing a sequence.

`vall` Contains data specific to each command.

`note` Always set this bit to 0 for MIDI commands. A `note` bit set to 1 causes the seqItem to be processed as a note command instead of a MIDI command.

`low` Contains data specific to each command.

`high` Contains data specific to each command.

MidiChannelPressure command

`cmd`	15
`chord`	1
`val1`	Bits 8 through 11 are the channel number
`note`	0
`low`	Channel pressure
`high`	0

Sends a MIDI Channel Pressure command to the channel specified in `val1`. The new pressure value is specified by the `low` byte.

MidiControlChange command

`cmd`	13
`chord`	1
`val1`	Bits 8 through 11 are the channel number
`note`	0
`low`	Control number
`high`	Control value

Sends a MIDI Control Change command to the channel specified in `val1`. The control number is specified in the `low` byte and the control's new value is in the `high` byte.

MidiNoteOff command

`cmd`	10
`chord`	1
`val1`	bits 8 through 11 are the channel number
`note`	0
`low`	Note number
`high`	Velocity

Sends a MIDI NoteOff command on the channel number specified in `val1`. The note turned off is specified in two parts—a note number in the `low` byte and a velocity in the `high` byte.

MidiNoteOn command

`cmd`	11
`chord`	1
`val1`	Bits 8 through 11 are the channel number
`note`	0
`low`	Note number
`high`	Velocity

Sends a MIDI NoteOn command on the channel number specified in `val1`. The note turned on is specified in two parts—a note number in the `low` byte and a velocity in the `high` byte.

MidiPitchBend command

`cmd`	16
`chord`	1
`val1`	Bits 8 through 11 are the channel number
`note`	0
`low`	Pitch bend least significant byte
`high`	Pitch bend most significant byte

Sends a MIDI Pitch Bend command to the channel specified by `val1`. The new pitch bend value is specified by the high word of the command, with the least significant byte of the value in the `low` byte and the most significant byte in the `high` byte.

MidiPolyphonicKeyPressure command

`cmd`	12
`chord`	1
`val1`	Bits 8 through 11 are the channel number
`note`	0
`low`	Note number
`high`	Key pressure

Sends a MIDI Polyphonic Key Pressure command on the channel number specified in `val1`. The note affected is specified as a note number in the `low` byte of the high word. Its new key pressure is in the `high` byte

MidiProgramChange command

`cmd`	14
`chord`	1
`val1`	Bits 8 through 11 specify the MIDI channel number ($0–$0F)
`note`	0
`low`	Program number
`high`	0

Sends a MIDI Program Change command to the channel specified in `val1`. The program number is specified in the `low` byte.

MidiSelectChannelMode command

`cmd`	17
`chord`	1
`val1`	Bits 8 through 11 are the channel number
`note`	0
`low`	First data byte
`high`	Second data byte

Sends a MIDI Select Channel mode command to the channel specified in `val1`. The new MIDI channel mode is specified by two data bytes, the first of which is passed in the `low` byte and the second in the `high` byte.

MidiSetSysExlHighWord command

`cmd`	21
`chord`	1
`val1`	0
`note`	0
`low`	Low byte of high word
`high`	High byte of high word

The MIDI System Exclusive command passes a two-word address to its target. That address is a pointer to a MIDI packet. The high word of the address is specified by this command, while the low word is specified by the MidiSystemExclusive command. The MidiSetSysExlHighWord command must precede the MidiSystemExclusive command. See the discussion of that command for more information about the format and content of the MIDI packet.

MidiSystemExclusive command

`cmd`	18
`chord`	1
`val1`	0
`note`	0
`low`	Least significant byte of low word of MIDI packet address
`high`	Most significant byte of low word of MIDI packet address

The MIDI System Exclusive command passes a two-word address to its target. That address is a pointer to a MIDI packet. The low word of the address is specified by this command, while the high word is specified by the MidiSetSysExlHighWord command. The MidiSetSysExlHighWord command must precede the MidiSystemExclusive command.

Here is an example of a 3-byte System Exclusive command.

$00	`length`	Word—Length of data to follow; must be set to 8
$02	`timeStamp`	4 Bytes—Time-stamp for send time; 0 for immediate send
$06	`sysExclusive`	Byte—System Exclusive flag byte; must be set to $F0
$07	`data1`	Byte—First MIDI data byte
$08	`data2`	Byte—Second MIDI data byte
$09	`data3`	Byte—Third MIDI data byte

MidiSystemCommon command

`cmd`	19
`chord`	1
`val1`	Bits 8 through 10—low nibble of status byte value varies from 1 through 7 Bits 11 and 12—number of data bytes: 00 - 0 data bytes 01 - 1 data byte 10 - 2 data bytes 11 - invalid value
`note`	0
`low`	First data byte
`high`	Second data byte (if appropriate)

Sends one or two bytes of MIDI data. The first data byte is passed in the `low` byte and the second data byte, if there is one, is passed in the `high` byte.

MidiSystemRealTime command

`cmd`	20
`chord`	1
`vall`	0
`note`	0
`low`	Real-time number ($01–$07)
`high`	0

Sends a MIDI System Real-Time command. The real-time number is specified in the low 3 bits of the `low` byte.

Patterns and phrases

A pattern is any series of seqItems. The Note Sequencer plays melodies by carrying out the seqItem commands in specified patterns. A phrase is an ordered set of pointers to patterns or to other phrases. Since a phrase can contain pointers to other phrases, it is possible to nest phrases. The Note Sequencer supports up to 12 levels of phrase nesting.

Phrases and patterns have a similar layout. Both phrases and patterns are preceded by a longword header. For phrases, this header is set to 1; for patterns, the header is set to 0. The Note Sequencer can distinguish between phrases and patterns by examining this header value. The last longword in both phrases and patterns must be set to $FFFFFFFF, and is called the phrase done flag.

When a program calls the Note Sequencer to play a sequence, it passes a parameter containing a handle to the first byte of the top-level phrase. This phrase consists of an ordered series of pointers to the patterns or phrases to be played, followed by a longword value ($FFFFFFFF) that marks the end of a pattern or phrase.

Each pattern consists of an ordered series of seqItems. The seqItems describe the characteristics of each note to be played in the sequence. Control and register commands allow the characteristics of the notes to be modified and also allow the programmer to build complex sequences by using conditional looping and branching. In this sense, the Note Sequencer is a simple programming language.

The following paragraphs introduce a sample phrase and a sample pattern, so that you can see the similarities in their structure.

A phrase is identified by a header value of 1

```
topPhrase     dc     i2'0001'           ; low word
              dc     i2'0000'           ; high word
```

The phrase body consists of a series of pointers. Each pointer can point either to other phrases or to patterns, which are sequences of executable seqItems. For example,

```
              dc     i4'phrase1'
              dc     i4'pattern1'
              dc     i4'phrase2'
```

A phrase always ends with a phrase done flag:

```
              dc     i4'$FFFFFFFF'
```

A pattern is identified by a header value of 0

```
pattern1      dc     i2'0000'           ; low word
              dc     i2'0000'           ; high word
```

The body of a pattern consists of seqItems, such as:

```
        dc      i4'$880ABC74'       ; play C4, duration=10, volume=115
        dc      i4'$880ABE74'       ; play D4, duration=10, volume=115
        dc      i4'$880AB074'       ; play E4, duration=10, volume=115
```

Again, the pattern must end with a phrase done flag:

```
        dc      i4'$FFFFFFFF'
```

A sample Note Sequencer program

The following example contains 65816 assembly language source code for a simple Note Sequencer program.

```
               mcopy     s.m

DPPointer      gequ      $10
DPHandle       gequ      $14
HelpingHand    gequ      $18                 ; for dereferencing handles

*****************************************************************
Main           Start
               Using     Common

               clc                           ;set Native mode
               xce
               long
               phk                           ;Set the data bank to the
same
               plb                           ;as the program bank

               jsl       StartTools
               jsl       MakeWaves
               jsl       SetInstruments
               jsl       PlaySequence
               jmp       CleanUp

StartTools     _TLStartUp                    ; Tool Locator
               pha                           ; space for ID returned
               _MMStartUp
               pla
               sta       MyID
               PushLong  #0                    ; Get direct page for tools
               PushWord  #0
               PushWord  #$600
               PushWord  MyID
               PushWord  #$C005                ; direct page
               PushLong  #0
               _NewHandle
               pla
               sta       HelpingHand
               pla
               sta       HelpingHand+2
               lda       [HelpingHand]
               sta       DPPointer
               pha
               PushWord  #0                    ; either 320 or 640 mode
               PushWord  #0                    ; max size of scan line
               PushWord  MyID
               _QDStartup

               PushLong  #ToolTable
               _LoadTools

               lda       DPPointer
               clc
               adc       #$300                 ; QuickDraw used $300 bytes
               pha
               Pushword  #0
               Pushword  #$200
               Pushword  #$10
               _SeqStartup                     ; starts Synth&Sound Tools
               rtl

MakeWaves      ldx       #0                    ; index thru SoundBuffer
               lda       #1                    ; base of triangle
               sta       SoundBuffer
Triangle1      inx                             ; step thru buffer
               sta       SoundBuffer,x
               inc       A                     ; slope up in triangle
               cmp       #$ff                  ; byte limit for sound data
               bne       Triangle1
```

```
Triangle2      inx                            ; start down slope
               dec       A
               sta       SoundBuffer,x
               cmp       #$01                 ; dont want zeros
               bne       Triangle2
               inx                            ; pad 3 bytes with 1
               sta       SoundBuffer,x
               inx
               sta       SoundBuffer,x
               inx
               sta       SoundBuffer,x

               ldy       #2                   ; make 2 teeth
MakeTooth      lda       #$ff                 ; start high
Sawtooth1      inx
               sta       SoundBuffer,x
               dec       A                    ; ramp down
               bne       Sawtooth1
               dey                            ; do 2nd tooth
               bne       MakeTooth
               lda       #1                   ; pad last 2 bytes
               inx
               sta       SoundBuffer,x
               inx
               sta       SoundBuffer,x

               ldy       #255                 ; make a square wave
               lda       #1
Square1        inx
               sta       SoundBuffer,x
               dey
               bne       Square1

               ldy       #255
               lda       #255
Square2        inx
               sta       SoundBuffer,x
               dey
               bne       Square2
```

```
                ldy       #256                ; noise wave
                inx
Noise1          phy
                phx
                pha                           ; space for random result
                _Random
                pla
                bne       NotZero
                inc       A
NotZero         plx
                ply
                sta       SoundBuffer,x
                inx
                inx
                dey
                bne       Noise1

                PushLong  #SoundBuffer
                PushWord  #$100               ;DOC start address
                PushWord  #$800               ;byte count
                _WriteRamBlock

                rtl

SetInstruments  Pushlong  #InstTableHandle
                _SetInstTabl
                ldx       #3                  ; do 4 tracks
TrackLoop       phx
                Pushword  #64                 ; push the priority
                phx
                phx
                _SetTrkInfo
                plx
                dex
                bpl       TrackLoop
                rtl
```

```
PlaySequence   PushLong  #0                  ; No Error Handler Routine
               PushLong  #0                  ; No Completion Routine
               PushLong  #Sequence
               _StartSeq

               PushWord  #0
               PushWord  #0
               _ReadChar
               pla

               Pushword  #0                  ; No Next Phrase
               _StopSeq
               rtl

CleanUp        _SeqShutdown
               _EMShutdown
               _QDShutdown
               PushWord  MyID
               _DisposeAll
               _Quit QuitParams

               End
****************************************************************
Common         Data
QuitParams     dc        i4'0,0,0'      ; Quit back to calling program
MyID           ds        2
tooltable      dc        i'2,26,0,25,0' ; two tools, numbers 26 & 25
SoundBuffer    ds        2048           ; 4 waves, 512 bytes each
InstTableHandle dc i4'InstTable'
InstTable      dc   i2'4'
               dc   i4'Sawtooth'
               dc   i4'Square'
               dc   i4'Triangle'
               dc   i4'Noise'
```

```
Sawtooth       dc   i1'127'               ; envelope breakpoint 1
               dc   i1'0,127'             ; Increment Value 1
               dc   i1'120'               ; breakpoint 2
               dc   i1'20,1'              ; increment 2
               dc   i1'120'               ; sustain at 120
               dc   i1'0,0'               ; zero increment is sustain
segment
               dc   i1'0'                 ; release to 0 volume
               dc   i1'60,12'             ; slowly
               dc   i1'0,0,0'             ; pad with extra breakpoint
               dc   i1'0,0,0'             ; increment pairs till the
               dc   i1'0,0,0'             ; total is 8
               dc   i1'0,0,0'
               dc   i1'3'                 ; release segment is 3rd segment
               dc   i1'32'                ; priority increment
               dc   i1'2,80,90,0,1,1'     ; pbrange,vibdep,vibf,spare,A,B
               dc   i1'127,1,2,6,0,12'    ; topkey,addr,size,ctrl,pitch
               dc   i1'127,1,2,1,0,12'    ; halt b, to be swapped in by a.

Square         dc   i1'127'               ; envelope breakpoint 1
               dc   i1'0,127'             ; Increment Value 1
               dc   i1'120'               ; breakpoint 2
               dc   i1'20,1'              ; increment 2
               dc   i1'120'               ; sustain at 120
               dc   i1'0,0'               ; zero increment is sustain
segment
               dc   i1'0'                 ; release to 0 volume
               dc   i1'60,12'             ; slowly
               dc   i1'0,0,0'             ; pad with extra breakpoint
               dc   i1'0,0,0'             ; increment pairs till the
               dc   i1'0,0,0'             ; total is 8
               dc   i1'0,0,0'
               dc   i1'3'                 ; release segment is 3rd segment
               dc   i1'32'                ; priority increment
               dc   i1'2,80,90,0,1,1'     ; pbrange,vibdep,vibf,spare,A,B
               dc   i1'127,3,2,6,0,12'    ; topkey,addr,size,ctrl,pitch
               dc   i1'127,3,2,1,0,12'    ; halt b,to be swapped in by a.
```

```
Triangle        dc    i1'127'              ; envelope breakpoint 1
                dc    i1'0,127'            ; Increment Value 1
                dc    i1'120'              ; breakpoint 2
                dc    i1'20,1'             ; increment 2
                dc    i1'120'              ; sustain at 120
                dc    i1'0,0'              ; zero increment is sustain
segment
                dc    i1'0'                ; release to 0 volume
                dc    i1'60,12'            ; slowly
                dc    i1'0,0,0'            ; pad with extra breakpoint
                dc    i1'0,0,0'            ; increment pairs till the
                dc    i1'0,0,0'            ; total is 8
                dc    i1'0,0,0'
                dc    i1'3'                ; release segment is 3rd segment
                dc    i1'32'               ; priority increment
                dc    i1'2,80,90,0,1,1'    ; pbrange,vibdep,vibf,spare,A,B
                dc    i1'127,5,2,6,0,12'   ; topkey,addr,size,ctrl,pitch
                dc    i1'127,5,2,1,0,12'   ; halt b,to be swapped in by a.

Noise           dc    i1'127'              ; envelope breakpoint 1
                dc    i1'0,127'            ; Increment Value 1
                dc    i1'120'              ; breakpoint 2
                dc    i1'20,1'             ; increment 2
                dc    i1'120'              ; sustain at 120
                dc    i1'0,0'              ; zero increment is sustain
segment
                dc    i1'0'                ; release to 0 volume
                dc    i1'60,12'            ; slowly
                dc    i1'0,0,0'            ; pad with extra breakpoint
                dc    i1'0,0,0'            ; increment pairs till the
                dc    i1'0,0,0'            ; total is 8
                dc    i1'0,0,0'
                dc    i1'3'                ; release segment is 3rd segment
                dc    i1'32'               ; priority increment
                dc    i1'2,80,90,0,1,1'    ; pbrange,vibdep,vibf,spare,A,B
                dc    i1'127,7,2,6,0,12'   ; topkey,addr,size,ctrl,pitch
                dc    i1'127,7,2,1,0,12'   ; halt b, to be swapped in by a.
```

```
Delay          equ       $80000000
T1             equ       $08000000
T2             equ       $10000000
T3             equ       $18000000
T0             equ       $0
Qtr            equ       $40000
Half           equ       $80000
Note           equ       $8000
C4             equ       $3C00
D4             equ       $3E00
E4             equ       $4100
F4             equ       $4200
G4             equ       $4300
Chord          equ       $80

Sequence       dc        i4'Phrase1'
Phrase1        dc        i4'1'                         ; phrase header
               dc        i4'Phrase2'
               dc        i4'Pattern1'
               dc        i4'Phrase2'
               dc        i4'Pattern1'
               dc        i4'Pattern2'
               dc        i4'$FFFFFFFF'                 ; terminator

Phrase2        dc        i4'1'                         ; phrase header
               dc        i4'Pattern2'
               dc        i4'Pattern1'
               dc        i4'$FFFFFFFF'                 ; terminator

Pattern1       dc        i4'0'                         ; pattern header
               dc        i4'Delay+T0+Qtr+Note+C4+127'  ; full volume
               dc        i4'T1+Qtr+Note+C4+Chord+127'
               dc        i4'Delay+T1+Qtr+Note+G4+127'
               dc        i4'Delay+T0+Half+Note+F4+127'
               dc        i4'$FFFFFFFF'                 ; terminator
```

```
Pattern2        dc          i4'0'                          ; pattern header
                dc          i4'T2+Note+G4+Chord+127'       ; note on
                dc          i4'Note+Half'                  ; filler note
                dc          i4'Delay+T2+Qtr+Note+F4+127'
                dc          i4'Delay+T3+Qtr+Note+D4+127'
                dc          i4'T3+Note+G4+Chord+127'       ; note off
                dc          i4'2'                          ; allnotesoff
                dc          i4'$FFFFFFFF'                  ; terminator

                End
```

Note Sequencer housekeeping calls

The following sections discuss Note Sequencer calls that perform common tool set functions.

`SeqBootInit` $011A

Initializes the Note Sequencer.

▲ **Warning** This call must not be made by an application.▲

Parameters This call has no input or output parameters. The stack is unaffected.

C

```
extern pascal void SeqBootInit();
```

`SeqStartUp` $021A

SeqStartUp starts up the Note Sequencer and performs all the necessary initializations for the tool set. It also makes startup calls to the Sound Tool Set and the Note Synthesizer, so an application should not start up those tool sets before making this call.

Your application must make sure that the MIDI Tool Set has been started before issuing this call.

Parameters

Stack before call

Previous contents	
dPageAddr	Word—Beginning of Note Sequencer direct page
mode	Word—MIDI flag
updateRate	Word—Rate of interrupt generation
increment	Word—Number of interrupts per system tick
	<—SP

Stack after call

Previous contents	
	<—SP

Errors			
	$1A03	`startedErr`	The Note Sequencer is already started up.
	$1A07	`nsWrongVer`	The Note Synthesizer is a version that is incompatible with the Note Sequencer.
	Sound Tool Set errors		Returned unchanged
	Note Synthesizer errors		Returned unchanged

C

```
extern pascal void SeqStartUp(dPageAddr, mode,
                    updateRate, increment);

Word      dPageAddr, mode, updateRate, increment;
```

dPageAddr Specifies the location for the Note Sequencer's direct page. This direct page must actually be three pages of bank zero memory, starting at the specified address. The first page is used by the Note Synthesizer and Sound Tool Set, and the other two by the Note Sequencer. All three pages must be locked and page-aligned.

mode Determines whether the Note Sequencer will operate in interrupt mode, in which updates are performed automatically as interrupts occur, or in step mode, in which updates occur only when explicit step commands are issued. If the low bit of *mode* is set to 0, then interrupts are used; if it is set to 1, then step mode is used.

The high bit of the *mode* parameter also determines whether MIDI processing is enabled. If an application uses MIDI commands or wants to support automatic generation of appropriate MIDI commands, then the high bit must be set to 1.

updateRate Specifies how often the Note Sequencer will update its actions, using interrupts. For example, an *updateRate* value of 500 specifies that the Note Sequencer will receive interrupts at 200 hertz, or every 5 milliseconds. A value of 250 means that interrupts will be available at 100 hertz, or every 10 milliseconds (500 is the default value) The same rate is used by the Note Synthesizer to update its instruments' envelopes.

increment Specifies how many interrupts constitute one tick of the Note Sequencer counter. If *updateRate* is 500 and *increment* is 20, then one tick will take 100 milliseconds. The Note Sequencer gets interrupts every 5 milliseconds, and the counter is incremented every 20 interrupts. If a quarter note equals 5 ticks, then it lasts half a second, which corresponds to a tempo of 120 beats per minute. In general, you can compute the number of beats per minute by using the following formula:

$$B = (24 * \mathit{updateRate}) / (\mathit{increment} * T)$$

B is beats per minute and T is the number of ticks in a beat.

Typical *updateRate* values might be:

60 Hz	60/0.4 = 150; *updateRate* = 150
100 Hz	100/0.4 = 250; *updateRate* = 250
200 Hz	200/0.4 = 500; *updateRate* = 500

Larger values for *updateRate* result in greater control of a sequence's tempo and smoother envelopes. On the other hand, a higher *updateRate* also requires more processor time to service.

One general method for choosing appropriate *updateRate* and *increment* values is to decide on the shortest note you will want to play. Suppose the shortest note that you want to play is a sixteenth note. Assign sixteenth notes a value of 1. Eighth notes are twice as long, so assign them a value of 2. Quarter notes then receive a value of 4, half notes 8, and whole notes 16. Now decide how long you want a whole note to be and compute the *updateRate* and *increment* so that the duration comes out the way you want it.

Once you have set the *updateRate* value, it remains in effect; you can only change it by making the Note Synthesizer `NSSetUpdateRate` call, or by shutting down and restarting the Note Sequencer. You can change the *increment* value, and the Note Sequencer provides tempo calls that vary the tempo for you.

SeqShutDown $031A

Shuts down the Note Sequencer tool set. It frees any buffers that the tools may have allocated. An application that uses the Note Sequencer should call `SeqShutDown` before quitting.

Parameters This call has no input or output parameters. The stack is unaffected.

Errors

$1923	`nsNotInit`	The Note Synthesizer has not been started.
$1A05	`noStartErr`	The Note Sequencer has not been started up.
$0812	`noSAppInitErr`	The Sound Tool Set was not started up.

C

```
extern pascal void SeqShutDown();
```

`SeqVersion` $041A

Returns the version number of the Note Sequencer that is currently in use. Refer to Appendix A, "Writng Your Own Tool Set," in Volume 2 of the *Toolbox Reference* for information on the format and content of the returned *versionNum* value.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
	<—SP

Stack after call

Previous contents	
versionNum	Word—Note Sequencer version number
	<—SP

Errors None

C

```
extern pascal Word SeqVersion();
```

`SeqReset` $051A

Resets the Note Sequencer. `SeqReset` is called when the Apple IIGS system is reset. All internal notes presently being played are turned off.

▲ **Warning** This call must not be made by an application.▲

Parameters This call has no input or output parameters. The stack is unaffected.

C

```
extern pascal void SeqReset();
```

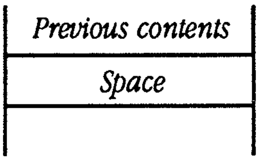
`SeqStatus` $061A

Returns a Boolean flag indicating whether or not the Note Sequencer is active. If the tool set is active, the flag is nonzero; otherwise it is zero.

◆ *Note:* If your program issues this call in assembly language, initialize the result space on the stack to NIL. Upon return from `SeqStatus`, your program need only check the value of the returned flag. If the Note Sequencer is not active, the returned value will be FALSE (NIL).

Parameters

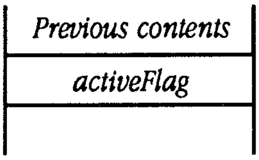
Stack before call



Word—Space for result

<—SP

Stack after call



Word—Boolean; TRUE if Note Sequencer is active

<—SP

Errors None

C `extern pascal Boolean SeqStatus();`

Note Sequencer calls

The following sections discuss the Note Sequencer tool calls.

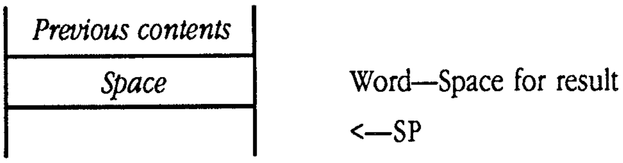
`ClearIncr` $0A1A

Sets the Note Sequencer's increment value to 0, halting the current sequence, and returns the previous increment value. Setting the increment to 0 does not disable the Note Sequencer's interrupts, so envelopes are still updated. This means that, although the sequence will not progress, notes being played when the increment was set to 0 may hang. This call is valid only while a sequence is playing.

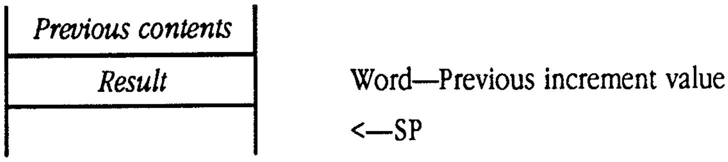
You might try using `SeqAllNotesOff` and `ClearIncr` when you want to stop a sequence and be able to start it again easily. A sequence stopped in this way can easily be restarted with a call to `SetIncr`.

Parameters

Stack before call



Stack after call



Errors None

C `extern pascal Word ClearIncr();`

`GetLoc` $0C1A

Returns certain information about the sequence that is playing. This call provides an index to the seqItem that is executing, the current pattern, and the nesting level. The nesting level indicates how deeply control has passed into a structure with phrases nested within phrases. A nesting level value of 0 indicates that the Note Sequencer is playing the top-level phrase.

For example, if the Note Sequencer is playing the third seqItem in Pattern 1, which occurs in Phrase 1, then `GetLoc` returns this information:

curPattItem = 3
curPhraseItem = 1
curLevel = 1

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
Space	Word—Space for result
Space	Word—Space for result
	<—SP

Stack after call

Previous contents	
curPhraseItem	Word—Current pattern in phrase specified by curLevel
curPattItem	Word—Current seqItem in pattern specified by curPhraseItem
curLevel	Word—Nesting level for current phrase
	<—SP

Errors None

C `extern LocRec GetLoc();`

`GetTimer` $0B1A

Returns the value of the Note Sequencer's tick counter. While the counter is advancing, the value returned is necessarily somewhat inexact, since the value changes as the call is executed. The call is valid only while a sequence is playing.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
	<—SP

Stack after call

Previous contents	
Result	Word—Current timer value
	<—SP

Errors None

C `extern pascal Word GetTimer();`

SeqAllNotesOff $0D1A

Switches off all notes that are playing, but does not stop the sequence. Thus, any notes that are held are turned off, but the sequence continues. Use this call to temporarily silence all instrument voices while a sequence is active. If the high bit of the *mode* parameter is set, then the Note Sequencer also turns off all external MIDI notes of which it is aware.

Parameters This call has no input or output parameters. The stack is unaffected.

Errors None

C

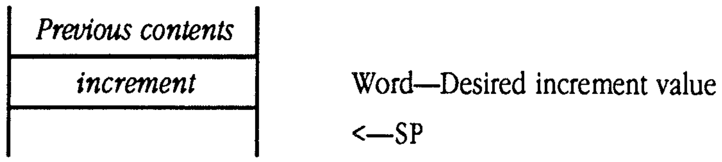
```
extern pascal void SeqAllNotesOff();
```

`SetIncr` $091A

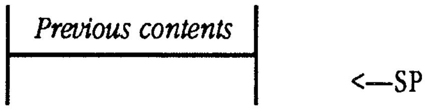
Sets the Note Sequencer's increment value. An application can use this facility to control the tempo of a sequence. If the increment value is set to 0, the sequence will halt.

Parameters

Stack before call



Stack after call



Errors None

C

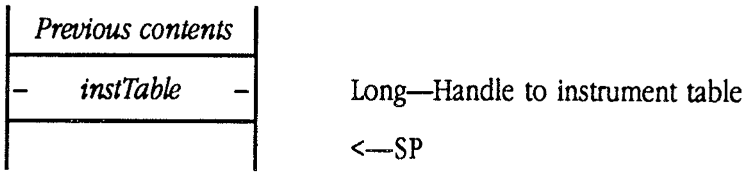
```
extern pascal SetIncr(increment);

Word      increment;
```

`SetInstTable` $121A

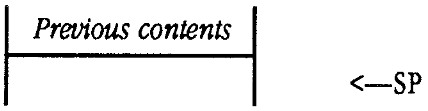
Sets the current instrument table to the one specified in *instTable.*

Parameters

Stack before call



Stack after call

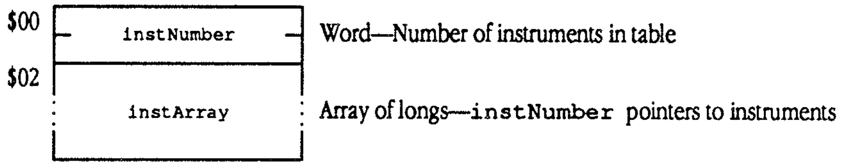


Errors None

C

```
extern pascal void SetInstTable(instTable);

Handle    instTable;
```

instTable The *instTable* parameter is a handle to an instrument table. The instrument table is a data structure in Apple IIGS memory that contains pointers to one or more instruments. The format of an instrument table is as follows:


$00
$02
instNumber
Word—Number of instruments in table
instArray
Array of longs—`instNumber` pointers to instruments

Note that the first pointer in the array corresponds to instrument 0. See Chapter 41, "Note Synthesizer," in this book for more information about instruments.

`SetTrkInfo` $0E1A

Assigns instruments in the current instrument table to logical tracks, and determines the priorities of the instruments so that the Note Sequencer can correctly allocate generators to them. An application should call `SetTrkInfo` for each track it uses, before starting to play a sequence.

If MIDI was enabled when the Note Sequencer was started up, then `SetTrkInfo` can be used to enable MIDI output on particular tracks. If the most significant bit of the ***trackNum*** parameter is set, then everything played on the specified track will produce MIDI output on the channel number specified by the second most significant byte of ***trackNum***. For example, a ***trackNum*** value of $8201 specifies that everything played on track 1 produces MIDI output on MIDI channel 2.

The application may disable the internal voices of the Apple IIGS for a specified track by issuing this call with the highest bit of the ***instIndex*** parameter set.

You must make a `SetInstTable` call before issuing this call.

Parameters

Stack before call

Previous contents	
priority	Word—Priority value
instIndex	Word—Index number for instrument (first instrument is number 0)
trackNum	Word—Track number for instrument
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$1A06	`instBndsErr`	The specified intrument number is out of the bounds of the instrument table.

C

```
extern pascal void SetTrkInfo(priority, instIndex,
                    trackNum);

Word        priority, instIndex, trackNum;
```

StartInts $131A

Enables interrupts. Use this call to restore normal functioning after a call to `StopInts`.

Parameters This call has no input or output parameters. The stack is unaffected.

Errors None

C `extern pascal StartInts();`

`StartSeq` $0F1A

Starts interpretation of a series of seqItems stored at the address specified by *sequence.*

Parameters

Stack before call

Previous contents	
– *errHndlrRoutine* –	Long—Pointer to error handler
– *compRoutine* –	Long—Pointer to completion routine
– *sequence* –	Long—Handle to sequence
	<—SP

Stack after call

Previous contents	
	<—SP

Errors			
	$1921	`nsNoneAvail`	Note Synthesizer error: no generator is available.
	$1A00	`noRoomMidiErr`	The Note Sequencer is tracking 32 notes that are currently playing; there is no room for a MIDI NoteOn.
	$1A01	`noCommandErr`	The current seqItem is not valid in its context.
	$1A02	`noRoomErr`	The sequence is nested more than twelve levels deep.
	$1A04	`noNoteErr`	Can't find the note for a NoteOff command.
	$1A05	`noStartErr`	The Note Sequencer has not been started up.
	$2004	`miToolsErr`	Required tools not active or wrong version
	$2007	`miNoBufErr`	No MIDI output buffer is allocated.

C

```
extern pascal void StartSeq(errHndlrRoutine,
          compRoutine, sequence);

Pointer   errHndlerRoutine, compRoutine;
Handle    sequence;
```

errHndlrRoutine The *errHandlrRoutine* parameter is a pointer to an error handling routine supplied by the application programmer. If *errHndlrRoutine* is set to NIL, then Note Sequencer will not invoke a routine. For information about error handling routines for the Note Sequencer, see "Error handlers and completion routines" in this chapter.

compRoutine The *compRoutine* parameter points to a routine to be called when `StartSeq` reaches the end of a sequence. If *compRoutine* is set to NIL, then Note Sequencer will not invoke a routine. For information about completion routines for the Note Sequencer, see "Error handlers and completion routines" in this chapter.

sequence The *sequence* parameter is a handle to the phrase to be executed by the Note Sequencer. The handle passed in *sequence* should be locked. If the Note Sequencer is running in interrupt mode, as specified by the *mode* parameter of the `SeqStartUp` call, then the Note Sequencer will simply start interpreting seqItems. If, however, the *mode* parameter specified that the Note Sequencer start up in step mode, then the `StartSeq` call must be followed by a series of calls to `StepSeq` to play the seqItems individually.

`StartSeqRel` $151A

Starts interpretation of a series of seqItems stored at the address specified by *sequence.* This call differs from `StartSeq` in that it uses relative addressing from the beginning of the sequence. That is, all phrase and pattern pointers are interpreted as offsets from the start of the sequence, rather than as absolute addresses. As a result, coding phrases and patterns is easier. Following the call description you will find a code sample showing how to specify these relative offsets.

The Note Sequencer uses the dereferenced value of *sequence* as the base address for all phrases and patterns. It does not check for overflow, and does not support negative offsets from the specified base address.

Parameters

Stack before call

Previous contents	
– *errHndlrRoutine* –	Long—Pointer to error handler
– *compRoutine* –	Long—Pointer to completion routine
– *sequence* –	Long—Handle to sequence
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$1921	`nsNoneAvail`	Note Synthesizer error: no generator is available.
$1A00	`noRoomMidiErr`	The Note Sequencer is tracking 32 notes that are currently playing; there is no room for a MIDI NoteOn.
$1A01	`noCommandErr`	The current seqItem is not valid in its context.
$1A02	`noRoomErr`	The sequence is nested more than twelve levels deep.
$1A04	`noNoteErr`	Can't find the note for a NoteOff command.
$1A05	`noStartErr`	The Note Sequencer has not been started up.
$2004	`miToolsErr`	Required tools not active or wrong version
$2007	`miNoBufErr`	No MIDI output buffer is allocated.

C

```
extern pascal void StartSeqRel(errHndlrRoutine,
              compRoutine, sequence);

Pointer   errHndlerRoutine, compRoutine;
Handle    sequence;
```

errHndlrRoutine The *errHandlrRoutine* parameter is a pointer to an error handling routine supplied by the application programmer. If *errHndlrRoutine* is set to NIL, then Note Sequencer will not invoke a routine. For information about error handling routines for the Note Sequencer, see "Error handlers and completion routines" in this chapter.

compRoutine The *compRoutine* parameter points to a routine to be called when `StartSeq` reaches the end of a sequence. If *compRoutine* is set to NIL, then Note Sequencer will not invoke a routine. For information about completion routines for the Note Sequencer, see "Error handlers and completion routines" in this chapter.

sequence The *sequence* parameter is a handle to the phrase to be executed by the Note Sequencer. The handle passed in *sequence* should be locked. If the Note Sequencer is running in interrupt mode, as specified by the *mode* parameter of the `SeqStartUp` call, then the Note Sequencer will simply start interpreting seqItems. If, however, the *mode* parameter specified that the Note Sequencer start up in step mode, then the `StartSeq` call must be followed by a series of calls to `StepSeq` to play the seqItems individually.

Sample sequence with relative addressing

The following example is a sequence presented in 65816 assembly language, showing how to set up relative addressing for `StartSeqRel`.

```
Delay       equ         $80000000
T1          equ         $08000000
T2          equ         $18000000
qtr         equ         $40000
hlf         equ         $80000
Note        equ         $8000
C4          equ         $3C00
D4          equ         $3E00
F4          equ         $4100
G4          equ         $4300
Chord       equ         $80

phrhndl     dc          i4'phr1-phrhndl'
phr1        dc          i4'01'                  ; it's a phrase
            dc          i4'phr2-phrhndl'
            dc          i4'pat1-phrhndl'
            dc          i4'phr2-phrhndl'
            dc          i4'pat1-phrhndl'
            dc          i4'pat2-phrhndl'
            dc          i4'$FFFFFFFF'           ; end of phrase 1

phr2        dc          i4'01'                  ; it's a phrase
            dc          i4'pat2-phrhndl'
            dc          i4'pat1-phrhndl'
            dc          i4'$FFFFFFFF'           ; end of phrase 2

pat1        dc          i4'00'                  ; it's a pattern
            dc          i4'Delay+T1+qtr+Note+C4+115'
            dc          i4'T1+qtr+Note+C4+Chord+115'
            dc          i4'Delay+T2+qtr+Note+G4+115'
            dc          i4'Delay+T1+hlf+Note+F4+115'
            dc          i4'$FFFFFFFF'           ; end of pat1
```

```
pat2        dc          i4'00'                    ; it's a pattern
            dc          i4'T1+Note+G4+Chord+115'        ; NoteOn
            dc          i4'Note+hlf'              ; filler note
            dc          i4'Delay+T2+qtr+Note+F4+115'
            dc          i4'Delay+T2+qtr+Note+D4+115'
            dc          i4'T1+Note+G4+Chord+115'        ; NoteOff
            dc          i4'$00000002'             ; AllNotesOff
            dc          i4'$FFFFFFFF'             ; end of pat2
```

`StepSeq` $101A

Increments the Note Sequencer counter, causing the appropriate seqItems in the current sequence to be processed. A `StepSeq` call is the equivalent of one tick of the Note Sequencer counter, which consists of a number of interrupts equal to the value of the *increment* parameter of the `SeqStartUp` call.

Parameters This call has no input or output parameters. The stack is unaffected.

Errors

$1921	`nsNoneAvail`	Note Synthesizer error: no generator is available.
$1A01	`noCommandErr`	The current seqItem is not valid in its context.
$1A02	`noRoomErr`	The sequence is nested more than twelve levels deep.
$1A04	`noNoteErr`	Can't find the note for a NoteOff command.

C

```
extern pascal void StepSeq();
```

StopInts $141A

Disables Note Synthesizer and Note Sequencer interrupts.

If the Note Sequencer is started up, and interrupts are enabled, the Note Synthesizer calls the Note Sequencer interrupt handler whenever an interrupt occurs. When no notes are being played, the overhead involved in this processing is unnecessary, so `StopInts` provides a way to cause the Note Synthesizer not to service the interrupts. To restart interrupt processing, use the `StartInts` call.

The `StartSeq` call starts interrupt processing automatically, and the `SeqShutDown` automatically halts it. No other Note Sequencer call affects interrupt processing except `StopInts`, `StartInts`, and `SeqShutDown`.

Parameters This call has no input or output parameters. The stack is unaffected.

Errors None

C

```
extern pascal void StopInts();
```

`StopSeq` $111A

Halts interpretation of a phrase. The ***next*** parameter specifies whether execution should continue if there are more phrases to be executed in the current sequence. If so, the next phrase begins. Otherwise, the sequencer simply stops and calls the application's completion routine. See "Error handlers and completion routines" in this chapter for more information on completion routines. If ***next*** is not equal to 0, then the current phrase terminates and execution continues with the next phrase.

If any notes are turned on with NoteOn commands, and a call to `StopSeq` halts the phrase in which they occur, they could continue to play forever, waiting for NoteOff commands that will never occur. You should thus take care to turn off any such notes before making a call to `StopSeq`.

Parameters

Stack before call

Previous contents	
next	Word—Boolean; TRUE to process remaining phrases
	<—SP

Stack after call

Previous contents	
	<—SP

Errors None

C

```
extern pascal StopSeq(next);

Boolean   next;
```

Note Sequencer error codes

This section lists the error codes that may be returned by Note Sequencer calls.

Value	Name	Definition
$1A00	`noRoomMidiErr`	The Note Sequencer is tracking 32 notes that are currently playing; there is no room for a MIDI NoteOn.
$1A01	`noCommandErr`	The current seqItem is not valid in its context.
$1A02	`noRoomErr`	The sequence is nested more than twelve levels deep.
$1A03	`startedErr`	The Note Sequencer is already started up.
$1A04	`noNoteErr`	Can't find the note for a NoteOff command.
$1A05	`noStartErr`	The Note Sequencer has not been started up.
$1A06	`instBndsErr`	The specified intrument number is out of the bounds of the instrument table.
$1A07	`nsWrongVer`	The Note Synthesizer is a version that is incompatible with the Note Sequencer.

Chapter 41 **Note Synthesizer**

This chapter documents the Note Synthesizer. This is new documentation, not previously presented in the *Apple IIGS Toolbox Reference.*

About the Note Synthesizer

The Note Synthesizer is a tool set that controls operation of the Apple IIGS Digital Oscillator Chip (DOC). With it, an application can turn the Apple IIGS into a digital synthesizer for playing music and generating sound effects. The Note Synthesizer provides far more control over a sound than the Sound Tool Set, and supports looping within a sound sequence and enveloping a sound.

Using the Note Synthesizer

An application that uses the Note Synthesizer must first start it up and write the wave information to the DOC RAM by using the Sound Tool Set's `WriteRAMBlock` call, then allocate Digital Oscillator Chip generators for its use with `AllocGen`. It can play musical notes with individual calls to `NoteOn` and `NoteOff` for each note that it plays. `NoteOn` starts a generator and the process that automatically updates envelopes as it plays its assigned instrument. When the application calls `NoteOff`, the Note Synthesizer enters the release phase of the envelope for that generator, and the note begins to die away.

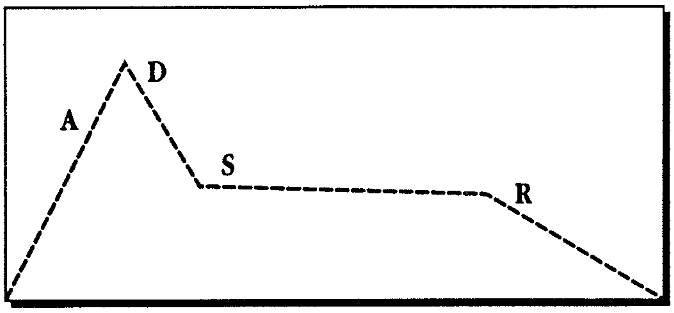
The Note Synthesizer requires that the Sound Tool Set be loaded and started up.

The sound envelope

The envelope describes the graph of a sound's loudness over time. The terms *loudness*, *amplitude*, and *volume* all refer to the same characteristic of a sound. In addition, the MIDI quantity velocity is normally mapped to a note's loudness, so that, for instance, the faster a key on a keyboard is struck, the louder its corresponding note will be. A note's envelope is what gives it its dynamic quality. A short, sharp sound has a steep, short envelope, and a long, smooth sound has a flatter, longer envelope.

A synthesizer's envelope is traditionally described in terms of **attack**, **decay**, **sustain**, and **release**, or **ADSR**. Figure 41-1 shows an example of a simple envelope described in terms of ADSR.

- **Figure 41-1** Sound envelope, showing attack, decay, sustain, and release



The attack portion of an envelope is the period when the sound is increasing from silence to its peak loudness. This part of the envelope determines the suddenness of a sound. A drumbeat or a plucked string has an extremely steep attack, whereas a bowed string or a softly blown wind instrument has a much flatter attack.

The decay part of the envelope is the period when the sound falls off from its peak loudness to the level it stays at, which is its sustain portion. Attack and decay together can be used to control a sound's percussiveness. Sounds with a steep attack and decay tend to sound plucked or percussive. A steep attack followed by a flat decay, or by little or no decay, blares like a loud trumpet. A very flat attack and decay produce a sound with a soft, smooth quality.

Sustain determines the note's overall perceived loudness and duration. A drumbeat has virtually no sustain or release; it consists almost entirely of attack and decay. A long, slow note on a violin, on the other hand, might have a very flat attack and decay, and a long, high sustain.

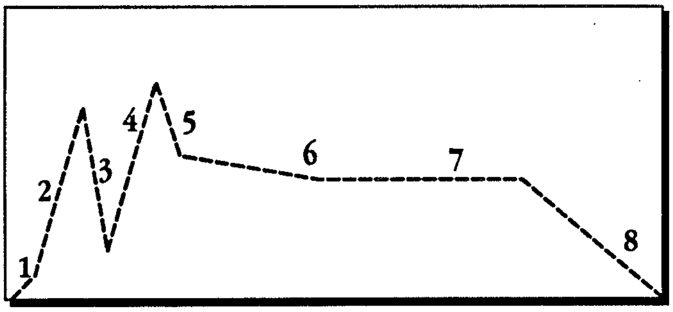
The release is the portion of a note as it dies away. Long releases can produce a nice ringing quality, but can also be a problem if a note is still sounding when another note starts and is dissonant with the first note.

Note Synthesizer envelopes

The envelope definition in the Note Synthesizer's instrument record is somewhat more complex than this simple four-part scheme. The instrument's envelope field can specify up to eight segments instead of just four, so more complex sequences of attack, decay, sustain, and release are possible. For example, the physical properties of pianos cause them to have a complex envelope with two attack segments. A simple ADSR is therefore limited in its ability to simulate a piano's envelope. The Note Synthesizer can do better, because its eight envelope segments allow a closer approximation of the piano's actual envelope.

Figure 41-2 shows an envelope created with eight envelope segments.

- **Figure 41-2** Typical Note Synthesizer envelope



An instrument's envelope definition is composed of up to eight linear segments. During each segment, the note's loudness slopes from its starting value toward its defined breakpoint value. The segments are defined as a series of breakpoints and increments.

The breakpoint respresents the loudness of the sound in a byte value between 0 and 127 on a logarithmic loudness scale. A value difference of 16 represents a change of 6 decibels in loudness.

The increment determines the amount of time to be spent reaching the breakpoint volume. The value is a 2-byte fixed-point number which indicates the amount by which the current volume is to be adjusted at each update (the default rate is 100 updates per second; you can set other values with the `NSStartUp` and `NSSetUpdateRate` tool calls). The low-order byte contains the numerator for a fractional increment. For example, an increment value of 1 translates to a fractional increment of $\frac{1}{256}$. In this case, the volume will be incremented once every 256 interrupts. The Note Synthesizer will process the segment until its volume reaches the specified breakpoint value, at which time it moves to the next segment.

The shape of the envelope is arbitrary; it can be any shape that can be specified in eight segments, so complex envelopes are possible. The last breakpoint, though, should always be 0, so that the note dies away at the end. The volume should not go to 0 before the end of the segment or the sound is considered done.

The length of time that a segment of the envelope lasts is given by the following formula:

$$T = \frac{|(L-N)| \cdot 256}{(0.4) \cdot (I \cdot R)}$$

where

T = segment's duration
L = last breakpoint
N = next breakpoint
I = increment value
R = update rate

As an example, for a segment that changes from 30 to 40 with an increment value of 25 and an update rate of 100 cycles per second, the formula becomes

$$T = \frac{|(30-40)| \cdot 256}{(0.4) \cdot (25 \cdot 100)} = \frac{2560}{(0.4)2500} = 2.56 \text{ seconds}$$

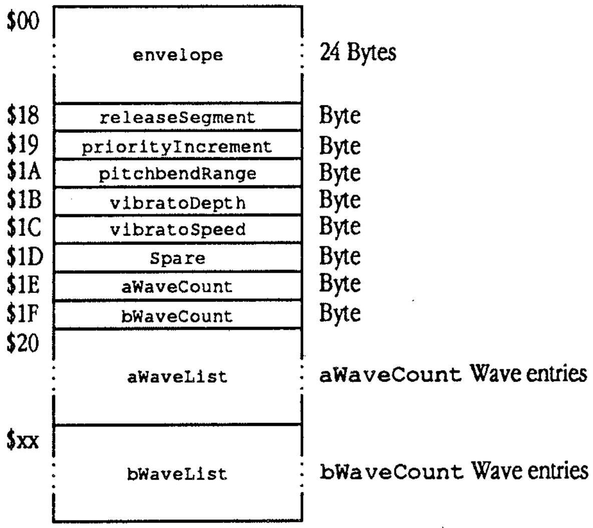
Thus, with the given parameters, the specified segment will last 2.56 seconds.

The increment value of a sustain segment is 0, so the previous formula cannot be used to calculate the duration of the sustain portion of an envelope. Instead, the sustain portion simply continues until a release is signaled. If the release portion of the note sustains, then the note will continue to be played until there are no available generators left, and the generator producing the note is reallocated to another note.

Instruments

The Note Synthesizer's basic functional unit is an instrument. This is a data structure stored somewhere in the memory of the Apple IIGS that defines the sound characteristics of a played note. When a program makes the `NoteOn` call, it passes a pointer to an instrument, and that instrument is used while generating the sound. Figure 41-3 shows the format of the instrument data structure.

- **Figure 41-3** Instrument data structure



$00	`envelope`	24 Bytes
$18	`releaseSegment`	Byte
$19	`priorityIncrement`	Byte
$1A	`pitchbendRange`	Byte
$1B	`vibratoDepth`	Byte
$1C	`vibratoSpeed`	Byte
$1D	`Spare`	Byte
$1E	`aWaveCount`	Byte
$1F	`bWaveCount`	Byte
$20	`aWaveList`	`aWaveCount` Wave entries
$xx	`bWaveList`	`bWaveCount` Wave entries

`envelope` Specifies the envelope for the sound, as a series of 8 segments, each a breakpoint and increment value pair (see "Note Synthesizer envelopes" in this chapter for detailed information on these concepts). Each breakpoint is a one-byte value specifying a target volume level in the range from 0 through 127. Each increment is a two-byte value which determines the amount of time the Note Synthesizer will spend reaching the breakpoint volume.

The `envelope` array has the following format:

$00	`breakpoint0`	Byte—Breakpoint value for segment 0
$01	`increment0`	Word—Increment value for segment 0
$03	`breakpoint1`	Byte—Breakpoint value for segment 1
$04	`increment1`	Word—Increment value for segment 1
	⋮	
$15	`breakpoint7`	Byte—Breakpoint value for segment 7
$16	`increment7`	Word—Increment value for segment 7

`releaseSegment`
Defines the segment at which release begins when a `NoteOff` call is made. Its value can be any number from 0 to 7, and identifies which segment in sequence is the beginning of the release phase of the envelope. The release portion may thus occupy several segments, but the last breakpoint should always be 0. For example, if `releaseSegment` is set to 5 and `breakpoint7` has a value of 0, the Note Synthesizer ramps through segments 5, 6, and 7 before shutting down.

`priorityIncrement`

Subtracted from the generator's priority value when the envelope reaches its sustain phase. The Note Synthesizer uses the changing priority values to reallocate generators, giving higher priority to notes that are just starting. When an envelope reaches the release portion, the priority value assigned to its generator is again reduced, this time to half its current value. Thus, the higher priorities go to notes that are just starting; notes being sustained are accorded lower priority, and notes in their release phase receive lowest priority. This is just a rule of thumb; the actual priority values depend on the priority that was specified when the generator was allocated. For more information on generator priorities, see "Generators" in this chapter.

`pitchbendRange`

Specifies the maximum pitch bend that is possible on the note. The maximum possible value for a pitch bend is 127; `pitchbendRange` specifies by how many semitones the pitch is raised with the pitch bend value of 127. The legal values are 1, 2, and 4 semitones. Note that the only way to change the pitchbend value for a note that is playing is to change the `pitchbend` field of the appropriate Generator Control Block (GCB) (see "Generators" in this chapter for information on the format and content of the GCB).

The `pitchbend` field is mainly used by the Note Sequencer. It is possible to set its value directly, but it is normally used by the Note Sequencer to pass information to the Note Synthesizer about how to play notes in a sequence.

`vibratoDepth` Any number from 0 to 127. A depth of 0 specifies that there is no vibrato effect on the note. Vibrato is produced by modulating the pitch of the two oscillators that make up a generator, using a triangle wave produced by a Low Frequency Oscillator (LFO). When the `vibratoDepth` parameter specifies that there is to be no vibrato effect, the vibrato software is switched off to save processing time, because the processing required to create the triangle wave can consume a large amount of processor time.

`vibratoSpeed` Controls the rate of vibrato. Higher values produce faster vibrato. The actual speed of vibrato effect depends on the update rate, which defaults to 100 updates per second. You can set other rates with the `NSStartUp` and `NSSetUpdateRate` tool calls.

`aWaveCount`, `bWaveCount`

Specify the number of waves in the wavelists that follow the wavecounts. The Note Synthesizer can use sampled or artificially created waveforms to produce its notes.

`aWaveList`, `bWaveList`

A wavelist is an array of variable length. The elements of the array are 6-byte structures A wavelist can contain up to 255 entries.

An entry in a wavelist data structure specifies wave data that is intelligible for the DOC. The Note Synthesizer places the data into the DOC registers.

$00	`topKey`	Byte
$01	`waveAddress`	Byte
$02	`waveSize`	Byte
$03	`DOCMode`	Byte
$04	`relPitch`	Word

`topKey`

When the Note Synthesizer plays a note, it examines the `topkey` field of each waveform in the wavelists until it finds a value that is greater than or equal to the value of the note it is attempting to play. The first waveform it finds with an acceptable `topkey` value is the one it plays. For this reason, waveforms should be stored in increasing order of `topkey` value. The last waveform in a wavelist should have a value of 127, the maximum valid pitch value.

`waveAddress`

The `waveAddress` parameter is the high byte of the waveform's address in Sound RAM. Its value is copied into the Address Pointer register of the DOC.

`waveSize`

The `waveSize` parameter sets the size of the DOC's wave table, and the frequency resolution of the DOC. This parameter is copied directly to the DOC's Bank-Select/TableSize/Resolution register. The resolution and table size should normally be equal. See Chapter 47, "Sound Tool Set Update," in this book and the *Apple IIGS Hardware Reference* for more information.

`DOCMode` The `DOCMode` parameter sets the mode of the Digital Oscillator Chip. This parameter corresponds to the Control register of the DOC, and supplies the stereo position of the oscillator. Bit 3 of this register (the interrupt enable bit for the DOC) should always be set to 0. See Chapter 47, "Sound Tool Set Update," in this book and the *Apple IIGS Hardware Reference* for more information.

`relPitch` The `relPitch` parameter is a word value that is used to tune the waveform. The high-byte value is the semitone, and the low byte is fractions of semitones. A value of 1 in the low byte corresponds to 1/256 of a semitone. A wavelist can specify a full range of notes for an instrument with entries for each note that differ only in the `relPitch` field. Such a wavelist would specify an instrument whose timbre is the same for every note; only the pitch is different.

For more information on DOC registers and waveforms, see Chapter 47, "Sound Tool Set Update," in this book and the *Apple IIGS Hardware Reference.*

DOC memory

One page of bank zero memory must be allocated to the Sound Tool Set for use as a direct page. The Note Synthesizer shares this direct-page space with the Sound Tool Set.

An application that uses the Note Synthesizer must load any waveforms that it can use into DOC memory with the Sound Tool Set call `WriteRAMBlock`. You must not place a 0 in the first 256 bytes of DOC memory because it halts the timer oscillator; this will cause a system failure. If the application uses the clock function of the MIDI tools, then it must not write to the first 256 bytes of DOC memory.

Generators

Each generator is a pair of DOC oscillators. There are 32 such oscillators; two of them are reserved for the use of Apple Computer, Inc. The remaining 30 are paired into 15 generators for the Note Synthesizer. The Note Synthesizer uses one of these generators as a timer, leaving 14 generators for general use. If the MIDI Tool Set is started up and is using the MIDI clock function, another generator is allocated to serve as the MIDI clock, leaving 13 general-purpose generators for application use.

The Note Synthesizer allocates generators to all the different sound tools that may need them. It therefore requires a priority scheme for allocating generators in the event that a generator is requested when all generators are in use. When a generator is allocated, it receives a priority. A generator's priority may range from 0 through 128. A priority of 0 means the generator is not being used and will be allocated to any use that requests it. A priority of 128 indicates that the generator is locked and cannot be reallocated. The remaining values in a generator's range are used by the Note Synthesizer to control allocation of generators.

The Note Synthesizer automatically lowers the priority of a generator that has reached the sustain portion of its envelope, and lowers it again when it reaches the release portion. When the note stops, the generator's priority becomes 0. An application specifies a priority when requesting allocation of a generator, so that allocation occurs when a generator is available with a priority lower than that requested.

The Note Synthesizer divides its direct-page area into 15 blocks of 16 bytes, called generator control blocks (GCB). The GCB contains the values of any "knobs" or "controllers" affecting the parameters of the note that it is currently playing. A programmer normally should not access the GCB.

Figure 41-4 shows the format and content of the GCB.

- **Figure 41-4** Generator control block

Offset	Field	Description
$00	synthID	Byte—Identifies user of generator
$01	genNum	Byte—Identifies the generator itself
$02	semitone	Byte—Note currently being played by the generator
$03	volume	Byte—Output volume for current note
$04	pitchbend	Byte—Pitchbend value for current note
$05	vibratoDepth	Byte—Vibrato for current note
$06	Reserved	10 Bytes—Note Synthesizer and Sound Tools reserved

`synthID` Identifies who is currently using the generator. Valid values are

0	Not used
1	Sound Tool Set freeform synthesizer
2	Note Synthesizer
3	Reserved for use by Apple Computer, Inc.
4	MIDI Tool Set
5–7	Reserved for use by Apple Computer, Inc.
8–15	User defined

`genNum` Uniquely identifies the generator. Valid values lie in the range from 0 through 13 ($00 through $0D). Your application uses this value to identify a specific generator to the Note Synthesizer. The tool set returns the identifier on the `AllocGen` call.

`semitone` Identifies the note currently being played. Contains a standard MIDI value in the range from 0 to 127, where middle C has a value of 60.

`volume` Identifies the output volume for the current note specified by `semitone`. Valid values lie in the range from 0 through 127, and correspond to MIDI velocity. A 16-step change in `volume` corresponds to a 6 decibel change in amplitude.

`pitchbend` Identifies pitch bend to be applied to the note specified by `semitone`. Valid values lie in the range from 0 through 127; a value of 64 specifies no pitch bend. The `pitchbendRange` field of the instrument record (see "Instruments" earlier in this chapter) specifies the maximum allowable pitch bend, in semitones.

`vibratoDepth` Specifies the depth of vibrato for the note. Valid values lie in the range from 0 through 127. A value of 0 indicates no vibrato (this is the recommended value). A value of 127 yields maximum vibrato depth.

Reserved Area reserved for internal use by the Note Synthesizer and the Sound Tool Set.

Note Synthesizer housekeeping calls

All the call descriptions for the Note Synthesizer are new. The tool calls were not previously documented in the *Apple IIGS Toolbox Reference.*

`NSBootInit` $0119

Initializes the Note Synthesizer.

▲ **Warning** An application must not make this call. ▲

Parameters This call has no input or output parameters. The stack is unaffected.

Errors None

C

```
extern pascal void NSBootInit();
```

`NSStartUp` $0219

Starts up the Note Synthesizer for use by an application. An application must make this call before it makes any other Note Synthesizer calls except `NSStatus` or `NSVersion`. The *updateRate* parameter specifies the rate at which interrupts are generated to update envelopes and low-frequency oscillations. The value is in units of 0.4 Hz. Reasonable values for this parameter include 150, 250, and 500. The default value is 500. Low rates require less overhead, but higher rates generate smoother sounding envelopes.

The *userUpdateRtn* parameter is a pointer to a routine that is called during every timer interrupt. Sequencer programs are an example of software that might use routines that run during Note Synthesizer interrupts, and, in fact, this is how the Note Sequencer works. A value of 0 indicates that there is no user update routine.

Parameters

Stack before call

Previous contents	
updateRate	Word—Rate of envelope generation
– *userUpdateRtn* –	Long—Pointer to custom interrupt routine
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$1901	`nsAlreadyInit`	Note Synthesizer already started up.
	$1902	`nsSndNotInit`	Sound Tool Set not started up.
	$1925	`soundWrongVer`	Incompatible version of Sound Tool Set.

C

```
extern pascal void NSStartUp(updateRate,
          userUpdateRtn);

Word      updateRate;
Pointer   userUpdateRtn;
```

`NSShutDown` $0319

Shuts down the Note Synthesizer and turns off all generators. An application should make this call before quitting.

Parameters This call has no input or output parameters. The stack is unaffected.

Errors	$1923	`nsNotInit`	Note Synthesizer not started up.

C `extern pascal void NSShutDown();`

`NSVersion` $0419

Returns the version number of the Note Synthesizer. Refer to Appendix A, "Writing Your Own Tool Set," in Volume 2 of the *Toolbox Reference* for information about the format and content of the *versionNum* return value.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
	<—SP

Stack after call

Previous contents	
versionNum	Word—Note Synthesizer version number
	<—SP

Errors None

C `extern pascal Word NSVersion();`

`NSReset` $0519

Resets the Note Synthesizer.

▲ **Warning** An application must not make this call. ▲

Parameters This call has no input or output parameters. The stack is unaffected.

Errors None

C

```
extern pascal void NSReset();
```

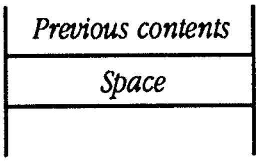
`NSStatus` $0619

Returns a Boolean value indicating whether the Note Synthesizer is active. If the Note Synthesizer is active, `NSStatus` returns TRUE. Otherwise, the call returns FALSE.

◆ *Note:* If your program issues this call in assembly language, initialize the result space on the stack to NIL. Upon return from `NSStatus`, your program need only check the value of the returned flag. If the Note Synthesizer is not active, the returned value will be FALSE (NIL).

Parameters

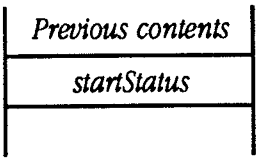
Stack before call



Word—Space for result

<—SP

Stack after call



Word—Boolean; TRUE if the Note Synthesizer is started

<—SP

Errors None

C `extern pascal Boolean NSStatus();`

Note Synthesizer calls

The following sections discuss the Note Synthesizer tool calls.

`AllNotesOff` $0D19

Turns off all Note Synthesizer generators and sets their priorities to 0. It does not affect generators not used by the Note Synthesizer, such as those allocated to the freeform synthesizer.

Parameters This call has no input or output parameters. The stack is unaffected.

Errors None

C

```
extern pascal void AllNotesOff();
```

`AllocGen` $0919

Requests the allocation of a sound generator. Returns a generator number from 0 to 13. The call will reallocate a generator if all generators are allocated and the specified *requestPriority* exceeds that of one of the previously allocated generators.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
requestPriority	Word—Desired generator priority
	<—SP

Stack after call

Previous contents	
genNum	Word—Number of allocated generator
	<—SP

Errors	$1921	`nsNotAvail`	No generators available to allocate.
	$1923	`nsNotInit`	Note Synthesizer not started up.

C

```
extern pascal Word AllocGen(requestPriority);

Word      requestPriority;
```

DeAllocGen $0A19

Sets the named generator's allocation priority to zero and halts its oscillators. Any subsequent allocation request with a valid *requestPriority* will then succeed.

Parameters

Stack before call

Previous contents	
genNum	Word—Number of generator to deallocate
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$1922	`nsBadGenNum`	Invalid generator number.

C

```
extern pascal void DeAllocGen(genNum);

Word      genNum;
```

`NoteOff` $0C19

Switches the specified generator to release mode, which causes the note being generated to die out. When the note's volume is 0, the generator's priority is set to 0, and it is considered to be off. The *genNum* and *semitone* parameters should be set to the same values specified in the corresponding `NoteOn` call.

Parameters

Stack before call

Previous contents	
genNum	Word—Generator number
semitone	Word—Note being played
	<—SP

Stack after call

Previous contents	
	<—SP

Errors None

C

```
extern pascal void NoteOff(genNum, semitone);

Word      genNum, semitone;
```

`NoteOn` $0B19

Initiates the generation of a note on a specified generator. The *genNum* parameter should be a value returned by the `AllocGen` call. The *semitone* parameter is a standard MIDI value from 0 to 127, where middle C is designated by the value 60. The *volume* parameter is a value from 0 to 127 that can be treated as synonymous with MIDI velocity. The value is copied into the generator control block, and is used to scale the note's amplitude. A change of 16 steps in this parameter specifies a change of 6 decibels in amplitude. The *instrumentPtr* parameter is a pointer to an instrument. See "Instruments" earlier in this chapter for more information on the instrument data structure.

◆ *Note:* Experiment with the *volume* parameter and envelope amplitudes; if the sum of these two values is too small, the note being played will be inaudible even if everything else is working correctly. The dynamic range of the DOC is 48 decibels.

Parameters

Stack before call

Previous contents	
genNum	Word—Generator number
semitone	Word—Desired pitch for note
volume	Word—Desired volume for note
– *instrumentPtr* –	Long—Pointer to instrument to play note
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$1924	`nsGenAlreadyOn`	The specified note is already being played.

C

```
extern pascal void NoteOn(genNum, semitone, volume,
          instrumentPtr);

Word      genNum, semitone, volume;
Pointer   instrumentPtr;
```

Example

The following example shows assembly-language code that allocates a generator, passes the correct parameters to NoteOn, plays a note, and turns off the note.

```
            pushword #0             ;space for GenNum
            pushword #64            ;priority of this note
            _AllocGen               ;retrieve an allocated generator
            pla                     ;get the generator number
            sta GenNum              ;store it

            pushword GenNum         ;push parameters:generator
            pushword Semitone       ;note
            pushword #127           ;maximum volume
            pushlong #Instrument    ;LONG pointer to instrument
definition
            _NoteOn
```

After some time...

```
            pushword GenNum         ;push parameters: generator
            pushword Semitone       ;note
            _NoteOff                ;turn off the note
```

`NSSetUpdateRate` $0E19

Sets the Note Synthesizer's *updateRate* parameter, as described in the `NSStartUp` section. The specified *updateRate* value becomes the new *updateRate*, and the old value is returned.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
updateRate	Word—New update rate
	<—SP

Stack after call

Previous contents	
oldRate	Word—Update rate before call
	<—SP

Errors $1923 `nsNotInit` Note Synthesizer not started up.

C

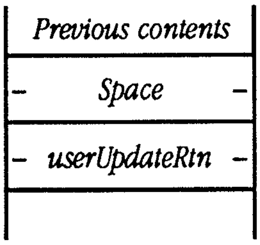
```
extern pascal Word NSSetUpdateRate(updateRate);

Word      updateRate;
```

`NSSetUserUpdateRtn` $0F19

Sets the user update routine described in the `NSStartUp` section. The update routine pointer is set to the value passed in the *userUpdateRtn* parameter, and the address of the old update routine is returned. If there is no user update routine when this call is made, it returns a NULL pointer. A NULL *userUpdateRtn* value disables the current update routine.

Parameters

Stack before call

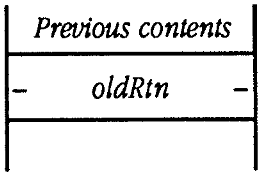


Long—Space for result

Long—Pointer to new update routine

<—SP

Stack after call



Long—Pointer to old update routine

<—SP

Errors	$1923	`nsNotInit`	Note Synthesizer not started up.

C

```
extern pascal VoidProcPtr
          NSSetUserUpdateRtn(userUpdateRtn);

Pointer   userUpdateRtn;
```

Note Synthesizer error codes

This section lists the error codes that may be returned by Note Synthesizer calls.

Value	Name	Definition
$1901	`nsAlreadyInit`	Note Synthesizer already started up.
$1902	`nsSndNotInit`	Sound Tool Set not started up.
$1921	`nsNotAvail`	No generators available to allocate.
$1922	`nsBadGenNum`	Invalid generator number.
$1923	`nsNotInit`	Note Synthesizer not started up.
$1924	`nsGenAlreadyOn`	The specified note is already being played.
$1925	`soundWrongVer`	Incompatible version of Sound Tool Set.

Chapter 42 **Print Manager Update**

This chapter documents new features of the Print Manager. The complete reference to the Print Manager is in Volume 1, Chapter 15 of the *Apple IIGS Toolbox Reference.*

Error corrections

This section documents errors in the *Toolbox Reference.*

- The diagram for the job subrecord, figure 15-10 on page 15-14 of the *Toolbox Reference,* shows that the `fFromUsr` field is a word. This is incorrect. The `fFromUsr` field is actually a byte. Note that the offsets for all fields following this one are incorrect, as a result. This error is also reflected in the tool set summary at the end of the chapter.
- The description of the `PrJobDialog` tool call states that "The initial settings displayed in the dialog box are taken from the printer driver. . . ." This is incorrect. The sentence should begin "The initial settings displayed in the dialog box are taken from the print record. . . ."

Clarifications

- The existing *Toolbox Reference* documentation for the `PrPicFile` tool call does not mention that your program may pass a NIL value for *statusRecPtr.* Passing a NIL pointer causes the system to allocate and manage the status record internally.
- The `PrPixelMap` call (documented in Volume 1 of the *Toolbox Reference)* provides an easy way to print a bitmap. It does much of the required processing, and an application need not make the calls normally required to start and end the print loop. The *srcLocPtr* parameter must be a pointer to a *locInfo* record (see Figure 16-3 in Chapter 16, "QuickDraw II," in the *Toolbox Reference* for the layout of the *locInfo* record).

New features in the Print Manager

The following functions have been added to the Print Manager:

- The port driver auxiliary file type of an AppleTalk driver is $0003. Its file type remains $BB.
- The PRINTER.SETUP file now saves separate settings for direct and network connections to printers. Old versions of the PRINTER.SETUP file are incompatible with these changes, so the Print Manager will delete such files and create a new one in the correct format. Old settings are discarded, and the default settings are used to create the new setup file.
- If the Print Manager attempts to load a driver and finds that it is missing, it will pass control to a routine that determines what call was being made to the driver, pops the parameters off the stack, and returns a `missingDriver` error ($1301). It will also display an alert asking the user to make sure a printer and port driver are selected, if `PrJobDialog` and `PrStlDialog` are called.
- The `PMStartup` call does not load any drivers into memory. Drivers are loaded only when they are needed. The Print Manager does not require that the DRIVERS folder be present, and if it is present, does not require that there be any drivers in it.
- The `PrChoosePrinter` call is no longer supported. Users should now use the Control Panel desk accessory to choose new printers. When an application issues the `prChoosePrinter` call, the Print Manager will display an alert directing the user to use the Control Panel. New applications should never issue this call and should not include the Choose Printer item in the file menu. Note that `PMStartup` will still load the List Manager if it has not already been loaded.
- Print Manager now allows you to assign a name to a document. This feature is primarily applicable to documents destined for AppleTalk printers.
- Currently, if a user wants to print multiple copies of a document in draft mode to an ImageWriter®, ImageWriter LQ, or Epson printer, your application must run through its print loop once for each copy. The draft mode flag and copy count fields are located in the Job subrecord of the print record.
- The LaserWriter driver will now use some PostScript® fonts that have been downloaded into the printer by another computer.

New Print Manager calls

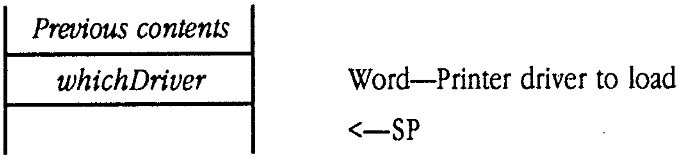
The following sections discuss new Print Manager tool calls.

`PMLoadDriver` $3513

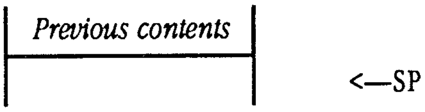
Loads the current printer driver, port driver, or both, depending on the input parameter. The current driver is determined by the settings saved in the PRINTER.SETUP file.

Parameters

Stack before call



Stack after call



Errors	$1309	`badParam`	The specified parameter is invalid.

C

```
extern pascal void PMLoadDriver(whichDriver);

Word      whichDriver;
```

whichDriver Specifies which printer driver to load. Legal values for the driver parameter include

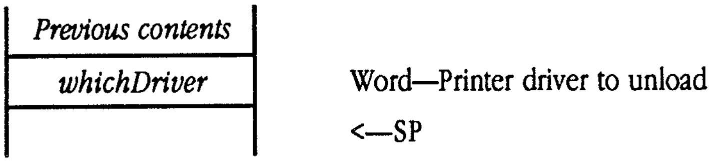
0 Load both drivers.
1 Load printer driver.
2 Load port driver.

`PMUnloadDriver` $3413

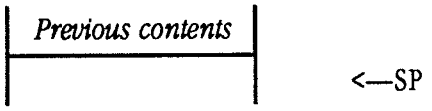
Unloads the current port driver, printer driver, or both, depending on the input parameter.

Parameters

Stack before call



Stack after call



Errors	$1309	`badParam`	The specified parameter is invalid.

C

```
extern pascal void PMUnloadDriver(whichDriver);

Word        whichDriver;
```

whichDriver Specifies which printer driver to unload. Legal values for the driver parameter include

0 Unload both drivers.
1 Unload printer driver.
2 Unload port driver.

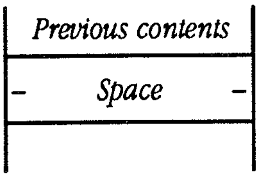
`PrGetDocName` $3613

Returns a pointer to the current document name string for your document. Use the `PrSetDocName` tool call to set or change the document name.

Note that there is only one active document name for the system at any given time. Your application must correctly manage this name in the context of the document being printed.

Parameters

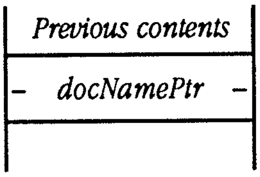
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to document name string (Pascal string)

<—SP

Errors None

C `extern pascal Pointer PrGetDocName();`

`PrGetPgOrientation` $3813

Returns a value indicating the current page orientation for the specified document.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
– prRecordHandle –	Long—Handle to print record for document
	<—SP

Stack after call

Previous contents	
orientation	Word—Page orientation: 0=portrait, 1=landscape
	<—SP

Errors None

C

```
extern pascal Word
            PrGetPgOrientation(prRecordHandle);

Handle      prRecordHandle;
```

`PrGetPrinterSpecs` $1813

Returns information about the currently selected printer.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
Space	Word—Space for result
	<—SP

Stack after call

Previous contents	
characteristics	Word—Word defining printer characteristics
printerType	Word—Word indicating the type of printer connected
	<—SP

Errors None

C `extern pascal PrinterSpecs PrGetPgOrientation();`

characteristics Defines the features of the particular printer:

Reserved	bits 2–15	Must be set to 0
`color`	bits 0–1	Indicates color capability: 00 - Can't determine 01 - Black and white only 10 - Color 11 - Invalid value

printerType Indicates the type of printer selected:

0	undefined
1	ImageWriter I or II
2	ImageWriter LQ
3	LaserWriter® family (LaserWriter, Plus, and II)
4	Epson

`PrSetDocName` $3713

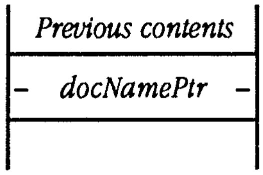
Sets the document name for use with AppleTalk printers. Print Manager passes this name when connecting to printers and spoolers, so that the destination printer may properly report the name.

Note that there is only one active document name for the system at any given time. Your application must correctly manage this name in the context of the document being printed.

In some status windows, the document name may be truncated. To avoid name truncation, you should use names containing fewer than 32 characters.

Parameters

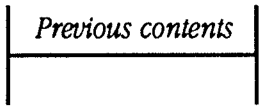
Stack before call



Long—Pointer to document name string (Pascal string)

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void PrSetDocName(docNamePtr);

Pointer   docNamePtr;
```

Previously undocumented Print Manager calls

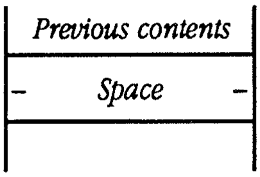
The following calls have not previously been documented, but may be useful to application programmers.

`PrGetNetworkName` $2B13

Returns the AppleTalk network name for the currently selected printer. If the user has selected a non-networked printer, the call returns a NIL pointer.

Parameters

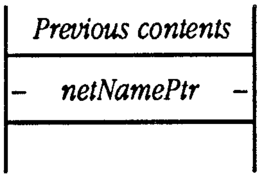
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to printer network name string (Pascal string)

<—SP

Errors None

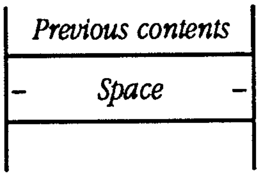
C `extern pascal Pointer PrGetNetworkName();`

`PrGetPortDvrName` $2913

Returns the name string for the currently selected port driver.

Parameters

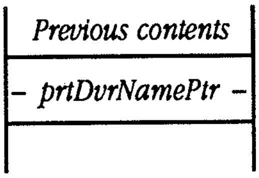
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to port driver name string (Pascal string)

<—SP

Errors None

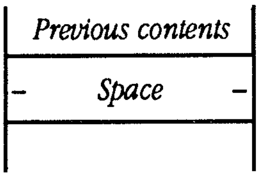
C `extern pascal Pointer PrGetPortDvrName();`

`PrGetPrinterDvrName` $2813

Returns the name string for the currently selected printer driver.

Parameters

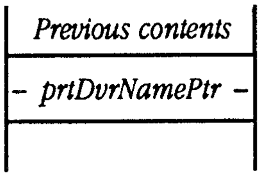
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to printer driver name string (Pascal string)

<—SP

Errors None

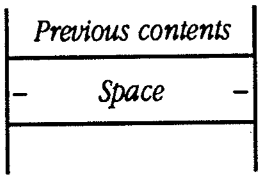
C `extern pascal Pointer PrGetPrinterDvrName();`

`PrGetUserName` $2A13

Returns the user name as entered in the Control Panel.

Parameters

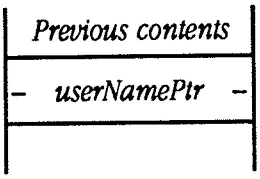
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to user name string (Pascal string)

<—SP

Errors None

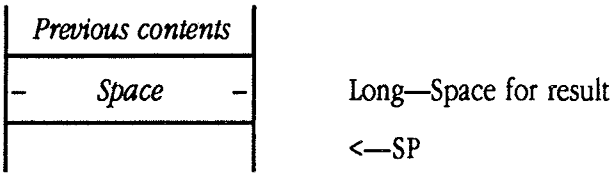
C `extern pascal Pointer PrGetUserName();`

`PrGetZoneName` $2513

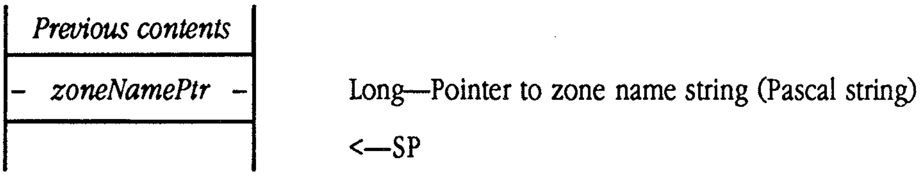
Returns the name string for the currently selected AppleTalk print zone. If the user has selected a non-networked printer, then the call returns a NIL pointer.

Parameters

Stack before call



Stack after call



Errors None

C `extern pascal Pointer PrGetZoneName();`

Print Manager error codes

This section lists all valid Print Manager error codes.

Value	Name	Definition
$1301	`missingDriver`	Specified driver not in the DRIVERS subdirectory of the SYSTEM subdirectory.
$1302	`portNotOn`	Specified port not selected in the control panel.
$1303	`noPrintRecord`	No print record specified
$1306	`papConnNotOpen`	Connection cannot be established with the LaserWriter.
$1307	`papReadWriteErr`	Read-write error on the LaserWriter.
$1308	`pntrConFailed`	Cannot establish connection with ImageWriter.
$1309	`badParam`	Invalid parameter for load or unload.
$130A	`callNotSupported`	Tool call is not supported by current version of the driver.
$1321	`startUpAlreadyMade`	`LLDStartUp` call already made.

Chapter 43 **QuickDraw II Update**

This chapter documents new features of QuickDraw™ II. The complete reference to QuickDraw II is in Volume 2, Chapter 16 of the *Apple IIGS Toolbox Reference.*

Error corrections

The following items provide corrections to the documentation for QuickDraw II in the *Apple IIGS Toolbox Reference:*

- The documentation in the *Toolbox Reference* that explains pen modes is somewhat misleading. There are, in fact, 8 drawing modes, and you may set the pen to draw lines and other elements of graphics in any of these modes. There are also 16 modes used for drawing text, and they are completely independent of the graphic pen modes. The 8 drawing modes listed in Table 16-9 on page 16-235 are valid modes for either the text pen or the graphics pen. You can set either pen to any of these modes by using the appropriate calls. You can also set the text pen to 8 other modes. These modes are listed in the table on page 16-260 of the *Toolbox Reference*. The `SetPenMode` call sets the mode used by the graphics pen; the `SetTextMode` call sets the mode used by the text pen. Setting either one does not affect the other.
- There are two versions of the Apple IIGS standard 640-mode color tables, one on page 16-36 and one on page 16-159. The two tables are different; Table 16-7 on page 16-159 is correct.
- In the QuickDraw II chapter, the *Apple IIGS Toolbox Reference* states that the coordinates passed to the `LineTo` and `MoveTo` calls should be expressed as global coordinates. In fact, the coordinates must be local coordinates, and must refer to the GrafPort in which the drawing or moving takes place.

New features in QuickDraw II

The following information describes new features in this version of QuickDraw II.

- QuickDraw II now supports 16 bit by 8 bit pixel patterns in 640 mode. To use these larger patterns, set the high-order bit (bit 15) of the `arcRot` word in the GrafPort record to 1. QuickDraw II will then use all 32 bytes of the passed pattern. Since the `OpenPort` and `InitPort` tool calls clear this bit, existing applications will work fine.

  `PointInRect` now works as previously documented.
- In the FONT folder on your system disk you will find a file named FASTFONT. This file contains a special version of the Shaston 8 font that will provide markedly improved performance for text drawing under many circumstances. Specifically, this font can be used whenever you are drawing plain, black text on a white background into a rectangularly clipped region. While this may sound overly restrictive, most applications draw text in precisely this way. This font reduces text drawing time by more than half.

  To use this font, QuickDraw II must find it in your FONT folder when the tool is started. If your application draws text to an off-screen bitmap, use `OpenPort` and `InitPort` to set up the off-screen buffers in order to insure that FASTFONT is properly installed.

QuickDraw II speed enhancement

In addition to FASTFONT, QuickDraw II has several other changes that improve drawing performance. First, pattern filling in `modeCopy` and `modeXOR` now operates between 2 and 4 times faster. The remaining changes require that you modify your application to take advantage of the performance improvements they offer.

QuickDraw II now supports hardware shadowing of screen images. This feature uses 32 KB of bank 1 memory to store the screen image. By storing the image in memory, QuickDraw II can offer an 8 to 20 percent speed improvement in all operations. You control whether QuickDraw II uses the shadow memory by setting a flag in the *masterSCB* parameter passed to the `QDStartUp` tool call. If QuickDraw II cannot allocate the needed memory, it will reset the flag and operate without shadowing in effect. Use the `GetMasterSCB` tool call to read the *masterSCB* back and check shadow status.

In addition, your application can further improve QuickDraw II performance by following some simple rules. First, your application must only change GrafPort fields via QuickDraw II tool calls, not by directly accessing the record fields. Next, for best results perform similar operations in groups. For example, if your application needs to erase and redraw four rectangles, it should do all the erases together, then all the draws. In this manner, QuickDraw II only has to change its drawing pattern twice, rather than eight times. Your application tells QuickDraw II that it will follow these fast port rules by setting a bit in the *masterSCB* passed to `QDStartUp`.

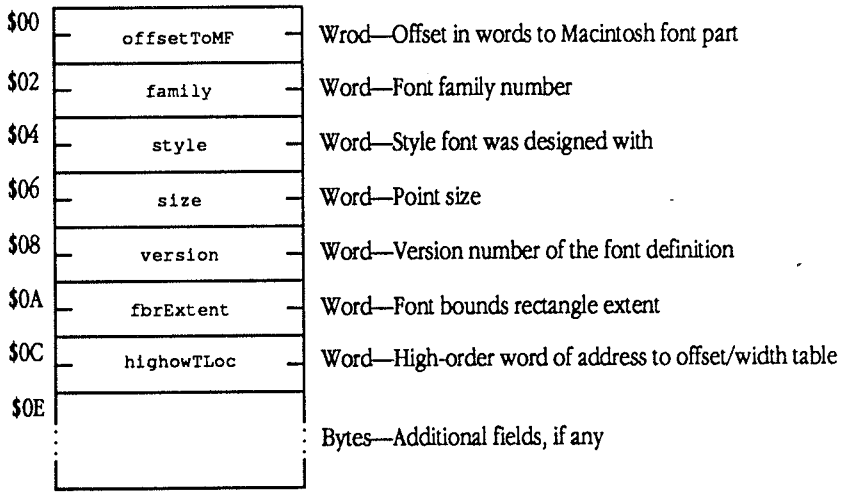
The *masterSCB* now has the following format:

`fUseShadowing`	bit 15	Controls use of hardware shadowing by QuickDraw II: 0 - No shadowing 1 - Shadowing
`fFastPortAware`	bit 14	Indicates whether application follows fast port rules: 0 - Does not use fast port rules 1 - Does use fast port rules
Reserved	bits 8–13	Must be set to 0
SCB	bits 0–7	Use standard SCB values

New font header layout

The font header has been expanded to include a new field containing additional addressing information. Figure 43-1 shows the new layout for the font header. For information about the old fields, see Chapter 16, "QuickDraw II," in Volume 2 of the *Toolbox Reference.*

■ **Figure 43-1** New font header layout



$00	`offsetToMF`	Wrod—Offset in words to Macintosh font part
$02	`family`	Word—Font family number
$04	`style`	Word—Style font was designed with
$06	`size`	Word—Point size
$08	`version`	Word—Version number of the font definition
$0A	`fbrExtent`	Word—Font bounds rectangle extent
$0C	`highowTLoc`	Word—High-order word of address to offset/width table
$0E		Bytes—Additional fields, if any

`highowTLoc` Defines the high-order word of the address to the offset/width table for the font. The `owTLoc` field contains the low-order word of the address. Together, these two fields form a full 32-bit address.

Chapter 44 **QuickDraw II Auxiliary Update**

This chapter documents new features in QuickDraw II Auxiliary. The complete reference to QuickDraw II Auxiliary is in Volume 2, Chapter 17 of the *Apple IIGS Toolbox Reference.*

New features in QuickDraw II Auxiliary

- QuickDraw II now supports text justification within pictures. Note that QuickDraw II justifies the text only when drawing the picture, not in the stored picture image. You control text justification in pictures by setting a bit flag in the `fontFlags` word of the GrafPort record. Use the `SetFontFlags` tool call to change the state of this bit.

  The `fontFlags` word is defined as follows:

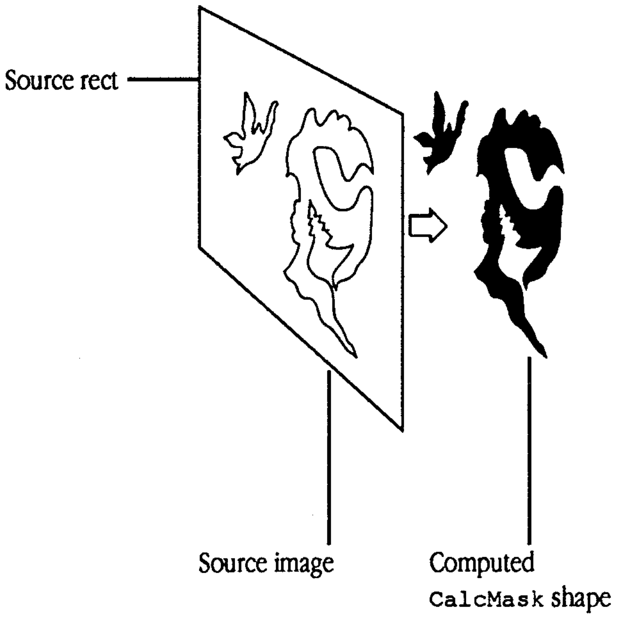
Reserved	bits 4–15	Must be set to 0
`fTextJust`	bit 3	Controls text justification in pictures: 0 - No text justification 1 - Justify text
	bits 0–2	Use standard `fontFlags` values (see page 16-56 in the *Toolbox Reference* for a description of these bits)

New QuickDraw II Auxiliary calls

Two new QuickDraw II tool calls, `CalcMask` and `SeedFill`, provide enhanced functionality to the application programmer who wants to create graphics entry or editing software. A third new call, `SpecialRect`, provides a high-performance rectangle frame and fill operation.

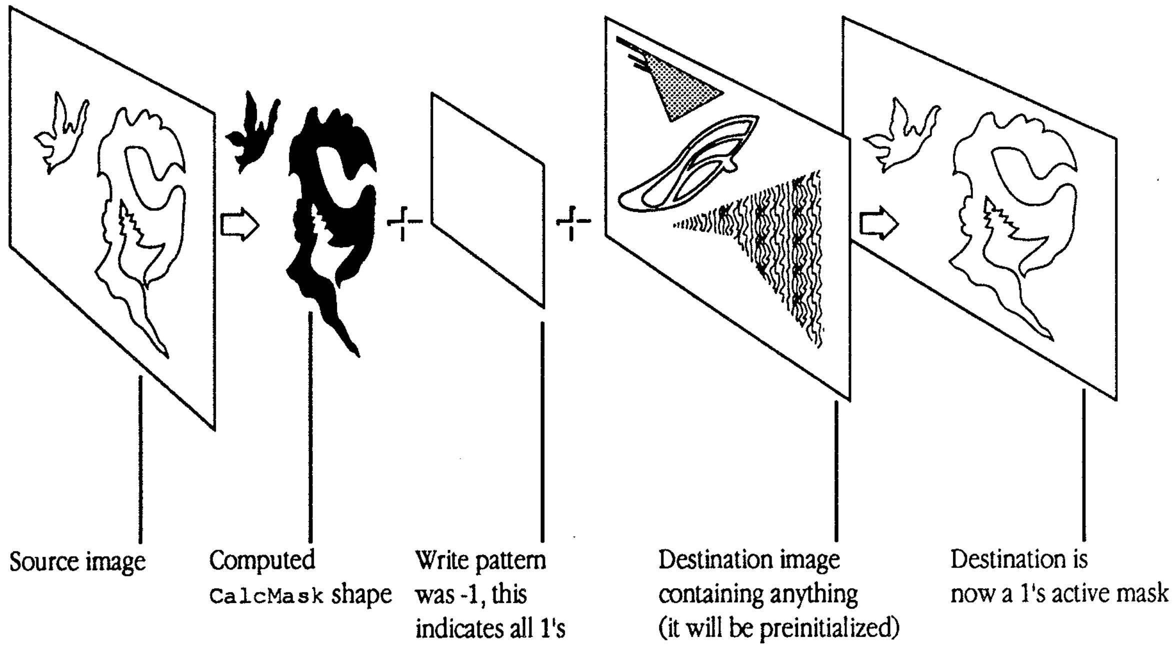
`CalcMask` $0E12

Generates a mask from a specified source image and pattern, by filling inward from the bounding rectangle. The shape of the resulting mask consists of all areas in the source image where leaking does not occur (all enclosed areas within the rectangle):



This call differs from `SeedFill` only in that it works from the "outside in", whereas `SeedFill` goes "inside out", filling all enclosed areas starting from a specified interior point (see "`SeedFill`" in this chapter for details).

`CalcMask` is most commonly used to implement a lasso tool, by having `CalcMask` determine the selected shape by filling inward from the lasso rectangle:


Source image
Computed CalcMask shape
Write pattern was -1, this indicates all 1's
Destination image containing anything (it will be preinitialized)
Destination is now a 1's active mask

For this use, set the call parameters as follows:

`destMode` portion of *resMode*	%0010 (clear destination to zeroes before drawing)
patternPtr	$FFFFFFFF (use all ones pattern when drawing to destination)

This call does not perform automatic scaling; therefore, the source and destination rectangles must be of equal size. In addition, note that the fill is not clipped to the current port, and that the resulting image cannot be stored into a QuickDraw II picture.

△ **Important** Your application must word-align both the source and destination rectangles in order to assure an accurate fill. △

Parameters

Stack before call

Previous contents	
– *srcLocInfoPtr* –	Long—Pointer to source `locInfo` data record
– *srcRect* –	Long—Pointer to source rectangle data record
– *destLocInfoPtr* –	Long—Pointer to destination `locInfo` data record
– *destRect* –	Long—Pointer to destination rectangle data record
resMode	Word—Resolution mode
– *patternPtr* –	Long—Pointer to fill pattern
– *leakTblPtr* –	Long—Pointer to leak-through color table
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$0201	`memErr`	NewHandle error occurred
$1211	`badRectSize`	1) Height or width is negative, 2) *destRect* not same size as *srcRect*, or 3) source or destination rectangle not within its boundary rectangle
$1212	`destModeError`	`destMode` portion of *resMode* invalid

C

```
extern pascal void CalcMask(srcLocInfoPtr, srcRect,
          destLocInfoPtr, destRect, resMode,
          patternPtr, leakTblPtr);

Pointer   srcLocInfoPtr, srcRect, destLocInfoPtr,
          destRect, patternPtr, leakTblPtr;
Word      resMode;
```

srcLocInfoPtr Points to a `locInfo` data record containing the definition for the source rectangle for the fill operation.

srcRect Points to a rectangle, in local coordinates, that contains the source pixel image.

destLocInfoPtr, destRect
Refer to output `locInfo` record and rectangle, respectively. These fields allow you to copy the output to a different location in a different rectangle. If you want the output of the operation to overlay the input image, set the source and destination pointers to the same values.

resMode Indicates the resolution mode for the fill as well as initialization and drawing options:

`destMode`	bits 12–15	Indicates initialization and drawing options: 0000 - copy source to destination (obliterating destination) 0001 - leave destination alone (overlay source onto destination) 0010 - initialize destination to zeroes before drawing 0011 - initialize destination to ones before drawing other values are invalid
Reserved	bits 2–11	Must be set to 0
`res`	bits 0–1	Indicates the resolution for the operation: 00 - 640 pure 01 - 640 dithered 10 - 320 mode 11 - invalid

patternPtr Pointer to the fill pattern for the operation, or flag specifying special fill pattern:

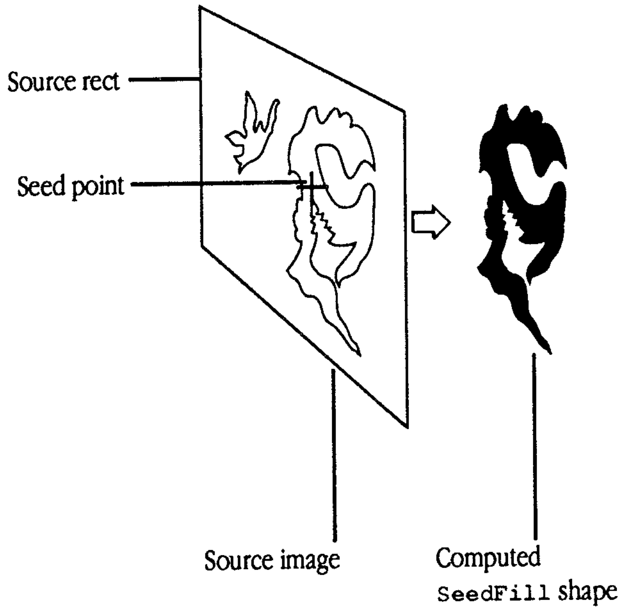
NIL	Use an all zeroes pattern when writing to destination
$FFFFFFFF	Use an all ones pattern when writing to destination
Other	Assumed to be valid pointer to fill pattern

leakTblPtr Pointer to a structure that defines the colors to be covered. The structure contains a count word, indicating the number of color entries in the table, and a color entry for each color to be leaked. Each color entry contains the offset into the color table for that color.

$00	`count`	Word—Count of color entries to follow
$02	`colorEntries`	`count` Words—Offset into color table for each color

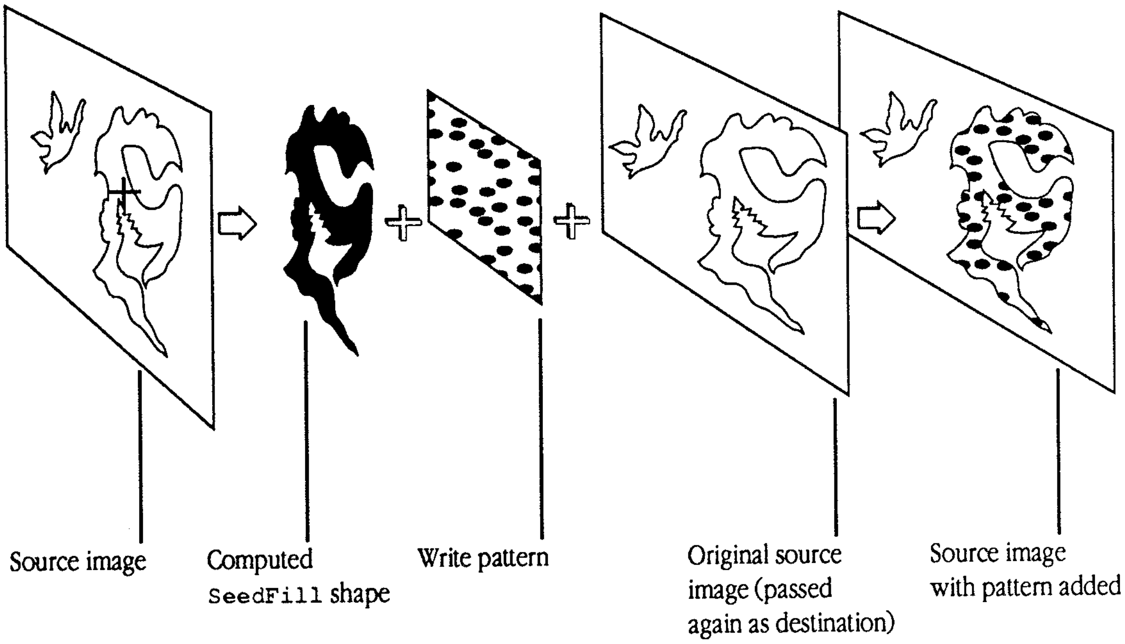
SeedFill $0D12

Generates a mask from a specified source image and pattern, by filling outward from a starting point within the source image. The shape of the resulting mask consists of the enclosed area in the source image surrounding the starting (or seed) point:



This call differs from CalcMask only in that it works from the "inside out", whereas CalcMask goes "outside in" (see "CalcMask" in this chapter for details).

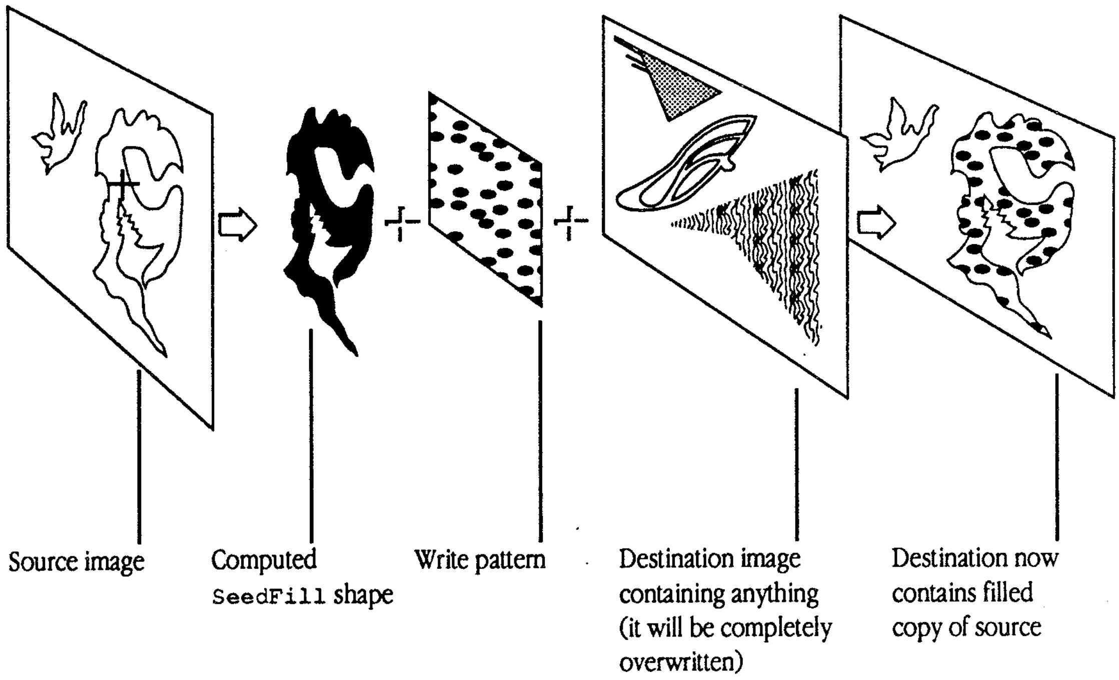
`SeedFill` is a versatile tool. Most simply, you can use it to implement a paint bucket tool:


Source image
Computed SeedFill shape
Write pattern
Original source image (passed again as destination)
Source image with pattern added

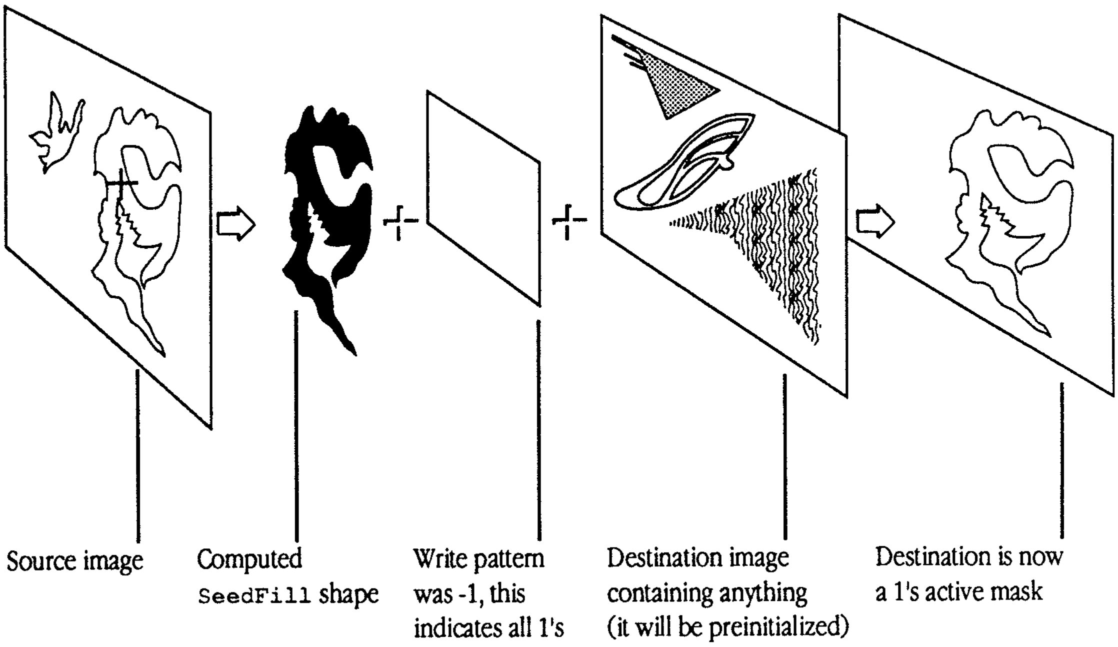
For this operation, use the following call parameter values:

`destMode` portion of *resMode*	%0001 (do not change destination image before drawing)
patternPtr	Pointer to fill color or pattern

In order to add an undo capability to the paint bucket, specify a different destination:


Source image
Computed `SeedFill` shape
Write pattern
Destination image containing anything (it will be completely overwritten)
Destination now contains filled copy of source

As a more complex example, consider this "from the inside" lasso tool:


Source image
Computed SeedFill shape
Write pattern was -1, this indicates all 1's
Destination image containing anything (it will be preinitialized)
Destination is now a 1's active mask

For this operation, use the following call parameter values:

`destMode` portion of *resMode*	%0010 (clear destination to zeroes before drawing)
patternPtr	$FFFFFFFF (use all ones pattern when drawing to destination)

This call does not perform automatic scaling; therefore, the source and destination rectangles must be of equal size. In addition, note that the fill is not clipped to the current port, and that the resulting image cannot be stored into a QuickDraw II picture.

△ **Important** Your application must word-align both the source and destination rectangles in order to assure an accurate fill. △

Parameters

Stack before call

Previous contents	
– *srcLocInfoPtr* –	Long—Pointer to source `locInfo` data record
– *srcRect* –	Long—Pointer to source rectangle data record
– *destLocInfoPtr* –	Long—Pointer to destination `locInfo` data record
– *destRect* –	Long—Pointer to destination rectangle data record
seedH	Word—Horizontal offset (pixel) to starting fill point
seedV	Word—Vertical offset (pixel) to starting fill point
resMode	Word—Resolution mode
– *patternPtr* –	Long—Pointer to fill pattern
– *leakTblPtr* –	Long—Pointer to leak-through color table
	<—SP

Stack after call

Previous contents	
	<—SP

Errors			
	$0201	`memErr`	NewHandle error occurred
	$1211	`badRectSize`	1) Height or width is negative, 2) *destRect* not same size as *srcRect*, or 3) source or destination rectangle not within its boundary rectangle
	$1212	`destModeError`	`destMode` portion of *resMode* invalid

C

```
extern pascal void SeedFill(srcLocInfoPtr, srcRect,
          destLocInfoPtr, destRect, seedH, seedV,
          resMode, patternPtr, leakTblPtr);

Pointer   srcLocInfoPtr, srcRect, destLocInfoPtr,
          destRect, patternPtr, leakTblPtr;
Word      seedH, seedV, resMode;
```

srcLocInfoPtr Points to a `locInfo` data record containing the definition for the source rectangle for the fill operation.

srcRect Points to a rectangle, in local coordinates, that contains the source pixel image.

destLocInfoPtr, destRect

Refer to output `locInfo` record and rectangle, respectively. These fields allow you to copy the output to a different location in a different rectangle. If you want the output of the operation to overlay the input image, set the source and destination pointers to the same values.

seedH, seedV Specify the horizontal and vertical offsets into the source pixel image of the point at which to start the fill operation.

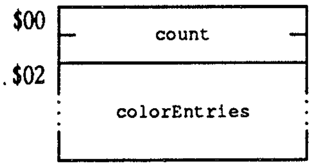
resMode Indicates the resolution mode for the fill as well as initialization and drawing options:

`destMode`	bits 12–15	Indicates initialization and drawing options: 0000 - copy source to destination (obliterating destination) 0001 - leave destination alone (overlay source onto destination) 0010 - initialize destination to zeroes before drawing 0011 - initialize destination to ones before drawing other values are invalid
Reserved	bits 2–11	Must be set to 0
`res`	bits 0–1	Indicates the resolution for the operation: 00 - 640 pure 01 - 640 dithered 10 - 320 mode 11 - invalid

patternPtr Pointer to the fill pattern for the operation, or flag specifying special fill pattern:

NIL	Use an all zeroes pattern when writing to destination
$FFFFFFFF	Use an all ones pattern when writing to destination
Other	Assumed to be valid pointer to fill pattern

leakTblPtr Pointer to a structure that defines the colors to be covered. The structure contains a count word, indicating the number of color entries in the table, and a color entry for each color to be leaked. Each color entry contains the offset into the color table for that color:



Word—Count of color entries to follow

`count` Words—Offset into color table for each color

`SpecialRect` $0C12

Frames and fills a rectangle in a single call, making separate calls to `FrameRect` and `FillRect` unnecessary.

The pen used to draw the rectangle frame in 640 mode is 2 pixels wide and 1 pixel high; in 320 mode, the pen is 1 pixel wide and 1 pixel high.

Parameters

Stack before call

Previous contents	
– *recPtr* –	Long—Pointer to rectangle to draw
frameColor	Word—Color of rectangle frame
fillColor	Word—Color of rectangle interior
	<—SP

Stack after call

Previous contents	
	<—SP

Errors None

C

```
extern pascal void SpecialRect(recPtr, frameColor,
          fillColor);

Pointer   recPtr;
Word      frameColor, fillColor;
```

frameColor, fillColor
The low-order four bits of each of these parameters specify the color.

Chapter 45 **Resource Manager**

This chapter documents the features of the new Resource Manager. This is a new tool set, not previously documented in the *Apple IIGS Toolbox Reference.*

About the Resource Manager

The Resource Manager provides applications access to **resources**, which can contain such items as menus, fonts, and icons. Most basically, a resource is a formatted collection of data. The Resource Manager does not know the format or content of any given resource. Your application defines the content of its resources, or may use standard resources defined by the system. Resource Manager facilities allow applications to create, use, and manipulate these resources.

Generally, your program will access the Resource Manager indirectly, as a result of using other tool sets, such as the Window Manager or Control Manager, which use resources. However, if your program manages its own resources, it will have to issue some Resource Manager calls directly. Further, you may want to write a program that creates and edits resources. Such a program would make thorough use of Resource Manager tool calls.

The following table summarizes the capabilities of the Resource Manager in a grouped listing of its tool calls. Later sections of this chapter discuss resources in greater detail, and define the precise syntax for the Resource Manager tool calls.

Routine	Description
Housekeeping routines	
`ResourceBootInit`	Called only by the Tool Locator—must not be called by an application
`ResourceStartUp`	Informs the Resource Manager that an application wants to use its facilities
`ResourceShutDown`	Informs the Resource Manager that an application is finished using resource tool calls
`ResourceReset`	Called only when the system is reset—must not be called by an application
`ResourceVersion`	Returns the Resource Manager version number
`ResourceStatus`	Returns the Resource Manager's operational status

Resource access routines

`AddResource`	Creates a new resource and adds it to a specified resource file
`RemoveResource`	Deletes a resource from a resource file
`LoadResource`	Loads a resource into memory
`LoadAbsResource`	Loads a resource into a specified memory location
`GetIndResource`	Loads a resource given an index into a specified resource type
`ReleaseResource`	Removes a loaded resource from memory
`DetachResource`	Removes a loaded resource from the control of the Resource Manager, but leaves the resource in memory
`WriteResource`	Writes a changed resource to its resource file

Resource maintenance routines

`GetResourceAttr`	Returns the attributes of a resource
`SetResourceAttr`	Sets the attributes of a resource
`GetResourceSize`	Returns the size in bytes of a resource
`MarkResourceChange`	Sets the value of the changed attribute of a resource
`SetResourceID`	Changes the ID of a resource
`UniqueResourceID`	Obtains a unique resource ID for a resource of a specified type
`CountTypes`	Returns the number of different resource types in all open resource files for an application
`GetIndTypes`	Returns a resource type value associated with an index into the array of all active resource types
`CountResources`	Returns the number of resources of a specified type
`MatchResourceHandle`	Finds the ID and type of a resource, given its handle
`ResourceConverter`	Installs resource converter routines
`SetResourceLoad`	Controls whether the Resource Manager loads resources from disk

Resource file routines

`CreateResourceFile`	Creates and initializes a resource file
`OpenResourceFile`	Opens a resource file for access by the Resource Manager
`CloseResourceFile`	Closes an open resource file
`UpdateResourceFile`	Writes all in-memory resource changes to the appropriate resource file, making those changes permanent
`GetCurResourceFile`	Returns the file ID of the current resource file
`SetCurResourceFile`	Sets the current resource file
`SetResourceFileDepth`	Sets the number of resource files that the Resource Manager will search when locating a specific resource
`GetOpenFileRefNum`	Returns the GS/OS file reference number for an open resource file
`HomeResourceFile`	Returns the file ID of the resource file that contains a specified resource
`GetMapHandle`	Returns the handle of a resource map for an open resource file

Application switching routines

`GetCurResourceApp`	Returns the User ID of the application currently using the Resource Manager
`SetCurResourceApp`	Sets the User ID of the application now using the Resource Manager

About resources

A resource is a formatted collection of data, such as a menu, a font, or a program itself. The format of the data in a resource is determined by the program that uses the resource, or by the system for standard resources. A program maintains its resources separate from the program code itself. This very separation is the primary benefit of using resources—program code is immune to data content changes, and program data is immune to program code changes, even to changes in programming language.

Resources, in turn, are grouped into **resource files**, which correspond to the resource forks of GS/OS files. A given resource file may contain one or more resources of various format. An application that uses resources may store those resources in its own resource file, or may access resources in a resource file that is not directly associated with the program. The Resource Manager provides routines to access and manipulate resources in a resource file.

You can create the resource fork for your program in a variety of ways. Resource compilers convert text-based resource definitions into resources in a valid resource file. You can use an existing resource compiler, or you can create your own. Alternatively, you can write a program that creates a resource file and its resources, using Resource Manager tool calls. Finally, resource editors allow you to create resources interactively.

Identifying resources

Programs identify resources with a resource specification consisting of a **resource type** and a **resource ID** number. The resource type (or just **type**) defines a class or group of resources that share a common format. The resource ID (or just **ID**) uniquely identifies a specific resource of a given type. Taken together, the resource type and ID completely identify the resource and define its format. The ID for a resource must be unique within the context of its type; however, the same ID number may be used for resources of different type.

Resource types

The resource type defines a class of resources that share a common format. The system defines several standard types for resources used to interact with system or toolbox functions. These standard types and the formats of their associated resources are documented in Appendix E, "Resource Types," in this book. In addition, your program may define unique resource types for its custom resources. Since the Resource Manager knows nothing about the format or content of the resources it manages, you have complete freedom to define the resources you need.

The resource type is a word value. The following table summarizes valid resource type values:

Type value range	Use
$0000	Invalid resource type; do not use
$0001 through $7FFF	Available for application use
$8000 through $FFFF	Reserved for system use

Resource IDs

The resource ID uniquely identifies a particular resource of a given type in a resource file. Every resource in a resource file must have an ID value that is unique within the context of its resource type. Resources of different type may, however, have the same ID value.

The resource ID is a long value. Even though the resource ID is only meaningful in the context of a given resource type, the system does place restrictions on the ID values you can assign. The following table summarizes the allowable ranges for ID values:

ID value range	Use
$00000000	Invalid resource ID; do not use
$00000001 through $07FEFFFF	Available for application use
$07FF0000 through $07FFFFFF	Reserved for system use
$08000000 through $FFFFFFFF	Invalid values; do not use

When creating a new resource, use the `UniqueResourceID` tool call to obtain a resource ID. The Resource Manager will allocate a new, unique resource ID for you. You can force the ID to fall within a desired range, in order to group resources by resource ID within resource type. Each ID range contains 65,535 possible values. The ID range value provides the high-order word of the long-word resource ID. The following table summarizes the allowable ranges:

ID Range	Lowest possible ID returned	Highest possible ID returned
$0000	$00000001 (NIL is invalid)	$0000FFFF
$0001	$00010000	$0001FFFF
$0002	$00020000	$0002FFFF
.		
.	(and so on)	
.		
$07FE	$07FE0000	$07FEFFFF
$07FF	Reserved for system use	
$0800-$FFFE	Invalid range values	
$FFFF	$00000001	$07FEFFFF
	(directs Resource Manager to allocate from any application range)	

Resource Names

As an alternative to identifying a resource of a given type by an ID, you may choose to assign it a **resource name**. Your application may then use the resource type and name combination to uniquely identify the resource. In some cases, this may be more handy than using the numeric ID. The resource name must be unique within the context of a given resource type. You should note that the Resource Manager does not provide call-level support for resource names. However, the rResName resource ($8014) defines the standard layout for resource names. If you choose to use resource names, or you use developer tools that support named resources, you should be careful to use the standard data structures for defining those names.

Using resources

In most cases, applications only use the Resource Manager indirectly, by using other tool sets that, in turn, use resources to store their data structures. Even if your program defines resources, either for its own data or for data to be used by the system, it will have to issue only a few Resource Manager calls to use those resources. However, programs that create and manipulate resources and resource files must use far more Resource Manager functionality. The next several paragraphs describe the steps your program must follow in order to use its predefined resources.

1. Unlike most other tool sets, your program does not have to start the Resource Manager. At startup time, the system automatically loads and initializes the Resource Manager from the RESOURCE.MGR file in the SYSTEM.SETUP directory of the boot disk. The Resource Manager then opens the system resources file, SYS.RESOURCES in the SYSTEM.SETUP directory, if it is present.
2. To use the Resource Manager, you program must log in, using the `ResourceStartUp` tool call. This call informs the Resource Manager that your program is going to be using its services. As an alternative, your program may issue the Tool Locator `StartUpTools` call.
3. Issue the `OpenResourceFile` tool call to open each resource file for your application. If your program issued the Tool Locator `StartUpTools` call, then it need not explicitly open its resource fork before trying to use resources located there. If, on the other hand, your program used the `ResourceStartUp` tool call, then it must issue an `OpenResourceFile` call for its resource fork before accessing any resources stored there.
4. As part of termination processing, call `ResourceShutDown` to log out from the Resource Manager. The Resource Manager will automatically close any open resource files. Once your program has issued a `ResourceShutDown` call, it should not make any other Resource Manager calls, except for `ResourceStartUp`.

Resource attributes

Every resource has an associated set of attributes that define the current state of the resource as well as limits on how the resource can be used. The Resource Manager stores these attributes in an attributes flag word (or Attributes word) for the resource (specifically, the `resAttr` field in the resource reference record). Your program can read and write this attribute word by means of the `GetResourceAttr` and `SetResourceAttr` tool calls. In addition, the `MarkResourceChange` tool call provides a convenient mechanism for changing the setting of the changed flag for the resource, which indicates whether the resource has been changed since it was read from disk.

Many of the attributes govern the type of memory used to store the resource when the Resource Manager reads it in from disk. These attributes directly correspond to flags in the Memory Manager `NewHandle` tool call memory Attributes word. When it allocates memory for a resource to be loaded from disk, the Resource Manager masks out the other bits and passes the Attributes word to the `NewHandle` call. See the `NewHandle` tool description in Chapter 12, "Memory Manager," in Volume 1 of the *Toolbox Reference* for the format and content of the memory Attributes word.

The following table describes the content of the attribute word for a resource:

`attrLocked`	Bit 15	Passed to Memory Manager `NewHandle` tool call when memory is allocated for the resource: 1 - Memory for resource locked; cannot be moved or purged 0 - Memory for resource not locked
`attrFixed`	Bit 14	Passed to Memory Manager `NewHandle` tool call when memory is allocated for the resource: 1 - Memory for resource is fixed and cannot be moved 0 - Memory for resource need not be fixed
Reserved	Bits 12–13	Must be set to 0.
`resConverter`	Bit 11	Indicates whether the resource requires a resource converter routine (see "Resource converter routines" later in this chapter for more information): 1 - Resource requires a converter routine 0 -Resource does not require a converter routine
`resAbsLoad`	Bit 10	Governs whether the resource must be loaded at a specific memory location. Resources that must be loaded at an absolute location must be created by a resource editor or compiler. 1 - Resource to be loaded at specific location 0 - Resource need not be loaded at a specific location

`attrPurge`	Bits 8–9	Passed to Memory Manager `NewHandle` tool call when memory is allocated for the resource: 11 - Purge level 3 10 - Purge level 2 01 - Purge level 1 00 - Purge level 0
`resProtected`	Bit 7	Indicates whether the resource is write-protected. If set to 1, then applications may not update the resource on disk: 1 - Resource is write-protected 0 - Resource is not write-protected
`resPreLoad`	Bit 6	Specifies whether the Resource Manager should load the resource into memory at `OpenResourceFile` time. If set to 1, then this resource will be loaded into memory when the resource file is opened, rather than when the resource itself is accessed: 1 - Preload the resource 0 - Do not preload the resource
`resChanged`	Bit 5	Indicates whether the resource has been changed. If set to 1 for a non–write-protected resource, the Resource Manager will update the resource on disk at `CloseResourceFile` time: 1 - Resource has been changed in memory, and therefore differs from the version stored on disk 0 - Resource has not been changed in memory
`attrNoCross`	Bit 4	Passed to Memory Manager `NewHandle` tool call when memory is allocated for the resource: 1 - Memory may not cross bank boundary 0 - Memory may cross bank boundary
`attrNoSpec`	Bit 3	Passed to Memory Manager `NewHandle` tool call when memory is allocated for the resource: 1 - May not use special memory 0 - May use special memory
`attrPage`	Bit 2	Passed to Memory Manager `NewHandle` tool call when memory is allocated for the resource: 1 - Memory must be page-aligned 0 - Memory need not be page-aligned
Reserved	Bits 0–1	Must be set to 0

Resource file format

A resource file is not a file in the strictest sense; actually, it is one of two parts, or forks, of a GS/OS file. Every file has a resource fork and a data fork, either of which may be empty. The data fork contains information for the application as well as the application code itself, and is formatted according to the needs of the application. Programs manipulate data in the data fork with GS/OS file system calls.

The Resource Manager defines the format of the resource fork. Programs read and manipulate resources with Resource Manager tool calls. As a result, applications do not need to know the format of the resource fork to use the resources stored there. You can create resources and load them into a resource file with the aid of a Resource Editor, or with whatever tools are available in your development environment.

A resource file consists primarily of resource data and a resource map. The resources themselves comprise the resource data. The resource map is a directory to those resources, containing both location and size information. Each entry in the map on disk contains the offset of the resource into the file; in memory, the entry contains a handle to the resource if it is loaded. The Resource Manager reads the resource map into memory at resource file open time, and maintains it in memory until the file is closed.

Resource File IDs

an application opens a resource file, the Resource Manager assigns that open file a **file ID**, which identifies the file to the Resource Manager. Every open resource file has a file ID that is unique in the entire system. Many Resource Manager tool calls require the file ID in order to identify the resource file to be accessed. The file ID for the system resource file is always $0001 (sysFileID).

The OpenResourceFile tool call returns the file ID for a resource file. Note that the file ID does not correspond to the GS/OS file reference number. Use the GetOpenFileRefNum Resource Manager tool call to obtain the GS/OS file number for a resource file.

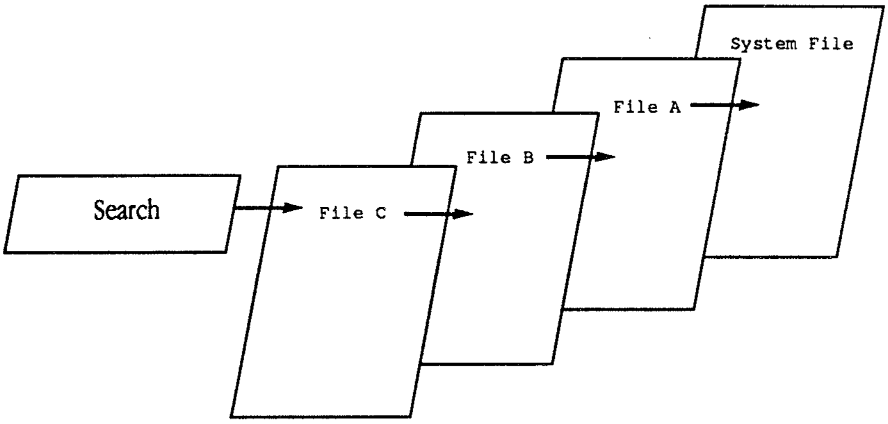
Resource file search sequence

As your program opens resource files, the Resource Manager adds those files to the head of the resource file search chain for your application. The Resource Manager uses this search chain for many of its operations, such as locating a resource. The system resource file is always the last file in the search sequence. When it runs the search chain, the Resource Manager first checks all files in the application chain, then checks in the system resource file, if one is defined.

You control the application file search sequence by the order in which your program opens its resource files. For example, if your program issues the following tool calls:

```
OpenResourceFile        File A
OpenResourceFile        File B
OpenResourceFile        File C
```

Resource Manager builds the following search chain for your application:



The most recently opened file (in this example, File C) is referred to as the current resource file (or simply, the current file). It is also called the first resource file (or first file), since it is first file accessed during a search. The least recently opened application resource file (File A) is called the last resource file (or last file), because it is the last application file to be searched.

During a search, which happens on nearly every Resource Manager tool call that accepts resource type and ID arguments, the Resource Manager starts with the current file, and searches through the chain until it either finds the desired resource or exhausts the file list. Note that the search stops with the first occurrence of a matching resource; a second instance of a resource with the same ID and type will not be found unless your application asserts further control over the resource search sequence.

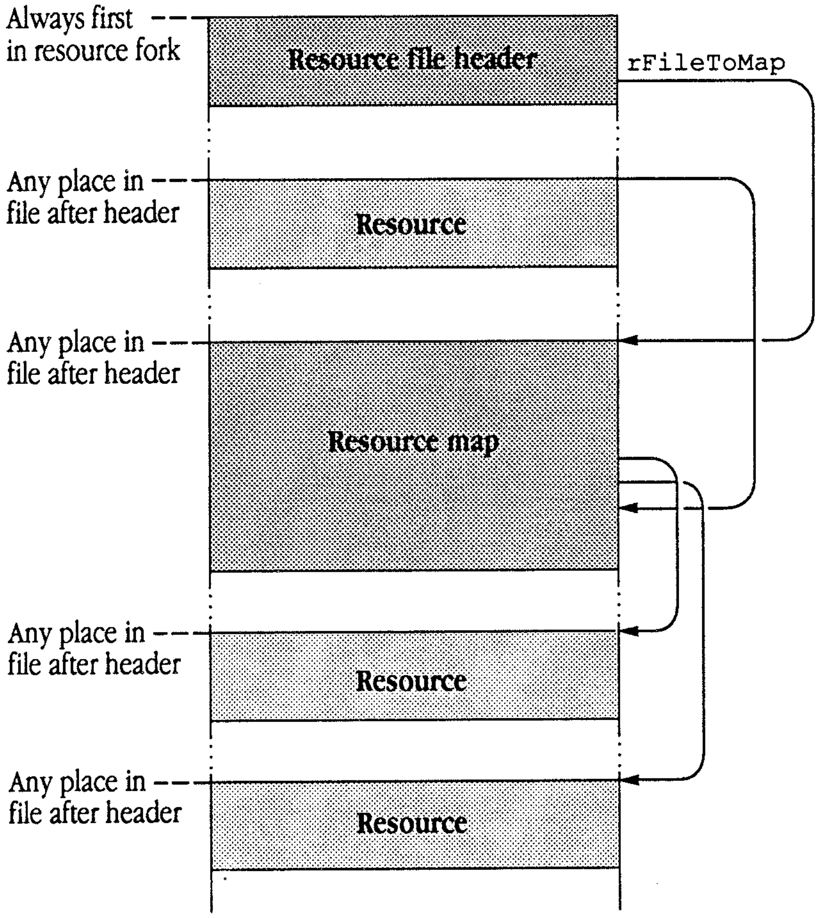
The Resource Manager provides tool calls that allow your program to control the search sequence for the resource file chain. The `SetCurResourceFile` tool call changes the current resource file, so that any resource file, including the System file, can be the first file searched, though the search still terminates when the Resource Manager either finds the desired resource, or hits the end of the file chain. The `SetResourceFileDepth` tool call controls the number of files the Resource Manager will search before giving up. By using these calls, your program can fine-tune resource searches for performance or can inhibit access to some resource files for some searches.

Resource file layout and data structures

This section describes the format of a resource file on disk. This information is intended only for application programmers who are writing tools to create, delete, or edit resources in the resource fork.

Figure 45-1 shows the internal layout of the resource fork of a file. The resource file header is the only data block that resides at a fixed location in the fork; it is always the first data item in the fork. Along with other control information, the resource file header contains the file offset to the resource map. The map, in turn, contains location and size information for each resource contained in the file.

- **Figure 45-1** Resource file internal layout

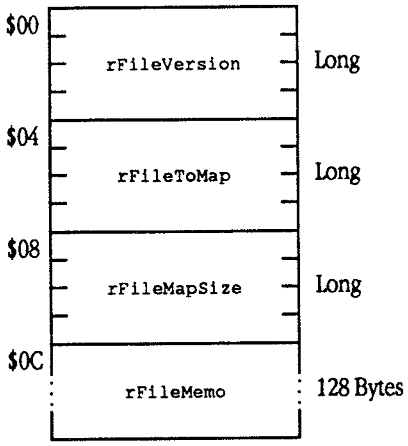

Always first in resource fork
Resource file header
rFileToMap
Any place in file after header
Resource
Any place in file after header
Resource map
Any place in file after header
Resource
Any place in file after header
Resource

The Resource Manager controls the relative positions of all elements of the resource fork. It will move or resize the map or resources as required. Therefore, your program should never rely on the location of any element in the fork, except for the resource file header.

The following sections present the format of the resource file header, resource map, and their associated data structures in greater detail. These descriptions present version 0 layout infomation. Future system releases may support other versions with different layouts. Your program should check the value in the `rFileVersion` field in the resource file header before manipulating a resource file.

Resource file header

The resource file header is the first data block in every resource fork, and has the following format:



`rFileVersion` Version number defining layout of resource file. Currently, only version 0 is supported. This field allows IIGS resource files to be distinguished from Macintosh resource files; the first long in Macintosh resource files must have a value that is greater than 127.

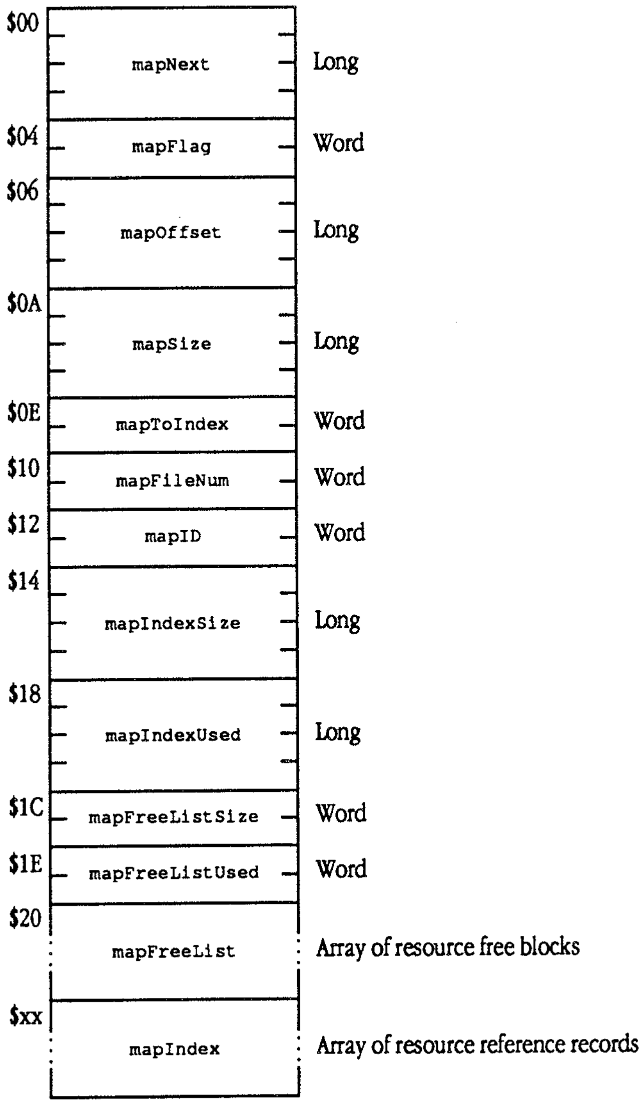
`rFileToMap` Offset, in bytes, to beginning of the resource map. This offset starts from the beginning of the resource file.

`rFileMapSize` Size, in bytes, of the resource map.

`rFileMemo` Reserved for application use. The Resource Manager does not provide any facility for reading or writing this field. Your program must use GS/OS file system calls to access the `rFileMemo` field.

Resource map

The resource map provides indexes to the resources stored in the resource file, and is formatted as follows:



$00	mapNext	Long
$04	mapFlag	Word
$06	mapOffset	Long
$0A	mapSize	Long
$0E	mapToIndex	Word
$10	mapFileNum	Word
$12	mapID	Word
$14	mapIndexSize	Long
$18	mapIndexUsed	Long
$1C	mapFreeListSize	Word
$1E	mapFreeListUsed	Word
$20	mapFreeList	Array of resource free blocks
$xx	mapIndex	Array of resource reference records

`mapNext` Handle to resource map for next resource file in the search chain. Set to NIL if last file in chain. This field is only valid when the map is in memory.

`mapFlag` Contains control flags defining the state of the resource file:

Reserved	bits 2–15	Set to 0
`mapChanged`	bit 1	Indicates whether the resource map has been modified, and must therefore be written to disk when the file is closed: 1 - Map changed 0 - Map not changed
Reserved	bit 0	Set to 0

`mapOffset` Offset, in bytes, to the resource map from the beginning of the resource file.

`mapSize` Size, in bytes, of the resource map on disk. Note that the memory image of the map may have a different size due to resource or resource file changes during program execution.

`mapToIndex` Offset, in bytes, from beginning of map to the beginning of the `mapIndex` array of resource reference records.

`mapFileNum` GS/OS file reference number. This field is valid only in memory.

`mapID` Resource Manager file ID for the open resource file. This field is valid only in memory.

`mapIndexSize` Total number of resource reference records in `mapIndex`.

`mapIndexUsed` Number of used resource reference records in `mapIndex`.

`mapFreeListSize`
Total number of resource free blocks in `mapFreeList`.

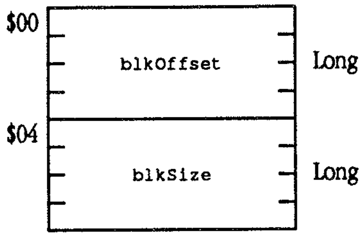
`mapFreeListUsed`
Number of used resource free blocks in `mapFreeList`.

`mapFreeList` Array of resource free blocks, which describe free space in the resource file.

`mapIndex` Array of resource reference records, which contain control information about the resources in the resource file.

Resource free block

The resource free block describes a contiguous area of free space in the resource file. The resource map contains a variable-sized array of these blocks at mapFreeList. Note that each resource file has at least one resource free block, defining free space from the end of the resource file to $FFFFFFFF. Each resource free block has the following format:

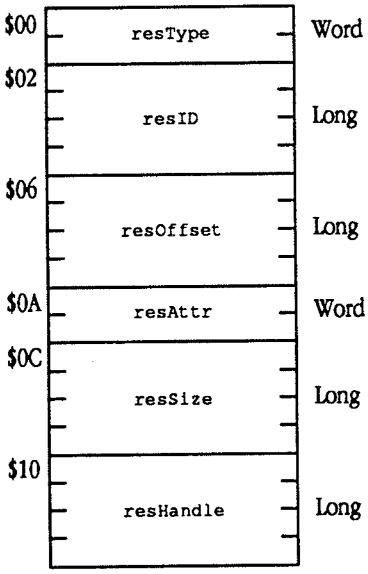


blkOffset Offset, in bytes, to the free block from the start of the resource fork. A NIL value indicates the end of the used blocks in the array.

blkSize Size, in bytes, of the free block of space.

Resource reference record

The resource reference record contains control information about a resource. The resource map contains a variable-sized array of these blocks, starting at the location specified in `mapToIndex`. Each resource reference record has the following format:



`resType` Resource type. NIL value indicates last used entry in the array.

`resID` Resource ID.

`resOffset` Offset, in bytes, to the resource from the start of the resource file.

`resAttr` Resource attributes. See "Resource attributes" earlier in this chapter for bit flag definitions.

`resSize` Size, in bytes, of the resource in the resource file. Note that the size of the resource in memory may differ, due to changes made to the resource by application programs or by resource converter routines.

`resHandle` Handle of resource in memory. NIL value indicates that the resource has not been loaded into memory. Your program can determine the in-memory size of the resource by examining the size of this handle.

Resource converter routines

The Resource Manager supports the concept of resource converter routines. Converter routines format resources for access by your program, allowing the memory format of a resource to differ from its disk representation. These routines can be used, for example, to store resources in a compressed form on disk, to reformat common resources for different programs or operating environments, or to perform code relocation.

When loading or unloading a resource, the Resource Manager determines whether to invoke a converter routine by examining the `resConverter` flag in the Attributes word for the resource. If that flag is set to 1, indicating that the resource must be converted before being read or written, the Resource Manager invokes the appropriate converter routine for the resource type. The converter routine may then reformat the resource in any way it chooses.

Your program registers a converter routine with the `ResourceConverter` tool call. At that time, your program must specify the resource type to be handled by the converter routine. One converter routine may handle more than one resource type; your program must issue separate `ResourceConverter` tool calls for each type to be converted.

The Resource Manager tracks resource converters in two types of lists. Each application has a private application routine list, which can contain up to 10,922 entries. In addition, the Resource Manager maintains a system routine list, which is available to all applications. When searching for a converter routine for a specific resource type, the Resource Manager first checks the application list, then the system list. As a result, your program can override a standard converter routine by registering a routine for the same resource type in its application converter routine list. Applications should never log routines into or out of the system list.

When the Resource Manager invokes a converter routine, it loads the stack with values specifying the operation to be performed and any needed parameters. Before returning control to the Resource Manager, the converter routine should set a condition code in the A register (any nonzero value indicates an error) and return the appropriate result value on the stack. The following sections provide detailed descriptions of the entry and exit conditions for each converter routine operation.

◆ *Note:* Not all resource converters support conversion when resources are written back to disk. The supplied code resource converter only functions on resource read operations, for example. Consequently, if you are unsure about the behavior of a given resource converter, you should not mark converted resources changed, since the converter may write them back to disk in an unexpected format.

ReadResource

Read a resource from disk into memory. The converter routine must load the file from disk and perform any necessary reformatting.

On entry, *convertParam* contains a pointer to a GS/OS read file parameter block (see the *GS/OS Reference* for more information on GS/OS file manipulation and data structures). The file mark is set to the beginning of the file and the block is set to read the entire resource from disk. In order to read the file, your program can do the following:

```
pushlong    convertParam        Pointer to read parameter block
pushword    $2012               GS/OS read command code
jsl         $E100B0             Call GS/OS
            check for errors
```

The `resPointer` field contains a pointer to the resource reference record, which contains location and size information about the resource in memory (see "Resource file format" earlier in this chapter for information on the format and content of the resource reference record). Your program should verify that the number of bytes loaded corresponds to the size of the resource on disk (compare `resSize` value to the size of the handle that received the resource). Your program should also check to see if the resource must be loaded at an absolute location (`resAbsLoad` flag set to 1 in `resAttr` word of the resource reference record). If so, be careful to convert the resource into the appropriate location.

If, during resource conversion, the converter routine must copy the resource into a different handle, the routine must load that new handle into the `resHandle` field of the resource reference record, and dispose of the original handle. Upon return, the handle to the converted resource should retain its original Memory Manager attributes (locked, and so on).

Upon successful completion, the converter routine should return a NIL *Result* value. In case of error, the routine should return a non-NIL *Result*, and must free the memory referenced by the `resHandle` field in the resource reference record, and set that field to NIL.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
convertCommand	Word—Command to be performed (will be 0: `readResource`)
– *convertParam* –	Long—Pointer to GS/OS read file parameter block
– *resPointer* –	Long—Pointer to resource reference record
	<—SP

Stack after call

Previous contents	
– *Result* –	Long—NIL if successful; error code if error (low-order word)
	<—SP

WriteResource

Write a resource from memory onto disk. The converter routine must perform any necessary reformatting and write the file to disk.

On entry, *convertParam* contains a pointer to a GS/OS write file parameter block (see the *GS/OS Reference* for more information on GS/OS file manipulation and data structures). The file mark is set to the beginning of the file on disk and the block is set to write the entire resource. Before issuing a `writeResource` command, the Resource Manager will determine the amount of disk space required for the resource by calling the `returnDiskSize` function in the converter routine.

In order to write the file, your program can do the following:

```
pushlong    convertParam      Pointer to read parameter block
pushword    $2013             GS/OS write command code
jsl         $E100B0           Call GS/OS
            check for errors
```

The `resPointer` field contains a pointer to the resource reference record, which contains location and size information about the resource in memory (see "Resource file format" earlier in this chapter for information on the format and content of the resource reference record). The Resource Manager will dispose of the handle to the resource after calling `writeResource`.

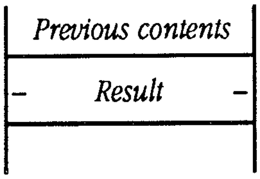
This function must return a NIL *Result* value.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
convertCommand	Word—Command to be performed (will be 2: `writeResource`)
– *convertParam* –	Long—Pointer to GS/OS write file parameter block
– *resPointer* –	Long—Pointer to resource reference record
	<—SP

Stack after call



Long—Must be set to NIL

<—SP

ReturnDiskSize

Determines the amount of space a resource will require on disk, and returns that value to the caller. Note that this call is not valid for resources that are loaded into absolute memory, since the size of these resources cannot change.

The *convertParam* parameter is undefined.

The `resPointer` field contains a pointer to the resource reference record, which contains location and size information about the resource in memory (see "Resource file format" earlier in this chapter for information on the format and content of the resource reference record).

On exit, *Result* contains the amount of disk space required to store the resource, in bytes. If this new size differs from the original file size, the Resource Manager frees the old space and allocates a new file.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
convertCommand	Word—Command to be performed (will be 4: `returnDiskSize`)
– *convertParam* –	Long—Undefined
– *resPointer* –	Long—Pointer to resource reference record
	<—SP

Stack after call

Previous contents	
– *Result* –	Long—Bytes of disk space required to store resource
	<—SP

Application switchers and desk accessories

Desk accessories and application-switching programs must be careful to preserve the state of the Resource Manager before using its facilities. The Resource Manager provides tool calls that allow such programs to switch the currently active Resource Manager application. The `GetCurResourceApp` tool call returns the User ID of the application that is currently using the Resource Manager. This call returns a special value if the Resource Manager is not in use. The `SetCurResourceApp` tool call changes the current application, by loading a new User ID value. It is the responsiblity of the code doing the switch to use these calls.

In the following example, the Resource Manager is already active and the application switcher has previously registered itself with the Resource Manager with the `ResourceStartUp` tool call. The switching program must save the User ID of the program that is currently using the Resource Manager before it issues any other Resource Manager tool calls:

```
;            . . .
             pha                ; space for result from GetCurResourceApp
             GetCurResourceApp
;                                 get current app user ID, save on stack
             pushword myUserID  ; pass my user ID to Resource Manager
             SetCurResourceApp
;                                 switch to my resources and files
;            . . .
;
             SetCurResourceApp
;                                 restore original application user ID
;                                  (saved on stack after GetCurResourceApp
;                                   tool call).
;            (return to caller)
```

In the case where your program must first log into the Resource Manager, it must issue the `ResourceStartUp` tool call before calling any other Resource Manager functions:

```
;            (on entry to desk accessory task handler)
;
             pushword #0        ; Prime for FALSE if Resource Manager
                                ;  is not active
             ResourceStatus     ; check for active Resource Manager
             pla                ;  .
             beq   NoResMgr     ; Exit if Resource Manager not active
;
             pha                ; Space for result
             GetCurResourceApp
;                                 get current app user ID, save on stack
             pushword myUserID ; pass my user ID to Resource Manager
             SetCurResourceApp
;                                 switch to my resources and files
;            . . .
;
             SetCurResourceApp
;                                 restore original application user ID
;                                  (saved on stack after GetCurResourceApp
;                                   tool call).
NoResMgr
;            (return to caller)
```

Resource Manager housekeeping routines

This section discusses the standard housekeeping routines, in call number order.

`ResourceBootInit` $011E

Initializes the Resource Manager.

▲ **Warning** An application must never make this call. ▲

Parameters The stack is not affected by this call. There are no input our output parameters.

Errors None

C Call must not be made by an application.

`ResourceStartUp` $021E

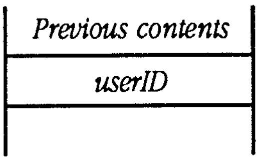
Notifies the Resource Manager that an application wishes to open and use its own resource files. Unlike other tool set `StartUp` calls, this call is not required in all circumstances. If your application only uses system resources (located in the system resource file), then it does not have to issue a `ResourceStartUp` tool call. On the other hand, if your application uses non-system resources, then it must issue this tool call prior to opening those resource files.

If your application issues this call, then it must issue the `ResourceShutDown` tool call before quitting.

Note that the Tool Locator `StartUpTools` tool call automatically starts the Resource Manager.

Parameters

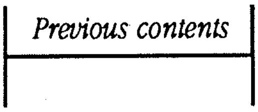
Stack before call



Word—Application's User ID (obtained at program start time)

<—SP

Stack after call



<—SP

Errors	Memory Manager errors	Returned unchanged

C

```
extern pascal void ResourceStartUp(userID);

Word      userID;
```

ResourceShutDown $031E

Notifies the Resource Manager that an application is finished using its own resource files. The Resource Manager updates, closes, and frees memory for any open resource files. Unlike other tool set ShutDown calls, the Resource Manager is still active after this call. However, after calling ResourceShutDown, your application can only access the system resource file.

If your application called ResourceStartUp, then it must issue a ResourceShutDown call before quitting.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

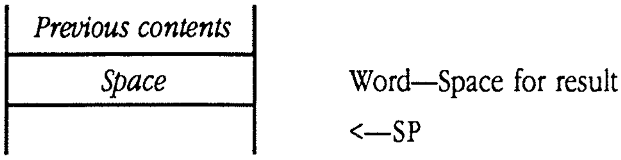
```
extern pascal void ResourceShutDown();
```

ResourceVersion $041E

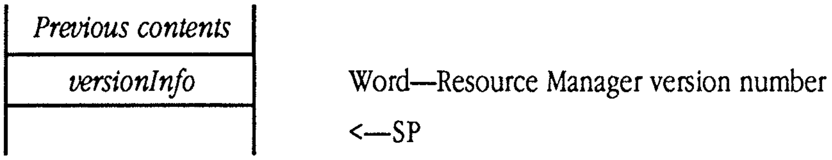
Retrieves the Resource Manager version number. The *versionInfo* result will contain the information in the standard format defined in Appendix A, "Writing Your Own Tool Set," in Volume 2 of the *Toolbox Reference.*

Parameters

Stack before call



Stack after call



Errors None

C `extern pascal Word ResourceVersion();`

`ResourceReset` $051E

Resets the Resource Manager; issued only when the system is reset.

> ▲ **Warning** An application must never make this call. ▲

Parameters The stack is not affected by this call. There are no input our output parameters.

Errors None

C Call must not be made by an application.

`ResourceStatus` $061E

Returns a flag indicating whether the Resource Manager is active. If the Resource Manager loaded and initialized successfully at system startup, then this function returns a value of TRUE. If the Resource Manager did not successfully load or initialize, then the Tool Locator returns a `funcNotFoundErr` error code ($0002).

◆ *Note:* If your program issues this call in assembly language, initialize the result space on the stack to NIL. Upon return from `ResourceStatus`, your program need only check the value of the returned flag. If the Resource Manager is not active, the returned value will be FALSE (NIL).

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
	<—SP

Stack after call

Previous contents	
activeFlag	Word—BOOLEAN; TRUE if Resource Manager is active
	<—SP

Errors	$0002	`funcNotFoundErr`	Resource Manager not active

C

```
extern pascal Word ResourceStatus();
```

Resource Manager tool calls

This section discusses the Resource Manager tool calls, in call name order.

`AddResource` $0C1E

Adds a resource to the current resource file. The Resource Manager marks the new resource as changed and will write the new resource to disk when the file is updated. Your program specifies the attributes of the new resource in a flag word passed to `AddResource`. Some of these attributes control how memory is allocated for the new resource when it is loaded by an application, others govern Resource Manager processing. For more information about the various attributes, see "Resource attributes" earlier in this chapter.

Parameters

Stack before call

Previous contents	
– *resourceHandle* –	Long—Handle of resource in memory
resourceAttr	Word—Attributes of the resource
resourceType	Word—Type for resource
– *resourceID* –	Long—ID for resource
	<—SP

Stack after call

Previous contents	
	<—SP

Errors			
	$1E04	`resNoCurFile`	There is no current resource file
	$1E05	`resDupID`	Another resource in the current file is using this ID
	$1E0E	`resDiskFull`	Volume full
	`WriteResource` errors		Returned unchanged
	Memory Manager errors		Returned unchanged
	GS/OS errors		Returned unchanged

C

```
extern pascal void AddResource(resourceHandle,
          resourceAttr, resourceType, resourceID);

Long      resourceHandle, resourceID;
Word      resourceAttr, resourceType;
```

resourceHandle Specifies the memory location and size of the resource to be added to the current resource file. If the handle is empty, `AddResource` creates a resource with zero length. Never pass a handle that was created by the Resource Manager, unless the resource in that handle has been detached (see `DetachResource` tool call).

If `resAbsLoad` in *resourceAttr* is set to 1, then the Resource Manager obtains the size of the resource from the `mapSize` field in the resource map.

resourceAttr Bit flags defining the attributes of the resource to be added. For information about the specific flags, see "Resource attributes" earlier in this chapter.

resourceType Type of resource to be added. See "Identifying resources" earlier in this chapter for details.

resourceID ID for new resource. Must be unique among resources of the same type. See "Identifying resources" earlier in this chapter for more information. Use the `UniqueResourceID` tool call to obtain a unique ID.

CloseResourceFile $0B1E

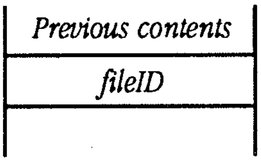
Updates a specified resource file, frees any memory used by the resource map for the file and any resources currently loaded, and closes the file. Your program passes the file ID of the resource file to be closed. This file ID is obtained from the `OpenResourceFile` tool call.

If the file being closed is the current resource file, the next file in the resource file list becomes the current resource file. Your program can close the system resource file by passing the system file ID ($0001). Note, however, that some tool calls require system resources (for example, the system stores the control definition procedure for icon button controls in the system resource file). These calls will not work if you close the system resource file, or set the search depth so shallow that the system resource file is inaccessible (see "`SetResourceFileDepth`" later in this chapter).

◆ *Note:* Your program does not need to issue `CloseResourceFile` calls for all open resource files when quitting. The `ResourceShutDown` call automatically updates and closes any open resource files.

Parameters

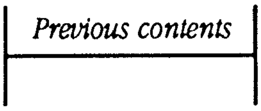
Stack before call



Word—ID of open resource file; NIL to close all open files

<—SP

Stack after call



<—SP

Errors	GS/OS errors	Returned unchanged
	`WriteResource` errors	Returned unchanged

C

```
extern pascal void CloseResourceFile(fileID);

Word      fileID;
```

`CountResources` $221E

Counts the number of resources of a specified type in all resource files available to the calling program in its search sequence. Your program specifies the resource type to be counted. The Resource Manager counts all resources of that type in open resource files available to your program, including the system resource file.

◆ *Note:* This call can be very slow when you have many resources or resource files. Do not issue this call in time-critical procedures.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
resourceType	Word—Resource type to be counted
	<—SP

Stack after call

Previous contents	
– *totalResources* –	Long—Number of resources of specified type
	<—SP

Errors None

C

```
extern pascal Long CountResources(resourceType);

Word      resourceType;
```

`CountTypes` $201E

Counts the number of different resource types in all resource files available to the calling program in its search sequence, including the system resource file.

◆ *Note:* This call can be very slow when you have many resources or resource files. Do not issue this call in time-critical procedures.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
	<—SP

Stack after call

Previous contents	
totalTypes	Word—Number of different resource types
	<—SP

Errors	Memory Manager errors	Returned unchanged
C	`extern pascal Word CountTypes();`	

`CreateResourceFile` $091E

Initializes a resource fork with no resources. If necessary, `CreateResourceFile` will create the file to contain the resource fork. The specific actions performed by this call depend upon the state of the specified input file:

No file of specified name	Create file with specified *auxType*, *fileType*, *fileAccess*, *fileName*. Create and initialize resource fork
File with no resource fork	Create and initialize resource fork
File with empty resource fork	Initialize resource fork
File with non-empty resource fork	Return `resForkUsed` error

Parameters

Stack before call

Previous contents	
– *auxType* –	Long—GS/OS auxiliary file type (used only if file does not exist)
fileType	Word—GS/OS file type (used only if file does not exist)
fileAccess	Word—GS/OS file access (used only if file does not exist)
– *fileName* –	Long—Pointer to GS/OS class 1 input pathname for resource file
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$1E01	`resForkUsed`	Resource fork for specified file is not empty
	GS/OS errors		Returned unchanged

C

```
extern pascal void CreateResourceFile(auxType,
          fileType, fileAccess, fileName);

Long      auxType, fileName;
Word      fileType, fileAccess;
```

`DetachResource` $181E

Instructs the Resource Manager to dispose of its control blocks for a specified resource. The resource itself remains in memory; the calling program is responsible for freeing its handle. The resource to be detached must be marked as unchanged.

This call can be used to copy resources between different resource files. After issuing the `DetachResource`, add the resource to the new resource file by calling `AddResource`. After issuing the `AddResource` call, the Resource Manager is again responsible for the resource handle.

Parameters

Stack before call

Previous contents	
resourceType	Word—Type of resource to be detached
– *resourceID* –	Long—ID of resource to be detached
	<—SP

Stack after call

Previous contents	
	<—SP

Errors			
	$1E06	`resNotFound`	Specified resource cannot be found
	$1E0C	`resHasChanged`	Resource has been changed and has not been updated

C

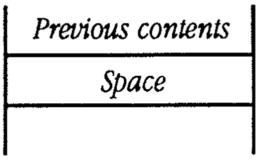
```
extern pascal void DetachResource(resourceType,
                    resourceID);

Word        resourceType;
Long        resourceID;
```

`GetCurResourceApp` $141E

Returns the User ID for the application that is currently using the Resource Manager. If the Resource Manager is not in use, this call returns the Resource Manager's User ID ($401E). This call is used by desk accessories and application switchers (see "Application switchers and desk accessories" earlier in this chapter for more information).

Parameters

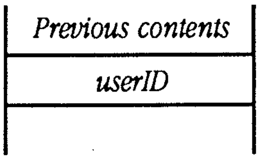
Stack before call



Word—Space for result

<—SP

Stack after call



Word—User ID of current application; $401E if none

<—SP

Errors None

C `extern pascal Word GetCurResourceApp();`

GetCurResourceFile $121E

Returns the file ID of the current resource file. This call returns a NIL value if there is no current file.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
	<—SP

Stack after call

Previous contents	
fileID	Word—File ID of current resource file; NIL if none
	<—SP

Errors	$1E04	`resNoCurFile`	No current'resource file

C `extern pascal Word GetCurResourceFile();`

`GetIndResource` $231E

Finds a resource of a specified type by means of its index, and returns the resource ID for that resource. The index value corresponds to the position of the desired resource among all resources of the specified type in all resource files available to the calling program in its search sequence; the first resource is number 1.

Use this call to find every resource of a given type by repeatedly issuing the call, incrementing the index value until the call returns resIndexRange.

◆ *Note:* This call can be very slow when you have many resources or resource files. Do not issue this call in time-critical procedures.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
resourceType	Word—Type of resource to find
– *resourceIndex* –	Long—Index of resource to find
	<—SP

Stack after call

Previous contents	
– *resourceID* –	Long—ID of resource matching type and index
	<—SP

Errors	$1E0A	`resIndexRange`	Index is out of range (no resource found)
	Memory Manager errors		Returned unchanged

C

```
extern pascal Long GetIndResource(resourceType,
           resourceIndex);

Word       resourceType;
Long       resourceIndex
```

`GetIndType` $211E

Finds a resource type value by means of its index. The index value corresponds to the 1-relative position of the desired resource type among all types in all resource files available to the calling program in its search sequence.

Use this call to find every resource type in all files available to an application by repeatedly issuing the call, incrementing the index value until the call returns resIndexRange.

- ◆ *Note:* This call can be very slow when you have many resources or resource files. Do not issue this call in time-critical procedures.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
typeIndex	Word—Index of type to find
	<—SP

Stack after call

Previous contents	
resourceType	Word—Type matching index
	<—SP

Errors	$1E0A	`resIndexRange`	Index is out of range (no more types)
	Memory Manager errors		Returned unchanged

C

```
extern pascal Word GetIndType(typeIndex);

Word      typeIndex;
```

`GetMapHandle` $261E

Returns a handle to the resource map for a specified resource file. Your program specifies the desired resource file by passing its file ID to `GetMapHandle`. This call searches all open resource files, irrespective of the search sequence in effect.

For information on the format and content of resource file maps, see "Resource file format" earlier in this chapter.

◆ *Note:* This call provides greater application flexibility; however, most applications will not need to issue this call.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
fileID	Word—ID of resource file to find
	<—SP

Stack after call

Previous contents	
– *mapHandle* –	Long—Handle of resource file map; NIL if none found
	<—SP

Errors	$1E07	`resFileNotFound`	Specified file ID does not match an open file

C

```
extern pascal Long GetMapHandle(fileID);

Word      fileID;
```

fileID Specifies the resource file whose map is to be returned. This value is obtained from the OpenResourceFile tool call. Typically, your program sets this parameter with the file ID for a particular resource file. However, this field also supports the following special values:

NIL	returns handle to map for current resource file
$FFFF	returns handle to map for system resource file

`GetOpenFileRefNum` $1F1E

Returns the GS/OS file reference number (`refNum`) associated with the resource fork of an open resource file. Your program specifies the resource file by means of its file ID. The Resource Manager searches all open resource files for a file with a matching ID.

Your program may use this reference number to read data from the resource file. However, your program should very careful to maintain the structure of the fork during write operations; careless writing could destroy the resource fork. Further, your program should never directly close the file using the reference number. Only the Resource Manager should close files it has opened.

For information on the format and content of resource file maps, see "Resource file format" earlier in this chapter.

◆ *Note:* This call provides greater application flexibility; however, most applications will not need to issue this call.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
fileID	Word—ID of resource file to find
	<—SP

Stack after call

Previous contents	
openRefNum	Word—GS/OS file reference number
	<—SP

Errors	$1E07	`resFileNotFound`	Specified file ID does not match an open file

C

```
extern pascal Word GetOpenFileRefNum(fileID);

Word      fileID;
```

fileID	Specifies the resource file whose reference number is to be returned. This value is obtained from the `OpenResourceFile` tool call. Typically, your program sets this parameter with the file ID for a particular resource file. However, this field also supports the following special values:

NIL	returns reference number for current resource file
$FFFF	returns reference number for system resource file

`GetResourceAttr` $1B1E

Returns the attribute flag word for a specified resource. Your program specifies the type and ID for the desired resource. For more information about the format and content of the attribute word, see "Resource attributes" earlier in this chapter.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
resourceType	Word—Type of resource to find
– *resourceID* –	Long—ID of resource to find
	<—SP

Stack after call

Previous contents	
resourceAttr	Word—Attribute word for specified resource
	<—SP

Errors	$1E06	`resNotFound`	Specified resource not found

C

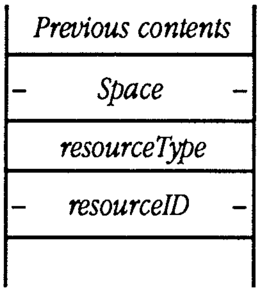
```
extern pascal Word GetResourceAttr(resourceType,
                    resourceID);

Word        resourceType;
Long        resourceID;
```

`GetResourceSize` $1D1E

Returns the size of the specified resource. Your program specifies the type and ID for the desired resource. Resource size is defined as the number of bytes the resource occupies in the resource fork on disk.

Parameters

Stack before call



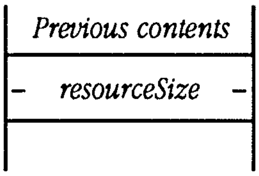
Long—Space for result

Word—Type of resource to find

Long—ID of resource to find

<—SP

Stack after call



Long—Size of specified resource

<—SP

Errors	$1E06	`resNotFound`	Specified resource not found

C

```
extern pascal Long GetResourceSize(resourceType,
                    resourceID);

Word        resourceType;
Long        resourceID;
```

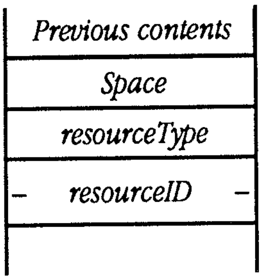
`HomeResourceFile` $151E

Returns the file ID of the resource file that contains a specified resource. Your program specifies the type and ID for the resource in question.

◆ *Note:* If multiple resources share the specified type and ID values, and your program has changed the resource search sequence (with the `SetCurResourceFile` tool call), this call may return a different result than on previous calls.

Parameters

Stack before call



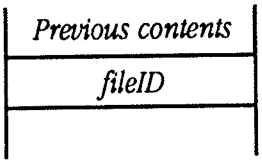
Word—Space for result

Word—Type of resource to find

Long—ID of resource to find

<—SP

Stack after call



Word—File ID for home resource file for resource; NIL if not found

<—SP

Errors	$1E06	`resNotFound`	Specified resource not found

C

```
extern pascal Word HomeResourceFile(resourceType,
              resourceID);

Word          resourceType;
Long          resourceID;
```

`LoadAbsResource` $271E

Loads a resource into a specified absolute memory location. Your program specifies the type and ID of the resource to load, the memory location into which the Resource Manager is to load the resource, and the maximum number of bytes to load. Note that the `resAbsLoad` flag in the Attributes word for the desired resource must be set to 1.

◆ *Note:* This call does not respect the disk load setting maintained by the `SetResourceLoad` tool call.

▲ **Warning** Most applications will not have to issue this call. In order to use this call you must have a thorough understanding of using absolute memory. Issuing this call with an incorrectly set *loadAddress* parameter will corrupt system memory. ▲

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
– *loadAddress* –	Long—Address at which to load resource
– *maxSize* –	Long—Maximum number of bytes to load
resourceType	Word—Type of resource to find
– *resourceID* –	Long—ID of resource to find
	<—SP

Stack after call

Previous contents	
– *resourceSize* –	Long—Size of resource on disk
	<—SP

Errors

$1E03	`resNoConverter`	No converter routine to load resource
$1E06	`resNotFound`	Specified resource not found
GS/OS errors		Returned unchanged

C

```
extern pascal Long LoadAbsResource(loadAddress,
           maxSize, resourceType, resourceID);

Word       resourceType;
Long       loadAddress, maxSize, resourceID;
```

loadAddress Specifies the memory location at which the Resource Manager is to load the resource. If your program passes a NIL value, the Resource Manager uses the address stored in the `resHandle` field of the appropriate entry in the resource index.

`LoadResource` $0E1E

Loads a resource into memory and returns a handle to that location. Your program specifies the type and ID of the resource to load. The returned handle provides addressability to the resource.

The `LoadResource` call searches both memory and disk for the specified resource. If the resource is already in memory, `LoadResource` returns a handle to that memory location. If the resource has been purged from memory, `LoadResource` reloads the resource and returns its handle. If the resource has not been loaded, `LoadResource` allocates a handle, loads the resource, and returns the handle to your program.

Your program may manipulate the resource while it is in memory, and may even change the size of the resource (to any size other than zero bytes). If you want the changes to be reflected in the resource file, use the `MarkResourceChange` tool call to set on the changed attribute for the file. The Resource Manager will then write the changed resource to disk the next time the resource file is updated. Your program can force the Resource Manager to write the resource to disk immediately by issuing either the `WriteResource` or the `UpdateResourceFile` tool call.

Note that your program should not dispose of the handle; only the Resource Manager should free the memory that it allocates.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
resourceType	Word—Type of resource to find
– *resourceID* –	Long—ID of resource to find
	<—SP

Stack after call

Previous contents	
– *resourceHandle* –	Long—Handle of resource in memory
	<—SP

Errors

$1E03	`resNoConverter`	No converter routine to load resource
$1E06	`resNotFound`	Specified resource not found
GS/OS errors		Returned unchanged
Memory Manager errors		Returned unchanged

C

```
extern pascal Long LoadResource(resourceType,
          resourceID);

Word      resourceType;
Long      resourceID;
```

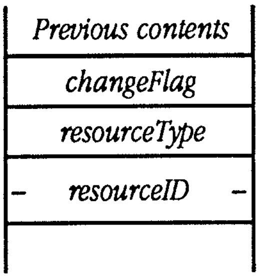
`MarkResourceChange` $101E

Instructs the Resource Manager to write the specified resource to disk the next time its resource file is updated. Your program specifies the type and ID of the resource to be marked as changed.

Use this call when you want in-memory changes to a resource to be permanent.

Parameters

Stack before call



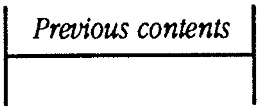
Word—BOOLEAN; TRUE for changed, FALSE for not changed

Word—Type of resource to find

Long—ID of resource to find

<—SP

Stack after call



<—SP

Errors	$1E06	`resNotFound`	Specified resource not found

C

```
extern pascal void MarkResourceChange(changeFlag,
            resourceType, resourceID);

Word        changeFlag, resourceType;
Long        resourceID;
```

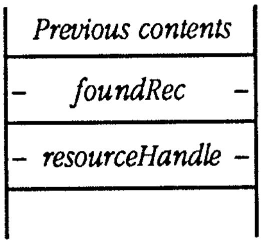
`MatchResourceHandle` $1E1E

Returns the type and ID of a resource, given a handle to that resource. The Resource Manager searches all open resource files for a match, without regard for the search sequence in effect.

◆ *Note:* The Resource Manager has been optimized to access resources by type and ID, irrespective of the number of resources in the system. While `MatchResourceHandle` works well with relatively small numbers of resources (less than 100), for files with large numbers of resources this call can be very slow. In order to avoid this overhead, consider storing the resource type and ID in the resource structure, so that your program can directly access this information.

Parameters

Stack before call

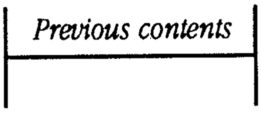


Long—Pointer to location in which to return type and ID

Long—handle of resource

<—SP

Stack after call



<—SP

Errors $1E06 `resNotFound` Specified resource not found

C

```
extern pascal void MatchResourceHandle(foundRec,
                    resourceHandle);

Pointer     foundRec;
Long        resourceHandle;
```

foundRec Must point to a location in memory that can accept 6 bytes of data: the type and ID of the resource in question. On successful return from `MatchResourceHandle`, that location will contain the following data:

$00	`resourceType`	Word—Type of resource
$02	`resourceID`	Long—ID of resource

`OpenResourceFile` $0A1E

Opens a specified resource file, making it the current file, and returns a unique file ID to the calling program. Your program specifies the class 1 GS/OS pathname to the desired resource file. The Resource Manager loads the resource map into memory, along with any resources marked to be pre-loaded (`resPreLoad` flag is set to 1 in the Attributes word for the resource).

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
openAccess	Word—File access
– *mapAddress* –	Long—Address of resource map in memory
– *fileName* –	Long—Pointer to GS/OS class 1 pathname for resource file
	<—SP

Stack after call

Previous contents	
fileID	Word—ID of open resource file
	<—SP

Errors

$1E06	`resNotFound`	Specified resource not found
$1E09	`resNoUniqueID`	Too many resource files open
$1E0B	`resSysIsOpen`	System resource file is already open
GS/OS errors		Returned unchanged (EOF if empty fork)
Memory Manager errors		Returned unchanged

C

```
extern pascal Word OpenResourceFile(openAccess,
                    mapAddress, fileName);

Word        openAccess;
Pointer     mapAddress, fileName;
```

openAccess	Contains GS/OS file access privileges for the resource file. See the *GS/OS Reference* for more information.
mapAddress	To open a resource file on disk, set this field to NIL. If the map is in memory, load this field with a pointer to that map. In this case, the Resource Manager opens the file that is already in memory.

`ReleaseResource` $171E

Sets the purge level of the memory used by a resource. Your program specifies the type and ID of the resource whose memory is to be freed, and the purge level to be assigned to the memory. See Chapter 12, "Memory Manager," in Volume 1 of the *Toolbox Reference* for more information about purge levels and memory management. Note that this call does not unlock the handle.

Parameters

Stack before call

Previous contents	
purgeLevel	Word—Purge level for memory
resourceType	Word—Type of resource to find
– *resourceID* –	Long—ID of resource to find
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$1E06	`resNotFound`	Specified resource not found
$1E0C	`resHasChanged`	Resource has been changed but not updated

C

```
extern pascal void ReleaseResource(purgeLevel,
                    resourceType, resourceID);

Word        purgeLevel, resourceType;
Long        resourceID;
```

purgeLevel Specifies the Memory Manager purge level to be assigned to the freed memory. Valid Memory Manager purge levels lie in the range of 0 to 3. In order to direct the Resource Manager to immediately dispose of the handle, set this field to a negative value.

RemoveResource $0F1E

Deletes a resource from its resource file and releases any memory used by the resource. Your program specifies the type and ID of the resource to be deleted. After successful return from this call, the specified resource is no longer available for access or loading.

Parameters

Stack before call

Previous contents	
resourceType	Word—Type of resource to find
– *resourceID* –	Long—ID of resource to find
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$1E06	`resNotFound`	Specified resource not found
$1E0E	`resDiskFull`	Volume full

C

```
extern pascal void RemoveResource(resourceType,
          resourceID);

Word      resourceType;
Long      resourceID;
```

`ResourceConverter` $281E

Installs or removes a converter routine from either the application or system converter list. Your program specifies the address of the converter routine, the type of resource the routine acts on, and flags indicating the type of operation to perform and the list to modify. For background information on resource converter routines, see "Resource converter routines" earlier in this chapter.

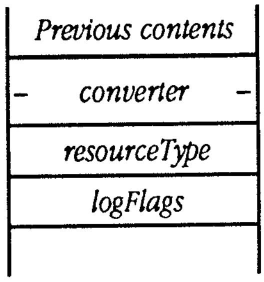
The Resource Manager maintains two classes of converter routine lists: one for your application and one for the system. Each application has its own converter routine list. All programs share access to the system list. When the Resource Manager searches for a routine to convert a resource of a given type, it first searches the application list for the calling program, then the system list. As a result, your program can override converter routines in the system list by installing a routine for the same resource type in its application list. Applications must never log routines into or out of the system converter list.

An application can log in up to 10,922 converter routines. Note, however, that the Resource Manager does not check for this limit. The same converter routine can be logged in for more than one resource type.

The system contains a standard routine to convert code resources. Use the `GetCodeResConverter` Miscellaneous Tool Set tool call to obtain the address of that routine (see Chapter 39, "Miscellaneous Tool Set Update," in this book for details on the `GetCodeResConverter` call).

Parameters

Stack before call



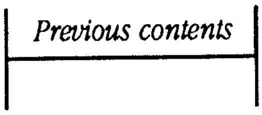
Long—Pointer to converter routine

Word—Type of resource acted on by the routine

Word—Flag governing action and list to access

<—SP

Stack after call



<—SP

Errors	$1E0D	`resDiffConverter`	Another converter already logged in for this resource type
	Memory Manager errors		Returned unchanged

C

```
extern pascal void ResourceConverter(converter,
          resourceType, logFlags);

Pointer   converter;
Word      resourceType, logFlags;
```

logFlags Specifies whether to log the converter routine into or out of its list. Also specifies which list (application or system) to access:

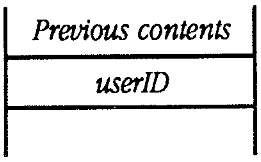
Reserved	Bits 2–15	Must be set to 0
`list`	Bit 1	Indicates which routine list to access: 1 - System converter list 0 - Application converter list
`action`	Bit 0	Specifies action to take: 1 - Log routine into list 0 - Log routine out of list

`SetCurResourceApp` $131E

Tells the Resource Manager that another application will now be issuing Resource Manager calls. This call is used by desk accessories and application switchers (see "Application switchers and desk accessories" earlier in this chapter for more information). Before issuing this call, your program must have registered itself with the Resource Manager by calling `ResourceStartUp`.

Parameters

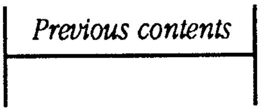
Stack before call



Word—User ID of application that will be using Resource Manager

<—SP

Stack after call



<—SP

Errors	$1E08	`resBadAppID`	User ID unknown to Resource Manager (has not called `ResourceStartUp`)

C

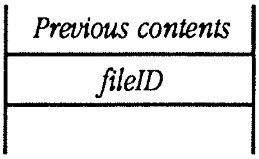
```
extern pascal void SetCurResourceApp(userID);

Word      userID;
```

`SetCurResourceFile` $111E

Makes a specified resource file the current file. The Resource Manager searches typically start with the current resource file, so by specifying a particular file as the current file, your program can control the file search sequence. For more information about Resource Manager search processing, see "Using resources" earlier in this chapter.

Parameters

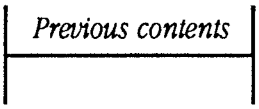
Stack before call



Word—File ID of resource file to be made current file

<—SP

Stack after call



<—SP

Errors $1E07 `resFileNotFound` Specified resource file not found

C

```
extern pascal void SetCurResourceFile(fileID);

Word      fileID;
```

`SetResourceAttr` $1C1E

Sets the attributes for a resource. Your program specifies the type and ID for the desired resource and a new attribute word for the resource. The Resource Manager replaces the existing attribute word for the resource with the one provided to this call. For more information about the format and content of the attribute word, see "Resource attributes" earlier in this chapter.

If your program changes the attributes of a resource, it should not also mark the resource as changed. The Resource Manager automatically tracks these changes.

Note that these changes only affect future use of the resource. For example, if your program changes the attributes for a resource to indicate that it should be locked into memory (sets `attrLocked` flag to 1), that action does not change the status of any current instances of that resource in memory. However, the next time the Resource Manager allocates a handle for the resource, the memory for that new handle will be locked.

Parameters

Stack before call

Previous contents	
resourceAttr	Word—New attribute flag word for resource
resourceType	Word—Type of resource to find
– *resourceID* –	Long—ID of resource to find
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$1E06	`resNotFound`	Specified resource not found

C

```
extern pascal void SetResourceAttr(resourceAttr,
            resourceType, resourceID);

Word        resourceAttr, resourceType;
Long        resourceID;
```

`SetResourceFileDepth` $251E

Sets the number of files Resource Manager is to search during a search operation, and returns the previous search depth setting. For more information about the Resource Manager's search sequence, see "Resource file search sequence" earlier in this chapter.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
searchDepth	Word—Number of files to search
	<—SP

Stack after call

Previous contents	
originalDepth	Word—Search file depth before call
	<—SP

Errors None

C

```
extern pascal Word
                    SetResourceFileDepth(searchDepth);

Word        searchDepth;
```

searchDepth Specifies the number of files to search. `SetResourceFileDepth` accepts the following special values:

NIL	Return current search depth without changing it
$FFFF	Search all files

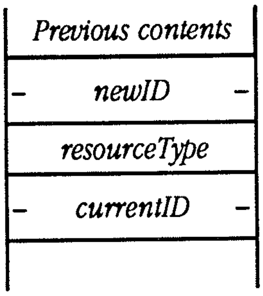
`SetResourceID` $1A1E

Changes the ID for a resource to a new value. Your program specifies the type and current ID for the resource to be changed.

If your program changes the ID value for a resource, it should not mark the resource as changed. The Resource Manager automatically tracks these changes.

Parameters

Stack before call



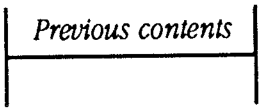
Long—New ID for resource

Word—Type of resource to find

Long—Current ID of resource to find

<—SP

Stack after call



<—SP

Errors			
	$1E05	`resDupID`	Specified new ID already in use for this type
	$1E06	`resNotFound`	Specified resource not found

C

```
extern pascal void SetResourceID(newID,
              resourceType, currentID);

Long      newID, current ID;
Word      resourceType;
```

`SetResourceLoad` $241E

Controls Resource Manager access to disk when loading resources. If you disable disk loading, the Resource Manager will not load resources from disk, but will allocate empty handles for requested resources. However, if a resource had been loaded into memory prior to disk loading being disabled, the Resource Manager will return a valid handle. For example, a `LoadResource` tool call will return an empty handle if loading is set to FALSE and the resource has not been loaded into memory previously.

◆ *Note:* Most applications will not issue this call.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
readFlag	Word—Flag controlling Resource Manager disk access
	<—SP

Stack after call

Previous contents	
originalFlag	Word—Flag setting prior to call
	<—SP

Errors None

C

```
extern pascal Word SetResourceLoad(readFlag);

Word       readFlag;
```

readFlag Specifies the new setting for the read flag. This call also supports a special value that just returns the current flag setting:

0	Do not read resources from disk
1	Read resources from disk, if necessary
Negative	Return current setting only—no change to current setting

originalFlag Contains the previous flag setting:

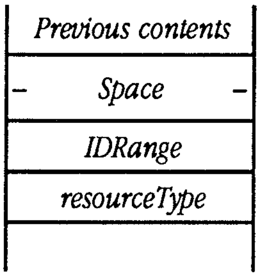
0	Do not read resources from disk
1	Read resources from disk, if necessary

`UniqueResourceID` $191E

Returns a unique resource ID for a specified resource type. Your program specifies the resource type for the ID, and may optionally constrain the new ID to a defined range. The Resource Manager allocates the new ID, guaranteeing that it is not used by any of your program's currently available resources.

Parameters

Stack before call



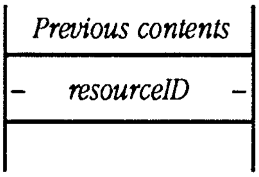
Long—Space for result

Word—Range of ID; $FFFF for any valid ID value

Word—Type of resource

<—SP

Stack after call



Long—Unique resource ID

<—SP

Errors			
	$1E04	`resNoCurFile`	There is no current resource file
	$1E09	`resNoUniqueID`	No unique ID found

C

```
extern pascal Long UniqueResourceID(IDRange,
                    resourceType);

Word        IDRange, resourceType;
```

IDRange Specifies a 65,535 element range within which Resource Manager is to allocate the new resource ID. The value of *IDRange* becomes the high-order word of the new ID. The Resource Manager then allocates a unique ID from the 65,535 possible remaining values. `UniqueResourceID` provides this facility so that application programs can manage logical groups of resources, differentiated by ID number ranges.

Resource IDs in the $07FF range are reserved for system use. Ranges from $0800 through $FFFE are invalid. The following table summarizes the valid values for *IDRange:*

IDRange	**Lowest possible ID returned**	**Highest possible ID returned**
$0000	$00000001 (NIL is invalid)	$0000FFFF
$0001	$00010000	$0001FFFF
$0002	$00020000	$0002FFFF
.		
.	(and so on)	
.		
$07FE	$07FE0000	$07FEFFFF
$07FF	Reserved for system use	
$0800-$FFFE	Invalid range values	
$FFFF	$00000001 (directs Resource Manager to allocate from any application range)	$07FEFFFF

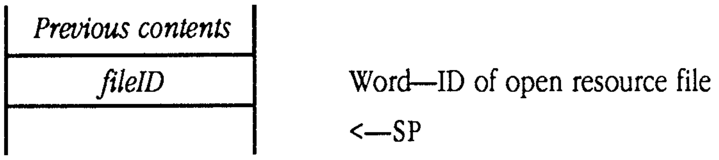
`UpdateResourceFile` $0D1E

Applies any modifications that have been made to resources in memory to their resource file, thus making those changes permanent. Your program specifies the file ID of the resource file to be updated. The Resource Manager then locates and updates all resources for that file. If necessary, `UpdateResourceFile` will write the resource map to disk.

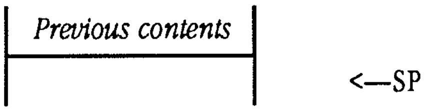
◆ *Note:* Most applications will not issue this call because the `ResourceShutDown` tool call automatically updates all resources opened by a program.

Parameters

Stack before call



Stack after call



Errors	$1E03	`resNoConverter`	No converter routine found for resource type
	$1E07	`resFileNotFound`	No resource file found with specified file ID
	$1E0E	`resDiskFull`	Volume full
	GS/OS errors		Returned unchanged

C

```
extern pascal void UpdateResourceFile(fileID);

Word      fileID;
```

`WriteResource` $161E

Directs the Resource Manager to write a modified resource to its resource file. Your program specifies the type and ID of the resource. If that resource has been modified (`resChanged` flag set to 1 in the Attributes flag word), the Resource Manager writes the resource to its resource file on disk. A resource is marked changed as a result of `AddResource`, `MarkResourceChange`, or `SetResourceAttr` (with `resChanged` set to 1) tool calls.

◆ *Note:* Most applications will not issue this call because the `ResourceShutDown` and `CloseResourceFile` tool calls automatically write all changed resources to the appropriate resource file (unless the resource is write-protected).

Parameters

Stack before call

Previous contents	
resourceType	Word—Type of resource to write
– *resourceID* –	Long—ID of resource to write
	<—SP

Stack after call

Previous contents	
	<—SP

Errors			
	$1E03	`resNoConverter`	No converter routine found for resource type
	$1E06	`resNotFound`	Resource not found
	$1E0E	`resDiskFull`	Volume full
	GS/OS errors		Returned unchanged

C

```
extern pascal void WriteResource(resourceType,
                    resourceID);

Word            resourceType;
Long            resourceID;
```

Resource Manager summary

This section briefly summarizes the constants, data structures, and error codes used by the Resource Manager.

■ **Table 45-1** Resource Manager constants

Name	Value	Description
mapFlag values		
`mapChanged`	$0002	Set to 1 if the map has changed and must be written to disk
resAttr flag values		
`resChanged`	$0020	Set to 1 if the resource has changed and must be written to disk
`resPreLoad`	$0040	Set to 1 if the resource should be loaded into memory by `OpenResourceFile`
`resProtected`	$0080	Set to 1 if the resource should never be written to disk
`resAbsLoad`	$0400	Set to 1 if the resource should be loaded at an absolute memory location
`resConverter`	$0800	Set to 1 if the resource requires a converter routine when being loaded into memory or being written to disk
`resMemAttr`	$C31C	Flags passed to the `NewHandle` Memory Manager tool call when allocating memory for the resource
System file ID		
`sysFileID`	$0001	File ID of the system resource file

■ **Table 45-2** Resource Manager data structures

Name	Offset/Value	Type	Description
ResHeaderRec (resource file header record)			
rFileVersion	$0000	LongWord	Format version of resource fork
rFileToMap	$0004	LongWord	Offset from start of fork to resource
map record			
rFileMapSize	$0008	LongWord	Size, in bytes, of resource map
rFileMemo	$000C	128 bytes	Space reserved for application use
rFileRecSize	$008C		Size of ResHeaderRec
MapRec (resource map record)			
mapNext	$0000	Handle	Handle of next resource map in memory
mapFlag	$0004	Word	Bit flags
mapOffset	$0006	LongWord	Offset from start of fork to resource map record
mapSize	$000A	LongWord	Size, in bytes, of resource map
mapToIndex	$000E	Word	Offset from start of map to mapIndex
mapFileNum	$0010	Word	GS/OS file reference number for open resource file
mapID	$0012	Word	Resource Manager file ID assigned to this resource file
mapIndexSize	$0014	LongWord	Total number of resource reference records in mapIndex
mapIndexUsed	$0018	LongWord	Number of used resource reference records
mapFreeListSize	$001C	Word	Total number of free block records in mapFreeList
mapFreeListUsed	$001E	Word	Number of used free block records
mapFreeList	$0020	n bytes	Array of free block records (FreeBlockRec)
mapIndex	$0020+n	m bytes	Array of resource reference records (ResRefRec)

FreeBlockRec (free block record)

blkOffset	$0000	LongWord	Offset, in bytes, to start of this block of free space
blkSize	$0004	LongWord	Size, in bytes, of this block of free space
blkRecSize	$0008		Size of FreeBlockRec

ResRefRec (resource reference record)

resType	$0000	Word	Resource type
resID	$0002	LongWord	Resource ID
resOffset	$0006	LongWord	Offset, in bytes, from start of resource fork to this resource
resAttr	$000A	Word	Attribute bit flags for the resource
resSize	$000C	LongWord	Size, in bytes, of the resource in the resource fork
resHandle	$0010	Handle	Handle of resource in memory
resRecSize	$0014		Size of ResRefRec

■ **Table 45-3** Resource Manager error codes

Code	Name	Description
$1E01	`resForkUsed`	Resource fork not empty
$1E02	`resBadFormat`	Resource fork not correctly formatted
$1E03	`resNoConverter`	No converter routine available for resource type
$1E04	`resNoCurFile`	There are no open resource files
$1E05	`resDupID`	Specified resource ID is already in use
$1E06	`resNotFound`	Resource was not found
$1E07	`resFileNotFound`	Resource file was not found
$1E08	`resBadAppID`	User ID not found; calling program has not issued `ResourceStartUp` tool call
$1E09	`resNoUniqueID`	No more resource IDs available
$1E0A	`resIndexRange`	Index is out of range
$1E0B	`resSysIsOpen`	System file is already open
$1E0C	`resHasChanged`	Resource marked changed; specified operation not allowed
$1E0D	`resDiffConverter`	Different converter already logged in for this resource type
$1E0E	`resDiskFull`	Volume full

Chapter 46 **Scheduler**

There are no changes in the Scheduler. The complete reference for the Scheduler is in Volume 2, Chapter 19 of the *Apple IIGS Toolbox Reference.*

Chapter 47 **Sound Tool Set Update**

This chapter documents new features of the Sound Tool Set. The complete reference to the Sound Tool Set is in Volume 2, Chapter 21 of the *Apple IIGS Toolbox Reference.*

- ◆ *Note:* You must read the *Apple IIGS Hardware Reference* to understand some of the concepts presented in this chapter.

Error corrections

This section provides corrections to the documentation of the Sound Tool Set in the *Apple IIGS Toolbox Reference.*

- The documentation of the `FFSoundDoneStatus` call includes an error. You will note that the paragraph that describes the call does not agree with the "Stack after call" diagram. The text states that the call returns TRUE if the specified sound is still playing, whereas the diagram states that it returns FALSE if still playing. The diagram, not the text, is correct.
- There is an undocumented distinction between a generator that is playing a sound and one that is active. A generator that is playing a sound returns FALSE in response to an `FFSoundDoneStatus` call. One that is active may or may not be playing a sound; the value of the flag returned by `FFSoundStatus` is TRUE. Active generators are those that are allocated to a voice. At any given moment the generator may be playing a sound, and so the `FFSoundDoneStatus` returns FALSE—or it may be silent between notes, in which case `FFSoundDoneStatus` returns TRUE.

Clarification

This section presents more complete information about the `FFStartSound` tool call, including further explanation of its parameters, a new error code, an example procedure for moving a sound from the Macintosh to the Apple IIGS and some sample code demonstrating the use of the call. The original documentation for this call is in Chapter 20, "Sound Tool Set," in Volume 2 of the *Toolbox Reference.*

`FFStartSound`

The freeform synthsizer is designed to play back long wave forms. In order to handle longer waveforms the synthesizer uses two buffers (which must be the same size), alternating its input from one to the other. When the synthesizer exhausts a buffer, it generates an interrupt and then starts reading data from the other buffer. The Sound Tool Set services the interrupt and begins refilling the empty buffer. This process continues until the waveform has been completely played.

Note that all synthesizer input buffers must be buffer-size aligned. That is, if you have allocated 4K buffers, then those buffers must be aligned on 4K memory boundaries.

Parameter Block

$00	`waveStart`	Long
$04	`waveSize`	Word
$06	`freqOffset`	Word
$08	`docBuffer`	Word
$0A	`bufferSize`	Word
$0C	`nextWavePtr`	Long
$10	`volSetting`	Word

`waveStart` The starting address of the wave to be played, not in DOC RAM but in Apple IIGS system RAM. The Sound Tool Set loads the waveform data into DOC RAM as it is played.

`waveSize` The size in pages of the wave to be played. A value of 1 indicates that the wave is one page (256 bytes) in size; a value of 2 indicates that it is two pages (512 bytes) in size—and so on as you might expect. The only anomaly is that a value of 0 specifies that the wave is 65,536 pages in size.

`freqOffset` This parameter is copied directly into the Frequency High and Frequency Low registers of the DOC. See the previous discussion of those registers for more complete information.

`docBuffer` Contains the address in Sound RAM where buffers are to be allocated. This value is written to the DOC WaveForm Table Pointer register. The low-order byte is not used, and should always be set to 0.

`bufferSize` The lowest three bits set the values for the table-size and resolution portions of the DOC Bank-Select/Table-size/Resolution register. See the previous discussion of that register for details.

`nextWavePtr` This is the address of the next waveform to be played. If the field's value is 0, then the current waveform is the last waveform to be played.

`volSetting` The low byte of the `volSetting` field is copied directly into the Volume register of the DOC. All possible byte values are valid.

New error code

$0817	`IRQNotAssignedErr`	No master IRQ was assigned.

Moving a sound from the Macintosh to the Apple IIGS

To move a digitized sound from the Macintosh to the Apple IIGS and play the sound, you will have to perform the following steps

1. Save the sound as a pure data file on the Macintosh.
2. Transfer the file to the Apple IIGS (using Apple File Exchange, for example).
3. Filter all the zero sample bytes out of the file by replacing them with bytes set to $01. This is very important, because the Apple IIGS interprets zero bytes as the end of a sample.
4. Load the sound into memory with GS/OS calls.
5. Play the sound with the `FFStartSound` tool call.

   Set the `freqOffset` parameter to $01B7 in order to play the sound at the same tempo as the Macintosh.

Sample code

This assembly-language code sample demonstrates the use of the `FFStartSound` tool call.

```
            PushWord    chanGenType         ; set generator for FFSynth
            PushLong    #STParamBlk         ; address of parm block
            _FFStartSound                   ; start free-from synth

ChanGenType DC.W $0201                      ; generator 2, FFSynth

STParamBlk  DS.L 1                          ; store the address of the
                                            ;  sound in system memory here
            Entry       WaveSize
WaveSize    DS.W 1                          ; store the number of pages to
                                            ;  play here
Freq        DC.W $200                       ; A9 set for each sample once
Start       DC.W $8000                      ; start at beginning
Size        DC.W $6                         ; 16k buffers
Nxtwave     DC.L $0                         ; no new param block
Vol         DC.W $FF                        ; maximum volume
```

New information

This section provides new information about the Sound Tool Set.

- The four sound and music tools, that is, the Note Sequencer, Note Synthesizer, MIDI Tool Set, and Sound Tool Set, work together and must have compatible versions. The current required versions are:

Sound Tool Set	version 2.4
Note Synthesizer	version 1.3
Note Sequencer	version 1.3
MIDI Tool Set	version 1.2

- The Sound Tool Set `SoundBootInit` call has been changed to initialize the `MidiInitPoll` vector ($E101B2) to an `RTL`.
- The `SetUserSoundIRQV` tool call allows you to establish a custom synthesizer interrupt handler. In addition to the description in the *Toolbox Reference*, note that your interrupt handler should verify that it should handle the interrupt by checking the synthesizer mode value, which is passed as an input parameter to the interrupt handler in the accumulator register.

  If your routine does not process the interrupt, it should make a `JMP` to the next routine in the interrupt chain, taking care to preserve the state of the accumulator. If your routine does process the interrupt, it should set the carry flag to 0 and return via an `RTL` instruction.

Introduction to sound on the Apple IIGS

This section provides some general background on the various sound-related tool sets available on the Apple IIGS. There are five sound tool sets: the Sound Tool Set, the Note Sequencer, the Note Synthesizer, the MIDI Tool Set, and the Audio Compression and Expansion (ACE) Tool set. Although each provides distinct functionality, they can complement one another and generate fairly sophisticated sound applications.

- The **Sound Tool Set** will play back a digitized sample of any length and at any frequency. Note that the sample must fit into system memory.
- The **Note Synthesizer** also plays digitized samples, but with much greater control over the sample, including the ability to loop within the sample and control the sound envelope, but is limited to sounds smaller than 65,536 bytes.
- The **MIDI Tool Set** allows you to send and receive MIDI data.
- The **Note Sequencer** combines the functionality of the Note Synthesizer and MIDI Tool Set, allowing you to send MIDI data and drive the Note Synthesizer simultaneously.
- The **Audio Compression and Expansion Tool Set** provides dramatic reduction in sound disk-storage requirements, with only slight degradation in sound quality.

By combining the facilities offered by these tools, you can easily build impressive sound applications. For example, you could develop a program that reads MIDI data into the Note Synthesizer while also saving that data to disk for later input to the Note Sequencer. This program would turn the Apple IIGS into a MIDI sound source with the capaibility to save its songs for later playback.

Note Sequencer

The DOC interrupts that drive the Note Synthesizer also drive the Note Sequencer. Before the Note Synthesizer handles an interrupt, it passes it to the Note Sequencer and allows other interrupt handlers access to it before taking control. The Note Sequencer checks its increment value against its clock value to determine whether to take any action. If enough time has passed, it checks for delay; if a delay is specified, it checks to determine whether it has waited long enough to satisfy the delay requirement. If it hasn't, it simply returns. If it has waited long enough, then it checks all playing notes with specified durations to determine whether it is time to turn them off. If so, it turns those notes off. It then parses the next seqItem in the current sequence and makes Note Synthesizer and MIDI Tool Set calls to execute it. If the `chord` bit is set in the current seqItem, it immediately fetches the next seqItem for execution. If the delay bit is set, then it calculates the required delay and sets the delay flag. It then returns.

Note Synthesizer

One DOC oscillator drives the Note Synthesizer and the Note Sequencer, using the interrupts that it generates at the end of waveforms, or at 0 values in the waveform. The Sound Tool Set services such interrupts, then passes them to the Note Synthesizer for further handling if it is needed. Because the Sound Tool Set and the Note Synthesizer use the same direct-page space, it is appropriate to use the Note Synthesizer to assign oscillators for your own purposes even if you don't use the Note Synthesizer any further with the assigned oscillators.

The Note Synthesizer's operation requires considerable processing. If processor time is in short supply and you want to use the Note Synthesizer to produce sounds, use envelopes that are simple and do not use vibrato, and use low `updateRate` values. See Chapter 41, "Note Synthesizer," in this book for further information.

The Note Synthesizer and Note Sequencer run at interrupt time, and current versions are fully compatible with the MIDI Tool Set.

Sound General Logic Unit (GLU)

One quirk of the sound general logic unit (GLU) is that the value for volume in the control register is a write-only value. It is possible, however, to maintain the system volume specified by the Control Panel setting and still write to the GLU. To find the system volume setting, use the Miscellaneous Tool Set's `GetAddr` call to find the address of `IRQ.Volume` and use the value stored at that address.

Vocabulary

This section describes a number of terms that have special meanings in the context of the Apple IIGS Digital Oscillator Chip (DOC).

Oscillator

There are 32 **oscillators** on the DOC. They are not true oscillators in the ordinary sense of a circuit that generates a waveform. Rather, they are circuits that take as input a waveform stored as digital data, and output an audio signal based on those data.

Generator

Each generator used by the Sound Tool Set is actually a pair of DOC oscillators, usually operating in swap mode when used by the Sound Tool Set. In swap mode the two oscillators alternate playing and halting, with each oscillator playing while the other is halted. When one oscillator reaches the end of its current waveform, it stops playing and the other oscillator takes over, until it reaches the end of its waveform and the first oscillator takes over again.

Voice

A single audio signal that can be independently controlled. A synthesizer that can play eight notes at one time is normally said to have eight voices.

Sample rate

A waveform is stored in the Apple IIGS computer's memory as some number of digital samples of a sound. The number of samples that the Apple IIGS plays each second is referred to as the **sample rate.** The sample rate of the DOC is fixed by the number of oscillators that are enabled, that is, by the value of register $E1 on the DOC. The sample rate depends only upon this value; changing other parameters does not affect sample rate.

$$S = \frac{\left(\frac{C}{8}\right)}{(O+2)}$$

where

S is the sample rate
C is the input clock rate, which is always 7.159 MHz
O is the number of oscillators enabled (32 is standard)

The default sample rate, with all 32 oscillators enabled, is about 26.31985 kHz; that is to say, the Apple IIGS, operating at its default sample rate, plays about 26,320 samples per second. It is possible to generate higher sampling rates by reducing the number of enabled oscillators. However, the low-pass filter on the Apple IIGS is a 5-pole Chebyshev active filter with a roll-off at 10 KHz. Consequently, higher sampling rates may not result in higher perceived sound quality.

Drop sample tuning

The DOC plays waveforms by looking up wave data in a table in memory and sweeping through a stored waveform. This strategy has the advantage that it can produce very faithful reproductions of digitally sampled sound. If, however, you want the DOC to play a waveform at a pitch different from that at which it was recorded, it cannot simply generate it at a different frequency as a true voltage-controlled oscillator can. Instead, the DOC changes the pitch by using a method called **drop sample tuning.** To raise the pitch of a sample one octave, the DOC doubles its frequency by skipping every other sample in the sequence. Similarly, to lower the pitch one octave, it cuts the frequency in half by playing each sample in the sequence twice.

The disadvantage of drop sample tuning is that at higher frequencies, some of the samples are dropped, or lost, and changing the pitch also changes the duration of each waveform.

Frequency

Frequency refers both to the output frequency of the audio signal generated by the DOC and to the value of the DOC frequency register. We normally mean the value of the rrequency register, which determines but is not equal to, the output frequency. Frequency directly determines the perceived pitch of a sound; higher frequencies are higher pitches.

Sound RAM

The DOC has 64 KB of random-access memory dedicated to storage of sound samples. This RAM, which contains the sampled waveforms the DOC plays, is referred to as *Sound RAM.*

Waveform

A waveform consists of data representing the stored form of a digitally sampled audio signal.

DOC registers

There are ten different registers in the DOC. There is a set of registers for each of the DOC oscillators. That is, each of the first seven registers has 32 different values, one for each DOC oscillator. The registers are Frequency Low, Frequency High, Volume, Waveform Data Sample, Waveform Table Pointer, Control, Bank Select/Table-Size/Resolution, Oscillator Interrupt, Oscillator Enable, and A/D Converter.

Frequency registers

Two 8-bit frequency registers, Frequency Low and Frequency High, are paired to produce a single 16-bit frequency value. The output frequency of a sample can be represented by

$$O = \left(\frac{S}{2^{(17+R)}}\right) * \mathrm{FHL}$$

where

O is the output frequency in hertz, assuming that one cycle of the sound exactly fills the table size
S is the sample rate (26.32 KHz)
R is the resolution value in the Bank-Select/Table-Size/Resolution register; valid values lie in the range from 0 through 7
FHL is the combined values of Frequency Low and Frequency High; valid values lie in the range from 0 through 65,535

This calculation assumes that the wave table contains exactly one cycle of the waveform. The resolution and the table size are 3-bit values, and this calculation assumes they are equal.

If one cycle of the sound does not exactly fill the table size, then you can use the following formula to calculate the output frequency:

$$0 = \left(\frac{Fi}{SRi}\right) * \left(\frac{S * \mathrm{FHL}}{2^{(9+R-TAB)}}\right)$$

where

O is the output frequency in hertz
Fi is the frequency of the sampled waveform in hertz
SRi rate at which you sampled the original sound (in samples per second)
S is the Apple IIGS sample rate (26.32 KHz)
FHL is the combined values of Frequency Low and Frequency High; valid values lie in the range from 0 through 65,535
R is the resolution value in the Bank-Select/Table-Size/Resolution register; valid values lie in the range from 0 through 7
TAB Table-size value in the Bank-select/Table-size/Resolution register; valid values lie in the range from 0 through 7

Volume register

The value in the Volume register directly controls the volume of the sound output for that oscillator.

Waveform Data Sample register

This is a read-only register that always contains the value of the sample that an oscillator is currently playing.

Waveform Table Pointer register

This register is also referred to as the Address Pointer register. It identifies which page of Sound RAM contains the start of the current sample. The `FFStartSound` parameter *docBuffer* is written directly to this register.

Control register

The Control register establishes several attributes of its associated oscillator. These attributes include the oscillator's mode, whether the oscillator is halted, whether it will generate an interrupt at the end of its cycle, and the channel to which the oscillator is assigned.

- **Interrupt enable bit** Bit 3 of the Control register is the interrupt enable bit. When this bit is set to 1, the oscillator generates an interrupt when it reaches the end of a waveform or plays a sample with a value of 0.

  Unless you have issued the `SetSoundMIRQV` tool call to set a custom interrupt vector (see Chapter 20, "Sound Tool Set," in Volume 2 of the *Toolbox Reference* for more information), the Sound Tool Set fields these interrupts first. Upon entry to the interrupt routine, the accumulator register contains the low-order nibble of the *genNumFFSynth* parameter from the `FFStartSound` tool call that assigned the oscillator. If the value of this nibble indicates that the interrupt is for the Sound Tool Set, the interrupt handler processes the interrupt. Otherwise, it passes the interrupt on to other interrupt routines (see the discussion of the `SetUserSoundIRQV` tool call in Chapter 20, "Sound Tool Set," in Volume 2 of the *Toolbox Reference* for information on setting vectors to user interrupt routines).

- **Mode** The mode value consists of two bits, M0 and M1. There are thus four possible modes, which are designated as free-run mode or loop mode (00), one-shot mode (01), sync/AM mode (10), and swap mode (11).

In free-run or loop mode, the oscillator sweeps through a waveform to the end, playing the values it encounters, then starts again at the beginning of the waveform and generates an interrupt if the interrupt enable bit is set to 1. The generator also interrupts if it encounters a 0, or if it reaches the end of the waveform.

In one-shot mode, the oscillator sweeps through its waveform to the first 0 value or to the end, generates an interrupt if the interrupt enable bit is setto 1, and halts.

In swap mode, an oscillator sweeps through its waveform to the first 0 value, or to the end of the waveform, generates an interrupt, and halts, turning control over to a partner oscillator. Only one halt bit can be set to 1 at any given time for a pair of oscillators in swap mode; setting the halt bit of one oscillator to 1 forces the other's to 0.

Generators always consist of an even/odd pair of oscillators—for example, oscillators 0 and 1 form a generator, as do oscillators 2 and 3, and so on. The Note Synthesizer normally uses each pair with the even-numbered oscillator in swap mode and the odd-numbered oscillator in loop mode. The Sound Tool Set normally uses both oscillators of a pair in swap mode.

Channel register

The Channel register specifies a sound's stereo position. Currently, only the low-order bit is significant. A value of 0 in this bit sets the oscillator's stereo position to the right channel; a value of 1 sets it to the left channel.

Bank-Select/Table-Size/Resolution register

This register sets the table size for stored waveforms, the resolution of the waveform, and the bank selection for the oscillator. When it plays a sound, the DOC adds the value of the Frequency register to its accumulator. It multiplexes the resulting value with the address pointer to determine the address in DOC RAM of the sample to play. The table size determines how many bits of the Address Pointer register are accessible to the DOC for this operation; a larger table size reduces the number of Address Pointer register bits used in the address calculation, and reduces the precision with which a particular sample can be located. If eight bits of the Address Pointer register are used to locate the next sample, the DOC can distinguish twice as many starting points as it can if only seven bits are used.

Each time the DOC cycles it adds the contents of the Frequency register to its 24-bit accumulator. It then appends a subrange of the accumulator's 24 bits to the value of the address pointer and uses the resulting value as an absolute address in DOC RAM. It then plays the sample stored at that location.

The resolution value, which is the lowest three bits of the Bank-Select/Table-Size/Resolution register, determines the lowest bit of the accumulator value that will be appended to the address pointer.

The table-size value, which is the next three bits above the resolution, determines both width of the address pointer value and the width of the accumulator value. The width of each value is the number of bits the DOC uses from that register. For example, the DOC accumulator is a 24-bit register, but the DOC uses only eight of those bits when the table size is 256 bytes.

The DOC uses only part of each register, the accumulator and the address pointer, to determine the place in memory that it will play next. For any table size greater than 256 bytes (1 page), it overwrites the lowest bits of the address pointer with bits from the accumulator. Figure 47-1 shows the correspondence between table size, resolution, and the portions of the Address Pointer register and accumulator used to determine the location of the next sample to be played.

- **Figure 47-1** DOC registers

TABLE TEXT	**FINAL ADDRESS**																**Resolution**		
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	R2	R1	R0
256	**Pointer Register**								23		**Accumulator Bits**					16	1	1	1
	7							0	16							9	0	0	0
512								23								15	1	1	1
	7						1	16								8	0	0	0
1024							23									14	1	1	1
	7					2	16									7	0	0	0
2048						23										13	1	1	1
	7				3	16										6	0	0	0
4096					23											12	1	1	1
	7			4	16											5	0	0	0
8192				23												11	1	1	1
	7		5	16												4	0	0	0
16384			23													10	1	1	1
	7	6	16													3	0	0	0
32768		23														9	1	1	1
	7	16														2	0	0	0

The resolution acts as an offset value, determining which bit is the lowest accumulator bit to be appended to the Address Pointer register. The effect of these computations is that if you increase the resolution by 1, the pitch of the waveform will be one octave lower. If you increase the resolution value by 2, the pitch will be four octaves lower—and so on in powers of two.

The table-size value is a 3-bit value that is equal to the resolution value in calls to `FFStartSound`. It specifies the size of the DOC RAM partitions used to contain waveforms that are to be played. The correspondence between table-size values and the table sizes is

Table-size	**3-bit value**	**RAM buffer size**
0	000	1 page (256 bytes)
1	001	2 page (512 bytes)
2	010	4 pages (1024 bytes)
3	011	8 pages (2048 bytes)
4	100	16 pages (4096 bytes)
5	101	32 pages (8192 bytes)
6	110	64 pages (16,384 bytes)
7	111	128 pages (32,768 bytes)·

Both the table-size and resolution values are copied into their respective bits in the Bank-Select/Table-Size/Resolution register from the lowest three bits of the *buffersize* parameter to the `FFStartSound` call.

- **Bank-select bit** The bank-select bit is bit 6. It is reserved for the use of Apple Computer, Inc. and should always be 0.

Oscillator Interrupt register

This register contains a bit that specifies whether an interrupt has occurred and, if so, contains the number of the oscillator that generated the interrupt. The oscillator number (0–31) is stored in bits 1 through 5 of this register.

Oscillator Enable register

The Oscillator Enable register specifies the number of enabled oscillators (0-31).

A/D Converter register

This register always contains the current sample from the analog-to-digital converter built into the Digital Oscillator Chip.

MIDI and interrupts

The MIDI Tool Set automatically recovers incoming MIDI data, but to do so it requires that interrupts never be disabled for longer than 290 microseconds. If an application disables interrupts for longer than this, it should call the `MidiInputPoll` vector at least every 270 microseconds to ensure that the data are properly received and the input buffer is cleared. When MIDI input is not enabled, the vector is still serviced, but at minimal cost in CPU cycles. Under these circumstances, the call to the vector sacrifices only two instructions, namely a `JSL` and an `RTL`.

Normally one CPU cycle on the Apple IIGS lasts about 1 microsecond, but it may take slightly longer if it accesses soft switches or slow memory.

New Sound Tool Set calls

There are four new tool calls that provide greater flexibility when playing free-form sounds. The `FFSetUpSound` and `FFStartPlaying` calls allow you to schedule a sound for playback at a later time. `SetDOCReg` and `ReadDOCReg` provide easy access to the DOC registers.

`FFSetUpSound` $1508

This call is identical to the `FFStartSound` tool call, but does not actually start playing the specified sound. Use the `FFStartPlaying` tool call to start playing. This call gives you the option of setting up a sound and playing it later.

Parameters

Stack before call

Previous contents	
channelGen	Word—Channel, generator type word
– *paramBlockPtr* –	Long—Pointer to `FFStartSound` parameter block
	<—SP

Stack after call

Previous contents	
	<—SP

Errors None

C

```
extern pascal void FFSetUpSound(channelGen,
                    paramBlockPtr);

Word        channelGen;
Pointer     paramBlockPtr;
```

channelGen For complete information on the *channelGen* parameter, refer to the description of the `FFStartSound` tool call in Chapter 20, "Sound Tool Set," in Volume 2 of the *Toolbox Reference.*

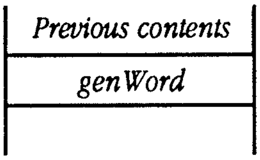
paramBlockPtr For complete information on the parameter block pointed to by the *paramBlockPtr* parameter, see "`FFStartSound`" earlier in this chapter.

`FFStartPlaying` $1608

Starts playing the sound specified by the `FFSetUpSound` tool call on a specified set of generators. Your program passes a parameter to this call indicating which generators are to play the sound.

Parameters

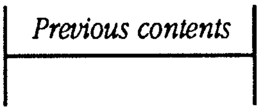
Stack before call



Word—Flag word indicating which generators to start

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void FFStartPlaying(genWord);

Word       genWord;
```

genWord Specifies which generators to start. Each bit in the word corresponds to a generator. Setting a bit to 1 indicates that the matching generator is to play the sound. For example, a *genWord* value of $4071 (%0100 0000 0111 0001) would start generators 0, 4, 5, 6, and 14.

▲ **Warning** A value of $0000 for this parameter is illegal and will cause the system to hang. ▲

ReadDOCReg $1808

Reads the DOC registers for a generator's oscillator and stores the register contents in a special format in the target memory location. Your program specifies the generator and which oscillator, as well as the destination for the register information. The format of the resultant data structure corresponds to the input to the `SetDOCReg` tool call.

▲ **Warning** This is a very low-level call. You should not use it unless you have a thorough understanding of the DOC. This call may not be supported in future versions of the system hardware. ▲

Parameters

Stack before call

Previous contents	
–dRegParmBlkPtr –	Long—Pointer to DOC register parameter block
	<—SP

Stack after call

Previous contents	
	<—SP

Errors None

C

```
extern pascal void ReadDOCReg(dRegParmBlkPtr);

Pointer   dRegParmBlkPtr;
```

dRegParmBlkPtr Refers to a location in memory to be loaded with the contents of the DOC registers for the specified generator.

$00	`oscGenType`	Word—(see below)
$02	`freqLow1`	Byte—Frequency Low register for first oscillator
$03	`freqHigh1`	Byte—Frequency High register for first oscillator
$04	`vol1`	Byte—Volume register for first oscillator
$05	`tablePtr1`	Byte—Waveform Table Pointer register for first oscillator
$06	`control1`	Byte—Control register for first oscillator
$07	`tableSize1`	Byte—Table-size register for first oscillator
$08	`freqLow2`	Byte—Frequency Low register for second oscillator
$09	`freqHigh2`	Byte—Frequency High register for second oscillator
$0A	`vol2`	Byte—Volume register for second oscillator
$0B	`tablePtr2`	Byte—Waveform Table Pointer register for second oscillator
$0C	`control2`	Byte—Control register for second oscillator
$0D	`tableSize2`	Byte—Table-size register for second oscillator

oscGenType Bits 8 through 11 specify the generator number ($0 through $F) whose registers are to be retrieved.

bit 15 Determines whether to get DOC registers for the first oscillator.
1 - Get the registers
0 - Do not get the registers

bit 14 Determines whether to get DOC registers for the second oscillator.
1 - Get the registers
0 - Do not get the registers

bits 12–13 Reserved; must be set to 0.

bits 8–11 Specify the generator number for the operation. Valid values lie in the range from $0 through $F.

bits 4–7 Reserved; must be set to 0.

bits 0–3 Specify who is using the generator (this value is returned)

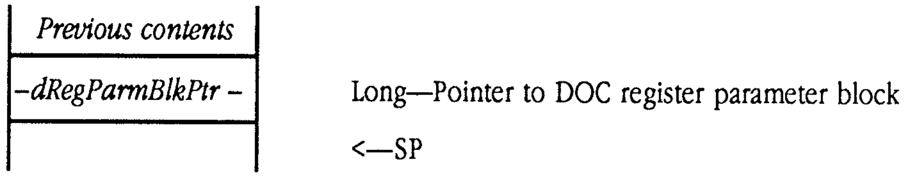
`SetDOCReg` $1708

Sets the DOC registers for a generator's oscillator from register contents stored in a special format. Your program specifies the generator, the oscillator(s) and the register information. The format of the input data structure corresponds to the output from the `ReadDOCReg` tool call.

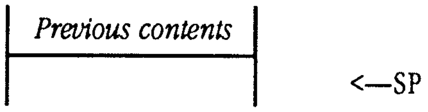
▲ **Warning** This is a very low-level call. You should not use it unless you have a thorough understanding of the DOC. This call may not be supported in future versions of the system hardware. ▲

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal void SetDOCReg(dRegParmBlkPtr);

Pointer   dRegParmBlkPtr;
```

dRegParmBlkPtr Refers to a location in memory containing the new contents of the DOC registers for the specified generator:

$00	`oscGenType`	Word—(see below)
$02	`freqLow1`	Byte—Frequency Low register for first oscillator
$03	`freqHigh1`	Byte—Frequency High register for first oscillator
$04	`vol1`	Byte—Volume register for first oscillator
$05	`tablePtr1`	Byte—Waveform Table Pointer register for first oscillator
$06	`control1`	Byte—Control register for first oscillator
$07	`tableSize1`	Byte—Table-size register for first oscillator
$08	`freqLow2`	Byte—Frequency Low register for second oscillator
$09	`freqHigh2`	Byte—Frequency High register for second oscillator
$0A	`vol2`	Byte—Volume register for second oscillator
$0B	`tablePtr2`	Byte—Waveform Table Pointer register for second oscillator
$0C	`control2`	Byte—Control register for second oscillator
$0D	`tableSize2`	Byte—Table-size register for second oscillator

oscGenType Specifies the generator whose oscillators are to be written, along with other generator control block (GCB) information (see Chapter 41, "Note Synthesizer," in this book for detailed information on the format and content of the GCB):

bit 15 Determines whether to set DOC registers for the first oscillator.
1 - Set the registers
0 - Do not set the registers

bit 14 Determines whether to set DOC registers for the second oscillator.
1 - Set the registers
0 - Do not set the registers

bits 12–13 Reserved; must be set to 0.

bits 8–11 Specify the generator number for the operation. Valid values lie in the range from $0 through $F.

bits 4–7 Reserved; must be set to 0.

bits 0–3 Specify who is using the generator.
$0 - Invalid value
$1 - Free-form synthesizer
$2 - Note Synthesizer
$3 - Reserved
$4 - MIDI
$5–$7 - Reserved
$8–$F - User-defined

Chapter 48 **Standard File Operations Tool Set**

This chapter documents new features of the Standard File Operations Tool Set. The complete reference to this tool set is in Volume 2, Chapter 22 of the *Apple IIGS Toolbox Reference.*

New features in the Standard File Operations Tool Set

This section explains new features of the Standard File OperationsTool Set.

- The Standard File Operations Tool Set now uses Class 1 calls to fully support GS/OS. As a result, new tool set calls accept full GS/OS filenames and pathnames:
  - □ A total of 13,107 files, with a total of up to 64 KB of name strings, can reside in a single folder.
  - □ A filename can now contain up to 253 characters.
  - □ A pathname can now contain up to 508 characters.

  New applications should use the new tool set calls to gain access to this functionality.

◆ *Note:* Since old The Standard File Operations Tool Set calls use the new, longer filenames and pathnames internally, it is possible for an old-style Get or Put call to access an AppleShare® file with a name that is more than 15 characters long. In this case, the system truncates the filename in the reply record. If necessary, the pathname is also truncated. Note, however, that if the pathname will fit in the reply record, then it is returned intact, regardless of the length of the filename portion of the path. As a result, this representation of the filename may exceed 15 characters. While this allows the application to open the file, programs that cannot accept a filename with more than 15 characters may not function predictably.

- The Standard File Operations Tool Set now uses the List Manager for some internal functions, freeing up memory for application use.
- The Standard File Operations Tool Set now requires that there be four pages available on the application stack (three for List Manager, one for the Standard File Operations Tool Set itself).
- The new tool calls use prefixes differently. These calls first check prefix 8 for a valid path. If prefix 8 is valid, the routines use that path. If not, they check prefix 0. If prefix 0 is valid, the routines copy it to prefix 8, then use it. If prefix 0 is also invalid, the search continues to the next volume.

  Anytime the user changes the pathname, even during a cancelled Standard File dialog, the new path is placed in prefix 8. In addition, this current path is placed into prefix 0, if it fits. If the path will not fit, prefix 0 is left unchanged and contains the last legitimate pathname entered.

  Internally, both old and new Standard File calls use prefix 8, allowing up to 508 characters in the pathname. However, the Standard File Operations Tool Set will display a warning if, as a result of an old call, a pathname longer than 64 characters will be created.

- The Standard File Operations Tool Set now scans AppleShare volumes every eight seconds for changes. The system automatically updates the displayed file list.
- The Standard File Operations Tool Set now returns error codes. For many internal errors, the Standard File Operations Tool Set will display a detailed information dialog and allow the user to cancel the operation.
- When displaying a complete path, the system now uses the separator character found in either prefix 8 or 0. Previously, the separator was always /, but now typically will be a colon (:).
- The system now disables the Save and NewFolder buttons in Put dialogs referencing write-protected volumes. In addition, the system will now display a lock icon for such volumes.
- The Standard File Operations Tool Set now supports multiple file Get calls. See "New Standard File calls" later in this chapter for call syntax details.

  Multifile dialogs include a new Accept button in addition to the Open button. These buttons operate as follows:

  - When the user has selected a single file, both the Open and Accept buttons are enabled. If the selected file is not a folder, clicking either button will return the filename. If the file is a folder, clicking Open lists the folder contents, while clicking Accept returns the folder name to the calling program. Double-clicking on a file works the same as Accept; double-clicking on a folder works the same as Open.
  - When the user has selected multiple files, the Open button is disabled. The user must press the Accept button to return the filenames to the calling application. In this case, the returned file list may contain both folder and file names. Double-clicking is not allowed when multiple files have been selected.
- The Standard File Operations Tool Set now uses static text items in its dialog templates. The system will automatically change custom dialog templates to use static text, rather than user items. In addition, the system now uses a custom draw routine for the path entry item. The system automatically changes input dialog templates to call the Standard File Operations Tool Set's custom draw routine, unless the input template already references a custom routine, in which case that reference is not changed.
- Your application can now provide custom draw routines for items in displayed file lists. The Standard File Operations Tool Set takes care of dimming and selecting the item.

- Standard File dialogs support the following keystroke equivalents:

Key	Button Equivalent
Esc	Close
Command–Up Arrow	Close
Tab	Volume
Command-period	Cancel
Command-o	Open
Command-O	Open
Command–Down Arrow	Open
Command-n	New Folder
Command-N	New Folder

New filter procedure entry interface

Many Standard File calls allow you to specify a custom filter procedure. These custom routines can perform specific checking of items for file list inclusion, beyond that performed by the system. To learn more about Standard File filter procedures, see Chapter 22, "Standard File Operations Tool Set," in Volume 2 of the *Toolbox Reference.*

The new Standard File calls support a different filter procedure entry interface. Previously, Standard File filter procedures received a pointer to a file directory entry (defined in the *Toolbox Reference*). New Standard File calls pass a pointer to a `GetDirEntry` record, which corresponds to the formatted output of the GS/OS `GetDirEntry` call. For the format and content of the `GetDirEntry` record, refer to Volume 1 of the *GS/OS Reference.*

The exit interface from these filter procedures has not changed. Your program must remove this pointer from the stack and return a response word indicating how the current file is to be displayed in the file list.

Value	Name	Description
0	`noDisplay`	Do not display file
1	`noSelect`	Display the file, but do not allow the user to select it
2	`displaySelect`	Display the file and allow the user to select it

Custom item drawing routines

Some new Standard File calls allow you to specify custom item-drawing routines. These routines give you the opportunity to create highly customized displays of items in file lists. The Standard File Operations Tool Set handles item dimming and selecting.

On entry to the custom item drawing routine, Standard File formats the stack as follows:

Previous contents	
– *memRectPtr* –	Long—Pointer to the member rectangle
– *memberPtr* –	Long—Pointer to the member record
– *controlHndl* –	Long—Handle to the list control
Reserved	Block—Reserved data for Standard File—24 bytes
– *itemDrawPtr* –	Long—Pointer to item draw record
– *returnAddr* –	Block—RTL address—3 bytes
	<—SP

The *itemDrawPtr* field contains a pointer to a record formatted as follows:

$00	`count`	Byte—Length of *nameString*, in bytes
$01	`nameString`	Array—*count* bytes of file name
count + 1	`fileType`	Word—File type
count + 3	`fileAuxType`	Long—Auxiliary file type

The routine must remove this pointer from the stack before returning to the Standard File Operations Tool Set. The custom item drawing routine should not change any of the other information on the stack.

The custom drawing routine must draw both the filename string and any associated icon. The Standard File Operations Tool Set assumes that the standard system font and character size are used for all list items; changing either font or character size is not recommended. Note that any icons must also comply with these restrictions (current icons are eight lines high).

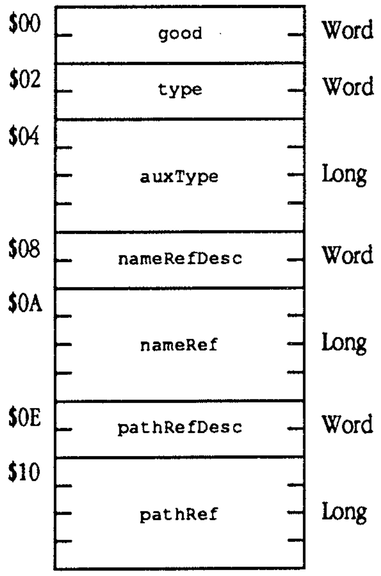
Standard File data structures

The new Standard File tool calls accept and return new style reply records and type lists. The formats for these records follow.

Reply record

Figure 48-1 defines the layout for the new-style Standard File reply record. You pass this record to many of the new tool calls. Those calls, in turn, update the record and return it to your program.

■ **Figure 48-1** New-style reply record



$00	good	Word
$02	type	Word
$04	auxType	Long
$08	nameRefDesc	Word
$0A	nameRef	Long
$0E	pathRefDesc	Word
$10	pathRef	Long

`good` Boolean indicating the status of the request. TRUE indicates that the user opened the file; FALSE indicates that the user cancelled the request.

`type` Contains the GS/OS file type information.

`auxType` Contains the GS/OS auxiliary type information.

`nameRefDesc` Defines the type of reference stored in `nameRef` (your program must set this field):

$0000 Reference in `nameRef` is a pointer to a GS/OS class 1 output string.

$0001 Reference in `nameRef` is a handle to a GS/OS class 1 output string.

$0003 Reference in `nameRef` is undefined. The system will allocate a new handle, correctly sized for the resulting GS/OS class 1 output string, and return that handle in `nameRef`. This is the recommended option.

`nameRef` On input, may contain a reference to the output buffer for the filename string, depending upon the contents of `nameRefDesc`. On output, contains a reference to the filename string. The reference type is defined by the contents of `nameRefDesc`. If your program set `nameRefDesc` to $0003, then your program must release the resulting handle when it is done with the returned data.

`pathRefDesc` Defines the type of reference stored in *pathRef*(your program must set this field):

$0000 Reference in `pathRef` is a pointer to a GS/OS class 1 output string.

$0001 Reference in `pathRef` is a handle to a GS/OS class 1 output string.

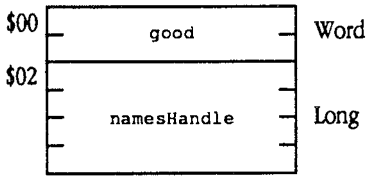
$0003 Reference in `pathRef` is undefined. The system will allocate a new handle, correctly sized for the resulting GS/OS class 1 output string, and return that handle in `pathRef`. This is the recommended option.

`pathRef` On input, may contain a reference to the output buffer for the file pathname string, depending upon the contents of `pathRefDesc`. On output, contains a reference to the pathname string. The reference type is defined by the contents of `pathRefDesc`. If your program set `pathRefDesc` to $0003, then your program must release the resulting handle when it is done with the returned data.

Multifile reply record

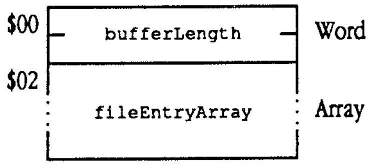
Figure 48-2 defines the format of the Standard File multifile reply record. The system returns this record format in response to multifile Get requests.

- **Figure 48-2** Multifile reply record



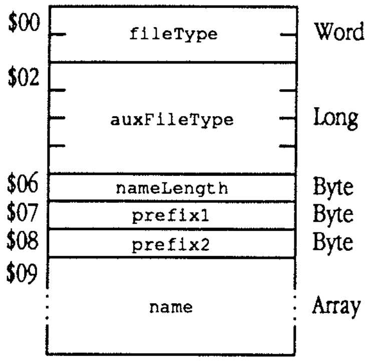
`good` Contains either the number of files selected, or FALSE if the user cancelled the request.

`namesHandle` Handle to the returned data record. Your program must dispose of this handle when it is done with the returned data. The returned data record is formatted as follows:



`bufferLength` Contains the total length, in bytes, of the returned data record, including the length of `bufferLength`.

`fileEntryArray`
An array of file entries, each formatted as follows:



`fileType` Contains the GS/OS file type.

`auxFileType` Contains the GS/OS auxiliary type.

`nameLength` Contains the length of the following filename, including the volume prefix bytes (`prefix1` and `prefix2`).

`prefix1` Volume prefix for the pathname, first byte. Always set to 8.

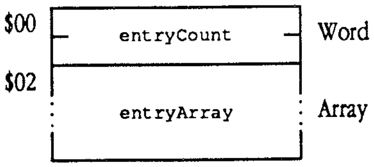
`prefix2` Volume prefix for the pathname, second byte. Always set to :.

`name` Filename string, containing (`nameLength` - 2) bytes of data, not to exceed 253 characters.

File type list record

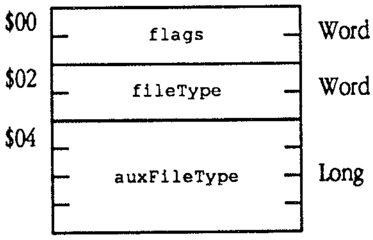
Figure 48-3 shows the layout of the Standard File type list record. You use this record with Standard File calls that require a file type list as input (such as `SFGetFile2`).

- **Figure 48-3** File type list record



entryCount Contains the number of items in entryArray.

entryArray Array of file type entries, each formatted as follows:



`flags` Defines how the system is to use `fileType` and `auxFileType` when selecting files to be displayed:

bit 15 Controls whether Standard File cares about auxiliary types:
`1` - Match any `auxFileType` value
0 - Match only the specified `auxFileType` value

bit 14 Controls whether Standard File cares about file types:
1 - Match any `fileType`
0 - Match only the specified `fileType` value

bit 13 Disable selection:
1 - Any files matching criteria specified in bits 14 and 15 will be displayed but will not be selectable (will be greyed out). Note that the file(s) will not be passed to the filter procedure for the tool call.

bits 12–0 Reserved; must be set to 0.

◆ *Note:* The settings of bits 14 and 15 are independent. By setting both bits to 1, The Standard File Operations Tool Set will match all files.

`fileType` Specifies a GS/OS file type value to match, according to the settings of the `flags` bits. If bit 15 of `flags` is set to 0, then this field is ignored.

`auxFileType` Specifies a GS/OS auxiliary type value to match, according to the settings of the `flags` bits. If bit 14 of `flags` is set to 0, then this field is ignored.

Standard File dialog templates

The Standard File Operations Tool Set allows you to define custom dialog boxes for the Open File and Save File dialog boxes. To use a custom dialog box, your program must provide a pointer to a dialog template to the appropriate Standard File routines (`SFPPutFile2`, `SFPGetFile2`, or `SFPMultiFile2`). The Standard File Operations Tool Set passes the dialog template to the Dialog Manager (`GetNewModalDialog` call) when it establishes the user dialog.

While the latest version of the Standard File Operations Tool Set uses some of the template fields differently, old templates should still work. The system internally modifies old-style input templates to make them compatible with current usage. New usage differs in the following ways:

- The boundary rectangle for a file list is taken from the Files item in each dialog template and copied to the List Manager record. The number of files to be displayed is derived from the rectangle coordinates by subtracting 2 from the height and dividing the result by 10. To avoid displaying partial filenames, you should set the rectangle height using the same formula; that is, height=((num_files * 10) + 2).
- The Scroll item is no longer used for single-file requests. However, it has been retained in the record for compatibility with old templates. For multifile Get requests, the new tool calls define the Accept button in the space previously used by the Scroll item.
- Standard File calls copy the input dialog template header to memory and then update it. Note that, for single-file Get calls, items 5 and 7 are not copied (for multifile Get calls, item 5 is copied). Similarly, items 6 and 8 are not copied for Put calls.

The following code examples contain the templates for the standard Open File and Save File dialog boxes. All these templates depend upon the following string definitions:

```
SaveStr         str        'Save'
OpenStr         str        'Open'
CloseStr        str        'Close'
DriveStr        str        'Volume'
CancelStr       str        'Cancel'
FolderStr       str        'New Folder'
AcceptStr       str        'Accept'
KbFreeStr       str        '^0 free of ^1 k.'      ;Dialog Manager routine
                                            ; replaces ^0 & ^1 with real
                                            ; values from the disk.
PPromptStr      dc.b 'Save which file:'
PEndBuf         dc.b 0                      ;end-of-string byte

GPromptStr      dc.b 'Load which file:'
GEndBuf         dc.b 0                      ;end-of-string byte
```

Open File dialog box templates

The Open File dialog box must contain the following items in this exact order:

Item	Item type	Item ID
Open button	`buttonItem`	1
Close button	`buttonItem`	2
Next button	`buttonItem`	3
Cancel button	`buttonItem`	4
Scroll bar	`userItem+itemDisable`	5
Path	`userItem`	6
Files	`userItem+itemDisable`	7
Prompt	`userItem`	8

◆ *Note:* The Scroll bar item (item 5) is not used for single-file calls. For multifile calls, this item contains the Accept button definition.

◆ *Note:* The Files item (item 7) contains the boundary rectangle for the List Manager, and serves no other purpose.

First, the templates for 640 mode:

```
GetDialog640

        dc.w    0,0,114,400         ; recommended drect of dialog
                                    ;  (640 mode)
        dc.w    -1
        dc.w    0,0                 ; reserved words
        dc.l    OpenBut640          ; item 1
        dc.l    CloseBut640         ; item 2
        dc.l    NextBut640          ; item 3
        dc.l    CancelBut640        ; item 4
        dc.l    Scroll640           ; dummy item or ACCEPT button
        dc.l    Path640             ; item 6
        dc.l    Files640            ; item 7
        dc.l    Prompt640           ; item 8
        dc.l    0
```

```
OpenBut640
        dc.w    1                   ;item #
        dc.w    61,265,73,375       ;drect
        dc.w    ButtonItem          ;type of item
        dc.l    OpenStr             ;Item descriptor
        dc.w    0,0                 ;Item value & bit flags
        dc.l    0                   ;color table ptr (nil is default)

CloseBut640
        dc.w    2                   ;item #
        dc.w    79,265,91,375       ;drect
        dc.w    ButtonItem          ;type of item
        dc.l    CloseStr            ;Item descriptor
        dc.w    0,0                 ;Item value & bit flags
        dc.l    0                   ;color table ptr (nil is default)

NextBut640
        dc.w    3                   ;item #
        dc.w    25,265,37,375       ;drect
        dc.w    ButtonItem          ;type of item
        dc.l    DriveStr            ;Item descriptor
        dc.w    0,0                 ;Item value & bit flags
        dc.l    0                   ;color table ptr (nil is default)

CancelBut640
        dc.w    4                   ;item #
        dc.w    97,265,109,375      ;drect
        dc.w    ButtonItem          ;type of item
        dc.l    CancelStr           ;Item descriptor
        dc.w    0,0                 ;Item value & bit flags
        dc.l    0                   ;color table ptr (nil is default)
```

```
Scroll640
;
; SPECIAL NOTE: Scroll items are no longer used by the new calls, since
;   the List Manager takes care of all scroll "stuff". In single-file
;   Get calls (also any OLD call), this item is simply ignored and its
;   pointer is left out of the header when copied to RAM. However, in
;   Multi-Get calls, this item IS used for the Accept button. The
;   following is the recommended content for the Accept button:
;
      dc.w    5                     ;item # (DUMMY or ACCEPT button)
      dc.w    43,265,55,375         ;drect
      dc.w    ButtonItem            ;type
      dc.l    AcceptStr             ;Item descriptor
      dc.w    0,0                   ;Item value and bit flags
      dc.l    0                     ;color table

Path640
      dc.w    6                     ;item #
      dc.w    12,15,24,395          ;drect
      dc.w    UserItem              ;type
      dc.l    PathDraw              ;Item descriptor (user app.
specific)
      dc.w    0,0                   ;Item value and bit flags
      dc.l    0                     ;color table

Files640
      dc.w    7                     ;item #
      dc.w    25,18,107,215         ;Boundary rect for List Manager
      dc.w    UserItem+ItemDisable  ;type
      dc.l    0                     ;Item descriptor
      dc.w    0,0                   ;Item value and bit flags
      dc.l    0                     ;color table

Prompt640
      dc.w    8                     ;item #
      dc.w    03,15,12,395          ;drect
      dc.w    StatText+ItemDisable  ;type
      dc.l    0                     ;Item descriptor (Text passed)
      dc.w    0                     ;size of text
      dc.w    0                     ;Bit flags
      dc.l    0                     ;color table
```

Now, the templates for 320 mode:

```
GetDialog320

      dc.w    0,0,114,260           ; drect of dialog (320 mode)
      dc.w    -1
      dc.w    0,0                   ; reserved word
      dc.l    OpenBut320            ; item 1
      dc.l    CloseBut320           ; item 2
      dc.l    NextBut320            ; item 3
      dc.l    CancelBut320          ; item 4
      dc.l    Scroll320             ; dummy item or ACCEPT button
      dc.l    Path320               ; item 6
      dc.l    Files320              ; item 7
      dc.l    Prompt320             ; item 8
      dc.l    0

OpenBut320
      dc.w    1                     ;item #
      dc.w    53,160,65,255         ;drect
      dc.w    ButtonItem            ;type of item
      dc.l    OpenStr               ;Item descriptor
      dc.w    0,0                   ;Item value & bit flags
      dc.l    0                     ;color table ptr (nil is default)

CloseBut320
      dc.w    2                     ;item #
      dc.w    71,160,83,255         ;drect
      dc.w    ButtonItem            ;type of item
      dc.l    CloseStr              ;Item descriptor
      dc.w    0,0                   ;Item value & bit flags
      dc.l    0                     ;color table ptr (nil is default)

NextBut320
      dc.w    3                     ;item #
      dc.w    27,160,39,255         ;drect
      dc.w    ButtonItem            ;type of item
      dc.l    DriveStr              ;Item descriptor
      dc.w    0,0                   ;Item value & bit flags
      dc.l    0                     ;color table ptr (nil is default)
```

```
CancelBut320
      dc.w    4                     ;item #
      dc.w    97,160,109,255        ;drect
      dc.w    ButtonItem            ;type of item
      dc.l    CancelStr             ;Item descriptor
      dc.w    0,0                   ;Item value & bit flags
      dc.l    0                     ;color table ptr (nil is default)

Scroll320
;
; SPECIAL NOTE: Scroll items are no longer used by the new calls, since
;   the List Manager takes care of all scroll "stuff". In single-file
;   Get calls (also any OLD call), this item is simply ignored and its
;   pointer is left out of the header when copied to RAM. However, in
;   Multi-Get calls, this item IS used for the Accept button. The
;   following is the recommended content for the Accept button:
;
      dc.w    5                     ;item # (see SPECIAL NOTE)
      dc.w    118,160,130,255       ;drect
      dc.w    ButtonItem            ;type
      dc.l    AcceptStr             ;Item descriptor
      dc.w    0,0                   ;Item value and bit flags
      dc.l    0                     ;color table

Path320
      dc.w    6                     ;item #
      dc.w    14,06,26,256          ;drect
      dc.w    UserItem              ;type
      dc.l    PathDraw              ;Item descriptor
      dc.w    0,0                   ;Item value and bit flags
      dc.l    0                     ;color table

Files320
      dc.w    7                     ;item #
      dc.w    27,05,109,140         ;Boundary rect for list manager
      dc.w    UserItem+ItemDisable  ;type
      dc.l    0                     ;Item descriptor
      dc.w    0,0                   ;Item value and bit flags
      dc.l    0                     ;color table
```

```
Prompt320
      dc.w     8                    ;item #
      dc.w     03,05,12,255         ;drect
      dc.w     StatText+ItemDisable ;type
      dc.l     0                    ;Item descriptor (Text passed)
      dc.w     0                    ;size of string
      dc.w     0                    ;Bit flags
      dc.l     0                    ;color table (0 = default)
```

Save File dialog box templates

The Save File dialog box must contain the following items in this exact order:

Item	Item type	Item ID
Save button	`buttonItem`	1
Open button	`buttonItem`	2
Close button	`buttonItem`	3
Next button	`buttonItem`	4
Cancel button	`buttonItem`	5
Scroll bar	`userItem+itemDisable`	6
Path	`userItem`	7
Files	`userItem+itemDisable`	8
Prompt	`userItem`	9
Filename	`editItem`	10
Free space	`statText`	11
Create button	`buttonItem`	12

◆ *Note:* The Scroll bar item (item 6) is not used for single-file calls.

◆ *Note:* The Files item (item 8) contains the boundary rectangle for the List Manager, and serves no other purpose.

First, the templates for 640 mode:

```
PutDialog640

      dc.w    0,0,120,320             ; recommended drect of dialog
                                      ;  (640 mode)
      dc.w    -1
      dc.w    0,0                     ; reserved word
      dc.l    SaveButP640             ; item 1
      dc.l    OpenButP640             ; item 2
      dc.l    CloseButP640            ; item 3
      dc.l    NextButP640             ; item 4
      dc.l    CancelButP640           ; item 5
      dc.l    ScrollP640              ; DUMMY item
      dc.l    PathP640                ; item 7
      dc.l    FilesP640               ; contains boundary rect only
      dc.l    PromptP640              ; item 9
      dc.l    EditP640                ; item 10
      dc.l    StatTextP640            ; item 11
      dc.l    CreateButP640           ; item 12
      dc.l    0

SaveButP640
      dc.w    1                       ;item #
      dc.w    87,204,99,310           ;drect
      dc.w    ButtonItem              ;type of item
      dc.l    SaveStr                 ;Item descriptor
      dc.w    0,0                     ;Item value & bit flags
      dc.l    0                       ;color table ptr (nil is default)

OpenButP640
      dc.w    2                       ;item #
      dc.w    49,204,61,310           ;drect
      dc.w    ButtonItem              ;type of item
      dc.l    OpenStr                 ;Item descriptor
      dc.w    0,0                     ;Item value & bit flags
      dc.l    0                       ;color table ptr (nil is default)
```

```
CloseButP640
        dc.w    3                       ;item #
        dc.w    64,204,76,310           ;drect
        dc.w    ButtonItem              ;type of item
        dc.l    CloseStr                ;Item descriptor
        dc.w    0,0                     ;Item value & bit flags
        dc.l    0                       ;color table ptr (nil is default)

NextButP640
        dc.w    4                       ;item #
        dc.w    15,204,27,310           ;drect
        dc.w    ButtonItem              ;type of item
        dc.l    DriveStr                ;Item descriptor
        dc.w    0,0                     ;Item value & bit flags
        dc.l    0                       ;color table ptr (nil is default)

CancelButP640
        dc.w    5                       ;item #
        dc.w    104,204,116,310         ;drect
        dc.w    ButtonItem              ;type of item
        dc.l    CancelStr               ;Item descriptor
        dc.w    0,0                     ;Item value & bit flags
        dc.l    0                       ;color table ptr (nil is default)

ScrollP640
;
; Special Note: Unlike Scroll item in Get, Scroll is never used
;  in Put, since there is not Multifile Put call.
;
        dc.w    6                       ;item # (Dummy item)
        dc.w    0,0,0,0                 ;dummy drect (must be 0)
        dc.w    UserItem                ;type
        dc.l    0                       ;Item descriptor
        dc.w    0,0                     ;Item value and bit flags
        dc.l    0                       ;color table
```

```
PathP640
        dc.w    7                       ;item #
        dc.w    0,10,12,315             ;drect
        dc.w    UserItem                ;type
        dc.l    PathDraw                ;Item descriptor (user app.specific)
        dc.w    0,0                     ;Item value and bit flags
        dc.l    0                       ;color table

FilesP640
        dc.w    8                       ;item #
        dc.w    26,10,88,170            ;Boundary rect for list manager
        dc.w    UserItem+ItemDisable    ;type
        dc.l    0                       ;Item descriptor
        dc.w    0,0                     ;Item value and bit flags
        dc.l    0                       ;color table

PromptP640
        dc.w    9                       ;item #
        dc.w    88,10,100,200           ;drect
        dc.w    StatText+ItemDisable    ;type
        dc.l    0                       ;Item descriptor
        dc.w    0                       ;size of text (Text passed)
        dc.w    0                       ;Bit flags
        dc.l    0                       ;color table

EditP640
        dc.w    10                      ;item #
        dc.w    100,10,118,194          ;drect
        dc.w    EditLine+ItemDisable    ;type
        dc.l    0                       ;Item descriptor
        dc.w    63                      ;size of text
        dc.w    0                       ;Bit flags
        dc.l    0                       ;color table
```

```
StatTextP640
      dc.w    11                   ;item #
      dc.w    12,10,22,200         ;drect
      dc.w    StatText+ItemDisable ;type
      dc.l    KbFreeStr            ;Item descriptor
      dc.w    0                    ;size of text
      dc.w    0                    ;Bit flags
      dc.l    0                    ;color table

CreateButP640
      dc.w    12                   ;item #
      dc.w    29,204,41,310        ;drect
      dc.w    ButtonItem           ;type
      dc.l    FolderStr            ;Item descriptor
      dc.w    0                    ;size of text
      dc.w    0                    ;Bit flags
      dc.l    0                    ;color table
```

Now, the templates for 320 mode:

```
PutDialog320

      dc.w    0,0,128,270          ; drect of dialog (320 mode)
      dc.w    -1
      dc.w    0,0                  ; reserved word
      dc.l    SaveButP320          ; item 1
      dc.l    OpenButP320          ; item 2
      dc.l    CloseButP320         ; item 3
      dc.l    NextButP320          ; item 4
      dc.l    CancelButP320        ; item 5
      dc.l    ScrollP320           ; DUMMY item
      dc.l    PathP320             ; item 7
      dc.l    FilesP320            ; contains Boundary rect
      dc.l    PromptP320           ; item 9
      dc.l    EditP320             ; item 10
      dc.l    StatTextP320         ; item 11
      dc.l    CreateButP320        ; item 12
      dc.l    0
```

```
SaveButP320
      dc.w    1                    ;item #
      dc.w    93,165,105,265       ;drect
      dc.w    ButtonItem           ;type of item
      dc.l    SaveStr              ;Item descriptor
      dc.w    0,0                  ;Item value & bit flags
      dc.l    0                    ;color table ptr (nil is default)

OpenButP320
      dc.w    2                    ;item #
      dc.w    54,165,66,265        ;drect
      dc.w    ButtonItem           ;type of item
      dc.l    OpenStr              ;Item descriptor
      dc.w    0,0                  ;Item value & bit flags
      dc.l    0                    ;color table ptr (nil is default)

CloseButP320
      dc.w    3                    ;item #
      dc.w    72,165,84,265        ;drect
      dc.w    ButtonItem           ;type of item
      dc.l    CloseStr             ;Item descriptor
      dc.w    0,0                  ;Item value & bit flags
      dc.l    0                    ;color table ptr (nil is default)

NextButP320
      dc.w    4                    ;item #
      dc.w    15,165,27,265        ;drect
      dc.w    ButtonItem           ;type of item
      dc.l    DriveStr             ;Item descriptor
      dc.w    0,0                  ;Item value & bit flags
      dc.l    0                    ;color table ptr (nil is default)

CancelButP320
      dc.w    5                    ;item #
      dc.w    111,165,123,265      ;drect
      dc.w    ButtonItem           ;type of item
      dc.l    CancelStr            ;Item descriptor
      dc.w    0,0                  ;Item value & bit flags
      dc.l    0                    ;color table ptr (nil is default)
```

```
ScrollP320
;
; Special Note: Unlike Scroll item in Get, Scroll is never used
;  in Put, since there is not Multifile Put call.
;
      dc.w    6                      ;item # (Dummy item)
      dc.w    00,00,00,00            ;drect
      dc.w    UserItem               ;type
      dc.l    0                      ;Item descriptor
      dc.w    0,0                    ;Item value and bit flags
      dc.l    0                      ;color table

PathP320
      dc.w    7                      ;item #
      dc.w    00,10,12,265           ;drect
      dc.w    UserItem               ;type
      dc.l    PathDraw               ;Item descriptor
      dc.w    0,0                    ;Item value and bit flags
      dc.l    0                      ;color table

FilesP320
      dc.w    8                      ;item #
      dc.w    26,10,88,145           ;Boundary rect for list manager
      dc.w    UserItem+ItemDisable   ;type
      dc.l    0                      ;Item descriptor
      dc.w    0,0                    ;Item value and bit flags
      dc.l    0                      ;color table

PromptP320
      dc.w    9                      ;item #
      dc.w    88,10,100,170          ;drect
      dc.w    StatText+ItemDisable   ;type
      dc.l    0                      ;Item descriptor (Text passed)
      dc.w    0
      dc.w    0                      ;Bit flags
      dc.l    0                      ;color table
```

```
EditP320
      dc.w     10                  ;item #
      dc.w     100,10,118,157      ;drect
      dc.w     EditLine+ItemDisable ;type
      dc.l     0                   ;Item descriptor
      dc.w     32                  ;size of text
      dc.w     0                   ;Bit flags
      dc.l     0                   ;color table

StatTextP320
      dc.w     11                  ;item #
      dc.w     12,10,22,160        ;drect
      dc.w     StatText+ItemDisable ;type
      dc.l     KbFreeStr           ;Item descriptor
      dc.w     0                   ;size of text
      dc.w     0                   ;Bit flags
      dc.l     0                   ;color table

CreateButP320
      dc.w     12                  ;item #
      dc.w     33,165,45,265       ;drect
      dc.w     ButtonItem          ;type
      dc.l     FolderStr           ;Item descriptor
      dc.w     0                   ;size of text
      dc.w     0                   ;Bit flags
      dc.l     0                   ;color table
```

New or changed Standard File calls

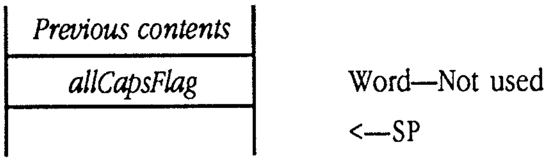
The following sections discuss several new or changed Standard File tool calls.

`SFAllCaps` $0D17

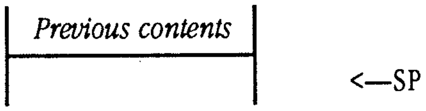
This call has been disabled in order to show filenames exactly as entered. Existing programs may still issue the call, but it will have no effect.

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal void SFAllCaps(allCapsFlag);

Word     allCapsFlag;
```

`SFGetFile2` $0E17

Displays the standard Open File dialog box and returns information about the file selected by the user. This call differs from `SFGetFile` in that it uses Class 1 GS/OS calls, thereby allowing selection of a file with a full name length of up to 763 characters.

Parameters

Stack before call

Previous contents	
whereX	Word—x coordinate of upper left corner of dialog box
whereY	Word—y coordinate of upper left corner of dialog box
promptRefDesc	Word—Defines type of reference in *promptRef*
– *promptRef* –	Long—Reference to Pascal string for file prompt
– *filterProcPtr* –	Long—Pointer to filter procedure; NIL for none
– *typelistPtr* –	Long—Pointer to type list record; NIL for none
– *replyPtr* –	Long—Pointer to new-style reply record
	<—SP

Stack after call

Previous contents	
	<—SP

Errors			
	$1701	`badPromptDesc`	Invalid *promptRefDesc* value
	$1704	`badReplyNameDesc`	Invalid reply record `nameRefDesc` value
	$1705	`badReplyPathDesc`	Invalid reply record `pathRefDesc` value
	GS/OS errors		Returned unchanged

◆ *Note:* The GS/OS `bufferTooSmall` error can occur if the output strings you supply in the reply record are too small to receive the resulting filename string. In this case, the buffer will contain as many name characters as would fit, and the length word will contain the name length the Standard File Operations Tool Set tried to return.

C

```
extern pascal void SFGetFile2(whereX, whereY,
          promptRefDesc, promptRef, filterProcPtr,
          typelistPtr, replyPtr);

Pointer   filterProcPtr, typelistPtr, replyPtr;
Word      whereX, whereY, promptRefDesc;
Long      promptRef;
```

promptRefDes Defines the type of reference in *promptRef*:

$0000	Reference in *promptRef* is a pointer to a Pascal string.
$0001	Reference in *promptRef* is a handle of a Pascal string.
$0002	Reference in *promptRef* is the resource ID of a Pascal string

filterProcPtr Pointer to a new-style filter procedure, as described in "New filter procedure entry interface" earlier in this chapter.

`SFMultiGet2` $1417

Displays the standard Open Multifile dialog box and returns information about the file or files selected by the user. The call returns file selection information in a multifile reply record. Note that folders may be included in the list of returned files; your program should check the file type field before using any returned filenames.

Parameters

Stack before call

Previous contents	
whereX	Word—x coordinate of upper left corner of dialog box
whereY	Word—y coordinate of upper left corner of dialog box
promptRefDesc	Word—Defines type of reference in *promptRef*
– *promptRef* –	Long—Reference to Pascal string for file prompt
– *filterProcPtr* –	Long—Pointer to filter procedure; NIL for none
– *typelistPtr* –	Long—Pointer to type list record; NIL for none
– *replyPtr* –	Long—Pointer to multifile reply record
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$1701	`badPromptDesc`	Invalid *promptRefDesc* value

C

```
extern pascal void SFMultiGet2(whereX, whereY,
          promptRefDesc, promptRef, filterProcPtr,
          typelistPtr, replyPtr);

Pointer   filterProcPtr, typelistPtr, replyPtr;
Word      whereX, whereY, promptRefDesc;
Long      promptRef;
```

promptRefDesc	Defines the type of reference in *promptRef*:
$0000	Reference in *promptRef* is a pointer to a Pascal string.
$0001	Reference in *promptRef* is a handle of a Pascal string.
$0002	Reference in *promptRef* is the resource ID of a Pascal string
filterProcPtr	Pointer to a new-style filter procedure, as described in "New filter procedure entry interface" earlier in this chapter.

`SFPGetFile2` $1017

Displays a custom Open File dialog box and returns information about the file selected by the user. This call differs from `SFGetFile` in that it uses Class 1 GS/OS calls, thereby allowing selection of a file with a full name length of up to 763 characters.

Parameters

Stack before call

Previous contents	
whereX	Word—x coordinate of upper left corner of dialog box
whereY	Word—y coordinate of upper left corner of dialog box
– *itemDrawPtr* –	Long—Pointer to item draw routine; NIL for none
promptRefDesc	Word—Defines type of reference in *promptRef*
– *promptRef* –	Long—Reference to Pascal string for file prompt
– *filterProcPtr* –	Long—Pointer to filter procedure; NIL for none
– *typelistPtr* –	Long—Pointer to type list record; NIL for none
– *dlgTempPtr* –	Long—Pointer to dialog template
– *dialogHookPtr* –	Long—Pointer to routine to handle item hits
– *replyPtr* –	Long—Pointer to new-style reply record
	<—SP

Stack after call

Previous contents	
	<—SP

Errors			
	$1701	`badPromptDesc`	Invalid *promptRefDesc* value
	$1704	`badReplyNameDesc`	Invalid reply record `nameRefDesc` value
	$1705	`badReplyPathDesc`	Invalid reply record `pathRefDesc` value
	GS/OS errors		Returned unchanged

◆ *Note:* The GS/OS `bufferTooSmall` error can occur if the output strings you supply in the reply record are too small to receive the resulting filename string. In this case, the buffer will contain as many name characters as would fit, and the length word will contain the name length the Standard File Operations Tool Set tried to return.

C

```
extern pascal void SFPGetFile2(whereX, whereY,
          itemDrawPtr, promptRefDesc, promptRef,
          filterProcPtr, typelistPtr, dlgTempPtr,
          dialogHookPtr, replyPtr);

Pointer   itemDrawPtr, filterProcPtr, typelistPtr,
          dlgTempPtr, dialogHookPtr, replyPtr;
Word      whereX, whereY, promptRefDesc;
Long      promptRef;
```

itemDrawPtr Pointer to a custom item-drawing routine, as described in "Custom item drawing routines" earlier in this chapter.

promptRefDesc Defines the type of reference in *promptRef*:

$0000	Reference in *promptRef* is a pointer to a Pascal string.
$0001	Reference in *promptRef* is a handle to a Pascal string.
$0002	Reference in *promptRef* is the resource ID to a Pascal string

filterProcPtr Pointer to a new-style filter procedure, as described in "New filter procedure entry interface" earlier in this chapter.

dlgTempPtr, dialogHookPtr
For more information about these fields, see the discussion of the `SFPutFile` call in Chapter 22, "Standard File Operations Tool Set," in Volume 2 of the *Toolbox Reference.*

`SFPMultiGet2` $1517

Displays a custom Open Multifile dialog box and returns information about the file or files selected by the user. The call returns file selection information in a multifile reply record. Note that folders may be included in the list of returned files; your program should check the file type field before using any returned filenames.

Parameters

Stack before call

Previous contents	
whereX	Word—x coordinate of upper left corner of dialog box
whereY	Word—y coordinate of upper left corner of dialog box
– *itemDrawPtr* –	Long—Pointer to item draw routine; NIL for none
promptRefDesc	Word—Defines type of reference in *promptRef*
– *promptRef* –	Long—Reference to Pascal string for file prompt
– *filterProcPtr* –	Long—Pointer to filter procedure; NIL for none
– *typelistPtr* –	Long—Pointer to type list record; NIL for none
– *dlgTempPtr* –	Long—Pointer to dialog template
– *dialogHookPtr* –	Long—Pointer to routine to handle item hits
– *replyPtr* –	Long—Pointer to multifile reply record
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$1701	`badPromptDesc`	Invalid *promptRefDesc* value

C

```
extern pascal void SFPMultiGet2(whereX, whereY,
          itemDrawPtr, promptRefDesc, promptRef,
          filterProcPtr, typelistPtr, dlgTempPtr,
          dialogHookPtr, replyPtr);

Pointer   itemDrawPtr, filterProcPtr, typelistPtr,
          dlgTempPtr, dialogHookPtr, replyPtr;
Word      whereX, whereY, promptRefDesc;
Long      promptRef;
```

itemDrawPtr Pointer to a custom item-drawing routine, as described in "Custom item drawing routines" earlier in this chapter.

promptRefDesc Defines the type of reference in *promptRef*:

$0000	Reference in *promptRef* is a pointer to a Pascal string.
$0001	Reference in *promptRef* is a handle of a Pascal string.
$0002	Reference in *promptRef* is the resource ID of a Pascal string

filterProcPtr Pointer to a new-style filter procedure, as described in "New filter procedure entry interface" earlier in this chapter.

dlgTempPtr, dialogHookPtr
For more information about these fields, see the discussion of the `SFPutFile` call in Chapter 22, "Standard File Operations Tool Set," in Volume 2 of the *Toolbox Reference.*

`SFPPutFile2` $1117

Displays a custom Save File dialog box and returns information about the file specification entered by the user. This call performs the same function as `SFPPutFile`, but uses Class 1 GS/OS calls, allowing the user to specify a full filename. In addition, this call does not support the *maxLen* parameter provided in `SFPPutFile`. This parameter allowed the calling program to limit the filename length.

Parameters

Stack before call

Previous contents	
whereX	Word—x coordinate of upper left corner of dialog box
whereY	Word—y coordinate of upper left corner of dialog box
– *itemDrawPtr* –	Long—Pointer to item draw routine; NIL for none
promptRefDesc	Word—Defines type of reference in *promptRef*
– *promptRef* –	Long—Reference to Pascal string for file prompt
origNameRefDesc	Word—Defines type of reference in *origNameRef*
– *origNameRef* –	Long—Reference to GS/OS Class 1 Input String with default name
– *dlgTempPtr* –	Long—Pointer to dialog template
– *dialogHookPtr* –	Long—Pointer to routine to handle item hits
– *replyPtr* –	Long—Pointer to new-style reply record
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$1701	`badPromptDesc`	Invalid *promptRefDesc* value
	$1702	`badOrigNameDesc`	Invalid *origNameDesc* value
	$1704	`badReplyNameDesc`	Invalid reply record `nameRefDesc` value
	$1705	`badReplyPathDesc`	Invalid reply record `pathRefDesc` value
	GS/OS errors		Returned unchanged

◆ *Note:* The GS/OS `bufferTooSmall` error can occur if the output strings you supply in the Reply record are too small to receive the resulting filename string. In this case, the buffer will contain as many name characters as would fit, and the length word will contain the name length the Standard File Operations Tool Set tried to return.

C

```
extern pascal void SFPPutFile2(whereX, whereY,
            itemDrawPtr, promptRefDesc, promptRef,
            origNameRefDesc, origNameRef, dlgTempPtr,
            dialogHookPtr, replyPtr);

Pointer     itemDrawPtr, dlgTempPtr, dialogHookPtr,
            replyPtr;
Word        whereX, whereY, promptRefDesc,
origNameRefDesc;
Long        promptRef, origNameRef;
```

itemDrawPtr Pointer to a custom item-drawing routine, as described in "Custom item drawing routines".

promptRefDesc Defines the type of reference in *promptRef*:

$0000	Reference in *promptRef* is a pointer to a Pascal string.
$0001	Reference in *promptRef* is a handle of a Pascal string.
$0002	Reference in *promptRef* is the resource ID of a Pascal string

origNameRefDesc Defines the type of reference in *origNameRef*:

$0000	Reference in *origNameRef* is a pointer to a GS/OS Class 1 Input string.
$0001	Reference in *origNameRef* is a handle to a GS/OS Class 1 Input string.
$0002	Reference in *origNameRef* is the resource ID to a GS/OS Class 1 Input string

origNameRef Reference to a GS/OS Class 1 Input string. On input to `SFPPutFile2`, this string contains the default filename for the Put operation. On output, this string contains the string confirmed by the user, which may not be the same as the default value.

dlgTempPtr, dialogHookPtr

For more information about these fields, see the discussion of the `SFPutFile` call in Chapter 22, "Standard File Operations Tool Set," in Volume 2 of the *Toolbox Reference.*

`SFPutFile2` $0F17

Displays the standard Save File dialog box and returns the file specification entered by the user. This call performs the same function as `SFPutFile`, but uses Class 1 GS/OS calls, allowing the user to specify a full filename. In addition, this call does not support the *maxLen* parameter provided in `SFPutFile`. This parameter allowed the calling program to limit the filename length.

Parameters

Stack before call

Previous contents	
whereX	Word—x coordinate of upper left corner of dialog box
whereY	Word—y coordinate of upper left corner of dialog box
promptRefDesc	Word—Defines type of reference in *promptRef*
– *promptRef* –	Long—Reference to Pascal string for file prompt
origNameRefDesc	Word—Defines type of reference in *origNameRef*
– *origNameRef* –	Long—Reference to GS/OS Class 1 Input String with default name
– *replyPtr* –	Long—Pointer to a new-style reply record
	<—SP

Stack after call

Previous contents	
	<—SP

Errors			
	$1701	`badPromptDesc`	Invalid *promptRefDesc* value
	$1702	`badOrigNameDesc`	Invalid *origNameDesc* value
	$1704	`badReplyNameDesc`	Invalid reply record `nameRefDesc` value
	$1705	`badReplyPathDesc`	Invalid reply record `pathRefDesc` value
	GS/OS errors		Returned unchanged

◆ *Note:* The GS/OS `bufferTooSmall` error can occur if the output strings you supply in the Reply record are too small to receive the resulting filename string. In this case, the buffer will contain as many name characters as would fit, and the length word will contain the name length the Standard File Operations Tool Set tried to return.

C

```
extern pascal void SFPutFile2(whereX, whereY,
          promptRefDesc, promptRef, origNameRefDesc,
          origNameRef, replyPtr);

Pointer   replyPtr;
Word      whereX, whereY, promptRefDesc,
origNameRefDesc;
Long      promptRef, origNameRef;
```

promptRefDesc Defines the type of reference in *promptRef*:

$0000	Reference in *promptRef* is a pointer to a Pascal string.
$0001	Reference in *promptRef* is a handle of a Pascal string.
$0002	Reference in *promptRef* is the resource ID of a Pascal string

origNameRefDesc Defines the type of reference in *origNameRef*:

$0000	Reference in *origNameRef* is a pointer to a GS/OS Class 1 Input string.
$0001	Reference in *origNameRef* is a handle of a GS/OS Class 1 Input string.
$0002	Reference in *origNameRef* is the resource ID of a GS/OS Class 1 Input string

origNameRef Reference to a GS/OS Class 1 Input string. On input to `SFPutFile2`, this string contains the default filename for the Put operation. On output, this string contains the string confirmed by the user, which may not be the same as the default value.

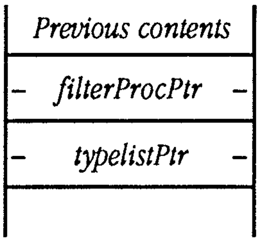
`SFReScan` $1317

Forces the system to re-build and re-display the current list of files. Your program may specify a new file type list or filter procedure.

Make this call only while `SFPGetFile`, `SFPGetFile2`, or `SFPMultiGet2` are running, and only from within a dialog hook routine (for information on dialog hook routines, see the description of `SFPGetFile` in Chapter 22, "Standard File Operations Tool Set," in Volume 2 of the *Toolbox Reference*).

Parameters

Stack before call

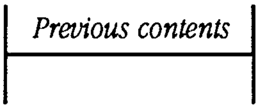


Long—Pointer to filter procedure; NIL for no change

Long—Pointer to type list record; NIL for no change

<—SP

Stack after call



<—SP

Errors	$1706	`badCall`	Neither `SFPGetFile`, `SFPGetFile2`, nor `SFPMultiGet2` is active

C

```
extern pascal void SFReScan(filterProcPtr,
          typelistPtr);

Pointer   filterProcPtr, typelistPtr;
```

`SFShowInvisible` $1217

Controls display of invisible files. When the Standard File Operations Tool Set initializes itself, invisible files are not displayed and are not passed to filter procedures.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
invisibleState	Word—Flag: 1 - display invisible files; 0 - no display (default)
	<—SP

Stack after call

Previous contents	
oldState	Word—Previous setting of invisible flag
	<—SP

Errors None

C

```
extern pascal word SFShowInvisible(invisibleState);

Word       invisibleState;
```

Standard File error codes

This section lists all valid Standard File error codes.

Value	Name	Definition
$1701	`badPromptDesc`	Invalid *promptRefDesc* value
$1702	`badOrigNameDesc`	Invalid *origNameDesc* value
$1704	`badReplyNameDesc`	Invalid reply record `nameRefDesc` value
$1705	`badReplyPathDesc`	Invalid reply record `pathRefDesc` value
$1706	`badCall`	Neither `SFPGetFile`, `SFPGetFile2`, nor `SFPMultiGet2` is active

Chapter 49 **TextEdit Tool Set**

This chapter documents the features of the new TextEdit tool set. This is a new tool set, not previously documented in the *Apple IIGS Toolbox Reference.*

About the TextEdit Tool Set

The TextEdit Tool Set provides basic text editing capabilities for any application. You can use TextEdit to support anything from a simple text-based dialog box to a complete text editor. While it has been loosely based on the Macintosh tool, TextEdit for the Apple IIGS includes many enhancements that expand both the flexibility and functionality of the tool set.

TextEdit for the Apple IIGS supports a number of capabilities and features, including:

- Text insertion, deletion, selection, copying, cutting and pasting, using standard keyboard and mouse interfaces
- Editing very large documents, up to the limit of system memory
- Word wrap, which avoids splitting words at the right text edge
- Optional support for intelligent cut and paste, which eliminates the need for the user to add or remove extra spaces after pasting word-based selections
- Style variations in the text, affecting text font, style, size or color
- Formatting for margins, indentation, justification, and tabs
- Left-justified tabs, either evenly spaced or at specified locations
- Vertical scrolling of text that extends beyond the current display window
- Vertical scroll bar

The following several paragraphs discuss general TextEdit concepts, and present more detail on some TextEdit features.

TextEdit provides your program with the facilities to create, display, and manage one or more windows of editable text. These windows can be controls (such as the text in a dialog box or the text window for an editor) or independently managed by your application. The Control Manager and the Window Manager help your program manage TextEdit controls; your program is solely responsible for TextEdit windows that are not controls. All TextEdit windows, whether or not they are controls, are called **records** or **TextEdit records**.

Since there can be many TextEdit records displayed at the same time, TextEdit provides a mechanism for distinguishing between them that is based upon the mechanism used by the Control Manager to move among controls on the screen. The current or active TextEdit record is referred to as the **target** record. That record receives all user keystrokes and mouse commands. The user can switch between TextEdit records by pressing the Tab key (if your program has enabled that option) or by clicking in the appropriate record.

TextEdit maintains a number of data structures for each record. The `TERecord` is the main structure for a TextEdit record. All control information needed for the record is either stored in, or is accessible through, the `TERecord`. In general, your program need not access or modify the `TERecord` unless you want to use some of TextEdit's internal features. Your program creates a TextEdit record, and its `TERecord`, by formatting a `TEParamBlock` and passing that structure to the `TENew` tool call. The `TERecord` is an extension of the generic control record defined by the Control Manager.

For each TextEdit record, your program can instruct TextEdit to use intelligent (or smart) cut and paste. The goal of intelligent cut and paste is to eliminate the need for the user to insert spaces to fix a paste. With intelligent cut and paste enabled, TextEdit will make the following adjustments to the current selection:

- When text is cut, remove all leading spaces; if there are no leading spaces, remove all trailing spaces
- When text is pasted, if the current selection is adjacent to a nonspace character, TextEdit first inserts a space, then the text

By making these adjustments, intelligent cut and paste allows the user to select a word (by double-clicking the mouse, for instance), and cut and paste that selected text without adding or removing any space characters. Your program specifies intelligent cut and paste by setting a bit flag in the `textFlags` field of the `TEParamBlock` used to create the record.

TextEdit supports four types of text justification: left, center, right, and full. Left-justified text lies flush with the left margin, with ragged right edges. Right-justified text is flush with the right margin, with ragged left edges. Each line of centered text is centered between the left and right margins. Fully justified text is blocked flush with both left and right margins; TextEdit pads spaces with extra pixels to fully justify the text. Your program specifies the type of justification for a TextEdit record as part of the initial style information in the `TEParamBlock` for the record. Your program can change the text justification for a record with the `TESetRuler` tool call.

TextEdit supports tabs in two ways. Regular tabs are spaced evenly in the text, at consistent pixel intervals. Absolute tabs reside at specified pixel locations and can be spaced irregularly in the text. All TextEdit tabs are left justified. Your program specifies whether a TextEdit records supports tabs and, if so, the type and spacing for those tabs in the `TEParamBlock` for the record. Your program can change the tabs for a record with the `TESetRuler` tool call.

TextEdit call summary

The following table summarizes the capabilities of the TextEdit Tool Set in a grouped listing of its tool calls. Later sections of this chapter discuss TextEdit, and its capabilities and data structures in greater detail, and define the precise syntax for the TextEdit tool calls.

Routine	Description
Housekeeping routines	
`TEBootInit`	Initializes TextEdit; called only by the Tool Locator
`TEStartUp`	Allocates TextEdit facilities for an application—must precede any other TextEdit tool calls
`TEShutDown`	Frees TextEdit facilities used by an application—TextEdit applications must issue this call before quitting
`TEVersion`	Returns TextEdit version number
`TEReset`	Resets TextEdit; called only when the system is reset
`TEStatus`	Returns status information about TextEdit
Record and text management	
`TENew`	Allocates a new TextEdit record
`TEKill`	Disposes of an old TextEdit record
`TESetText`	Sets the text for an existing TextEdit record
`TEGetText`	Returns the text from an existing TextEdit record
`TEGetTextInfo`	Returns information about the text in a TextEdit record
Insertion point and selection range	
`TEIdle`	Provides processor time to TextEdit so that it can blink the cursor and perform background tasks
`TEActivate`	Activates the current selection
`TEDeactivate`	Deactivates the current selection
`TEClick`	Activates a specified TextEdit record, and selects text within that record
`TEUpdate`	Redraws a TextEdit record

Editing

`TEKey`	Inserts a character into the target TextEdit record
`TECut`	Cuts the current selection and places it into the desk scrap
`TECopy`	Copies the current selection into the desk scrap
`TEPaste`	Pastes the contents of the desk scrap into the text, replacing the current selection
`TEClear`	Clears the current selection
`TEInsert`	Inserts the specified text before the current selection
`TEReplace`	Replaces the current selection with the specified text
`TEGetSelection`	Returns the starting and ending character offsets for the current selection
`TESetSelection`	Sets the current selection to the specified starting and ending character offsets

Text display and scrolling

`TEGetSelectionStyle`	Returns style information for the current selection
`TEStyleChange`	Changes the style of the current selection
`TEGetRuler`	Returns format information for a TextEdit record
`TESetRuler`	Sets format information for a TextEdit record
`TEScroll`	Scrolls to a specified line, text offset, or pixel position
`TEOffsetToPoint`	Converts a text offset into a point (in local coordinates)
`TEPointToOffset`	Converts a point (in local coordinates) into a text offset
`TEPaintText`	Paints TextEdit text into an off-screen port—used for printing

Miscellaneous Routines

`TEGetDefProc`	Returns a pointer to the TextEdit custom control definition procedure
`TEGetInternalProc`	Returns a pointer to the TextEdit low-level routine dispatcher
`TEGetLastError`	Returns the last error code generated for a TextEdit record
`TECompactRecord`	Compresses the data structures associated with a TextEdit record

How to use TextEdit

You may choose between several techniques for creating and controlling TextEdit records

- Create a TextEdit control with the `NewControl2` Control Manager tool call, and allow TaskMaster to manage the control for you
- Create a TextEdit control with the `NewControl2` tool call and manage the control yourself
- Create a TextEdit record that is not a control with the `TENew` TextEdit tool call and manage it yourself

The remainder of this section discusses each of these techniques in more detail.

The simplest technique is to create a TextEdit control, using either the `TENew` TextEdit tool call or the `NewControl2` Control Manager call, and use TaskMaster to manage the record (see Chapter 25, "Window Manager," in the *Toolbox Reference* for more information on TaskMaster). TaskMaster handles all TextEdit events and user interaction for single-style records. The following pseudo code describes the basic program logic for this technique.

```
Initialize all the tools including TextEdit.
Create a new window.
Call TENew. This allocates a new TextEdit record.
while( quitFlag != TRUE )
{
      Call TaskMaster. This handles all the events; it inserts all the
            keys that the user types, handles all the mouse activity and
            takes care of blinking the cursor. It even calls TECut,
            TECopy, TEPaste, and TEClear for the TextEdit record.
      if( the user selects the save item )
      {
            Call TEGetText. This extracts the text and style information
                  that the user has typed.
      }
}
Dispose of the window. This deallocates the TextEdit record and all
      other controls in the window.
Shutdown all the tools and exit.
```

Your application does not need to do anything if the user presses a key, clicks the mouse button, or selects a menu item. TaskMaster and the TextEdit control definition procedure handle all these standard events. The Window Manager will dispose of the TextEdit control when your application closes its window.

However, your program does give up some flexibility in exchange for the simplicity of this approach. In order to exert somewhat more control over the TextEdit record, you may choose to create a TextEdit control and manage that control in your program, rather than with TaskMaster. Your program would then issue the `GetNextEvent` Window Manager call to trap user actions, and then process those actions accordingly. The following pseudo code shows sample logic for this approach.

```
Initialize all the tools including TextEdit.
Create a new window.
Call TENew. This allocates a new TextEdit record.
while( quitFlag != TRUE )
      {
            Call TEIdle. This blinks the cursor and performs background
                  tasks.
            Call GetNextEvent.
            switch theEvent.what
            {
            case updateEvent:
                  Call DrawControls. This draws the TextEdit control
                        (and all others in the window)
            case mouseDownEvent:
                  Call FindWindow. This determines where in the
                        desktop the mouse was clicked.
                  if( FindWindow returned inMenu )
                  {
                        call MenuSelect. This tracks the menu and
                              returns which item the user clicked in.
                        switch theMenuItem
                        {
                        case CutItem:
                              Call TECut. This tells TextEdit to cut
                                    the current selection into the desk
                                    scrap.
                        case CopyItem:
                              Call TECopy. This tells TextEdit to copy
                                    the current selection into the desk
                                    scrap.
                        case PasteItem:
                              Call TEPaste. This tells TextEdit to
                              replace the current selection with the
                              desk scrap.
                        case ClearItem:
                              Call TEClear. This tells TextEdit to
                                    clear the current selection.
                        case SaveItem:
                              Call TEGetText. This extracts the text
                                    and style information that the user
                                    has typed.
                        case QuitItem:
                              Set the quitFlag to TRUE.
                        }
                  }
                  else if( FindWindow returned inContent )
                  {
                        Call FindControl. This returns which control
                              was clicked in.
                        if( FindControl returned the TextEdit control )
                        {
                              call TEClick. This tracks the mouse
                                    within the TextEdit record; it does
                                    all the selecting and all the
                                    scrolling.
                        }
                  }
                  else
                  {
                        ...
                  }
            case keyDownEvent,autoKeyEvent:
                  Call TEKey. This inserts the key that the user typed
                        into the TextEdit record. It also performs
                        editing operations if the key is a "control key"
                        (like DELETE, CTRL-Y, arrow keys, etc).
            }
      }
Dispose of the window. This deallocates the TextEdit record and all
other controls in the window.
Shutdown all the tools and exit.
```

Finally, you may choose to create TextEdit records that are not controls. In this case, your program must not only provide complete functional support for the record, as shown in the non-TaskMaster pseudo code, but also manage the TextEdit window itself. You must use the `TENew` call to create TextEdit records that are not controls. These TextEdit records are not inserted into the control list, so your program may not issue Control Manager calls to manipulate or control them. Similarly, your program may not use Window Manager calls on them. The following pseudo code presents sample logic for this approach.

```
Initialize all the tools including TextEdit.
Create a new window.
Call TENew. This allocates a new TextEdit record.
while( quitFlag != TRUE )
{
      Call TEIdle. This blinks the cursor and performs background
            tasks.
      Call GetNextEvent.
      switch theEvent.what
      {
      case updateEvent:
            Call TEUpdate. This draws the TextEdit record.
      case mouseDownEvent:
            Call FindWindow. This determines where in the desktop the
                  mouse was clicked.
            if( FindWindow returned inMenu )
            {
                  call MenuSelect. This tracks the menu and returns
                        which item the user clicked in.
                  switch theMenuItem
                  {
                  case CutItem:
                        Call TECut. This tells TextEdit to cut the
                              current selection into the desk scrap.
                  case CopyItem:
                        Call TECopy. This tells TextEdit to copy the
                              current selection into the desk scrap.
                  case PasteItem:
                        Call TEPaste. This tells TextEdit to replace
                              the current selection with the desk scrap.
                  case ClearItem:
                        Call TEClear. This tells TextEdit to clear the
                              current selection.
            case SaveItem:
                Call TEGetText. This extracts the text and
                    style information that the user has typed.
            case QuitItem:
                Set the quitFlag to TRUE.
            }
        }
        else if( FindWindow returned inContent )
        {
            Figure out whether the click occurred in the TextEdit
                record.
            if( the click occurred in the TextEdit record )
            {
                call TEClick. This tracks the mouse within the
                    TextEdit record; it does all the selecting
                    and all the scrolling.
            }
        }
        else
        {
            ...
        }
    case keyDownEvent,autoKeyEvent:
        Call TEKey. This inserts the key that the user typed into
            the TextEdit record. It also performs editing
            operations if the key is a "control key" (like DELETE,
            CTRL-Y, arrow keys, etc).
    }
}
Dispose of the window. This deallocates the TextEdit record and all
other controls in the window.
Shutdown all the tools and exit.
```

With this technique, your program has much more responsibility. First, your program must issue the `TEUpdate` call for each record that must be redrawn, rather than relying on the Control Manager `DrawControls` tool call. In addition, your program must use the `TEActivate` and `TEDeactivate` tool calls whenever the user switches between TextEdit records. Finally, for each mouse-down event, your program must determine in which TextEdit record the user clicked—`FindControl` will not work with TextEdit records that are not controls.

▲ **Warning** If you have defined TextEdit records that are controls in a window, you must not also try to define non-control TextEdit records in the same window. ▲

All TextEdit tool calls require you to specify a handle to the appropriate `TERecord`, so that TextEdit knows which record to address. For TextEdit records that are controls, your program may specify a NULL value for the `TERecord` handle. TextEdit will then access the currently active TextEdit control (the target TextEdit record).

▲ **Warning** Never pass a NULL `TERecord` handle to access TextEdit records that are not controls. ▲

Note that TextEdit routines always use the same `TERecord` layout, whether or not the TextEdit record is a control. However, if the TextEdit record is not a control, your program cannot issue Control Manager tool calls for it.

Standard TextEdit key sequences

TextEdit provides a keyboard and mouse interface that supports a number of editing keys. The following table summarizes the effect of control keystrokes and mouse clicks:

Key	Alias	Description
Left Arrow	Control-h	Moves the insertion point to the previous character in the text Command key causes movement by word rather than by character Option key moves the insertion point to the beginning of the previous line in the text Shift key extends the selection from the current insertion point back by a character, word (if the Command key is also held down), or line (if the Option key is also held down)

Right Arrow	Control-u	Moves the insertion point to the next character in the text Command key causes movement by word rather than by character Option key moves the insertion point to the end of the current line in the text Shift key extends the selection from the current insertion point forward by a character, word (if the Command key is also held down), or line (if the Option key is also held down)
Up Arrow	Control-k	Moves the insertion point up one line Command key moves the insertion point to the beginning of the current page Option key moves the insertion point to the beginning of the document Shift key extends the selection from the current insertion point up by a line or page (if the Command key is also held down), or to the beginning of the document (if the Option key is also held down)
Down Arrow	Control-j	Moves the insertion point down one line Command key moves the insertion point to the current column position on the last line of the page Option key moves the insertion point to the end of the document Shift key extends the selection from the current insertion point down by a line or page (if the Command key is also held down), or to the end of the document (if the Option key is also held down)
Delete	Control-d	If there is no current selection, removes the character to the left of the insertion point; if there is a selection, removes the selected text
Clear		Clears the current selection (does nothing if there is no selection)
Control-f		If there is no current selection, removes the character to the right of the insertion point; if there is a selection, removes the selected text
Control-y		Removes all characters from the insertion point to the end of the line, not including any terminating ASCII return characters ($0D)

Control-x	Cuts the current selection into the Clipboard (same as the `TECut` tool call)
Control-c	Copies the current selection into the Clipboard (same as the `TECopy` tool call)
Control-v	Pastes the contents of the clipboard at the current insertion point, or in place of any selected text (same as the `TEPaste` tool call)
Click	Moves the insertion point—dragging selects by character
Double-click	Selects a word—dragging extends the selection by words
Triple-click	Selects a line—dragging extends the selection by lines

Internal structure of the TextEdit Tool Set

This section discusses several topics relating to the internal structure and function of the TextEdit Tool Set. This information will not be relevant to most application programmers, but does provide insight into how TextEdit operates, and how to tailor TextEdit for special applications.

TextEdit controls and the Control Manager

As has been discussed, TextEdit records may or may not be controls. Your program creates TextEdit controls by issuing the `NewControl2` Control Manager tool call. The Control Manager handles nearly all the support calls needed to maintain the TextEdit record. However, you may choose to use certain Control Manager tool calls with the TextEdit control. The following tables list which Control Manager calls may or may not be used with TextEdit controls. In this context, the TextEdit control is taken to include the actual TextEdit record and its associated scroll bars and size box.

The following Control Manager calls may be used with TextEdit controls:

Call	Description
`DisposeControl`	Disposes of the TextEdit control—analogous to `TEKill` TextEdit tool call
`KillControls`	Disposes of all controls, including the TextEdit controls—analogous to `TEKill` tool calls for each control
`HideControl`	Hides the TextEdit control—note that this call does not affect the status of the control with respect to user keystokes; if you hide the target control, it is still the target control, although no user keystrokes will be displayed
`EraseControl`	Erases the TextEdit control—similar to `HideControl`, except that `EraseControl` does not invalidate the boundary rectangle for the control
`ShowControl`	Reshows the TextEdit control, reversing the effect of `HideControl` or `EraseControl`
`DrawControls`	Draws all controls in the window
`DrawOneCtl`	Draws the specified TextEdit control

`HilightControl` Activates or deactivates the TextEdit control—note that only *hiliteState* values of 0 and 255 are valid

`FindControl` Returns point location control identification information—returns *partCode* of 130 if point is in text, 131 if point is in vertical scroll bar, and 132 if point is in size box

`TestControl` Returns the same point location information as `FindControl`, without any control identification

`TrackControl` Selects text—*actionProcPtr* must be set to a negative value (forces the Control Manager to use TextEdit's built-in action procedure)

`MoveControl` Moves the TextEdit control

`DragControl` Allows the user to reposition the TextEdit control

`SetCtlRefCon` Sets the *refCon* field

`GetCtlRefCon` Returns the contents of the *refCon* field

Your program must not issue the following Control Manager tool calls with a TextEdit control:

```
GetCtlTitle
GetCtlValue
GetCtlAction
GetCtlParams
SetCtlTitle
SetCtlValue
SetCtlAction
SetCtlParams
```

TextEdit filter procedures and hook routines

TextEdit provides you with several mechanisms to tailor the operation of the tool set to the particular needs of your application. Filter procedures give you a chance to affect the behavior of the tool set by modifying screen display activity or user keystrokes. Hook routines allow you to replace standard TextEdit code for such functions as word wrap or word break. The following several sections discuss each of the various filter procedures and hook routines in more detail.

Generic filter procedure

TextEdit provides a facility whereby your application can supply special logic to replace some of the standard TextEdit routines. The program code that uses this facility is called a generic filter procedure. The generic filter procedure is, in turn, made up of several routines that address particular TextEdit processing requirements. At present, there are three functions that generic filter routines can provide:

- Erasing rectangles in the display port
- Erasing rectangles in the off-screen TextEdit buffer
- Updating the TextEdit record screen display

The `TERecord` `filterProc` field contains a pointer to the generic filter procedure. If this field has a non-NULL value, TextEdit will call the filter procedure to perform the activities just mentioned. You set this pointer by specifying the appropriate value in the `filterProcPtr` field of the `TEParamBlock` passed to `TENew` (or `NewControl2`) when you create the TextEdit record. TextEdit then loads the `filterProc` field of the `TERecord` from this value.

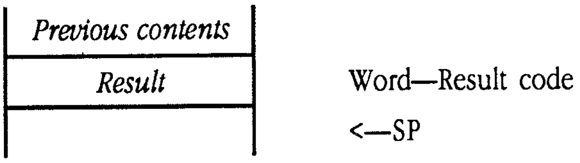
The routines in the filter procedure must adhere to the following rules:

- The routine must preserve the direct page and data bank registers, and must return in full native mode.
- All entry and exit parameter and status fields must be passed through the appropriate `TERecord`.
- Filter routines must not issue TextEdit tool calls, move memory, or cause memory to be moved.
- Any application-specific routine messages must have message numbers greater than $8000—TextEdit reserves all message number values in the range from $0000 through $7FFF.

TextEdit invokes the generic filter procedure in full native mode by executing a `JSL`. On entry to the filter procedure, the stack is formatted as follows:

Previous contents	
Space	Word—Space for result
– *teHandle* –	Long—Handle to the appropriate `TERecord`
message	Word—Message number indicating action to take
– *RTL* –	Three bytes—Return address
	<—SP

On exit, the filter procedure must format the stack as follows:



result Indicates whether the filter procedure handled the message. If the field is set to $0000, then TextEdit will perform its standard processing. If the field is nonzero, then the filter procedure handled the message, and TextEdit does not perform its standard processing.

The following sections discuss each defined filter procedure message, defining the actions the filter procedure is to take and the affected `TERecord` fields.

`doEraseRect` ($0001)

The filter procedure is to erase a rectangle in the display port for the current TextEdit record. TextEdit has already set up the port for the filter routine.

TextEdit provides this routine to support applications that maintain overlaying objects on the display. Under such circumstances, the application must decide what object to make visible after the user has caused a currently visible object to be deleted.

Input:

`TERecord` field	Contents
`theFilterRect`	Specifies the rectangle to erase, expressed in local coordinates for the port

Output:

None

doEraseBuffer ($0002)

The filter procedure is to move a rectangle from TextEdit's offscreen buffer to the display port. The TERecord contains information defining the source and destination data locations. TextEdit has already set up the port for the filter routine.

This routine is used in much the same way as doEraseRect, except that it operates off-screen, rather than on-screen.

Input:

TERecord field	Contents
theFilterRect	Specifies the rectangle to erase, expressed in local coordinates for the offscreen buffer port
theBufferVPos	Vertical position corresponding to the top of the destination buffer in the display port, expressed in local coordinates for the port
theBufferHPos	Horizontal position corresponding to the left edge of the destination buffer in the display port, expressed in local coordinates for the port

Output:

None

`doRectChanged` ($0003)

The filter procedure is to handle a change to the display window for the TextEdit record. Note that TextEdit has not set up the port; the filter procedure must obtain the port from the `inPort` field of the `TERecord`, and set up the display port.

With this routine your application can maintain an off-screen copy of its TextEdit display. Whenever TextEdit updates the screen, it issues this message to the generic filter procedure, which can update the saved screen copy.

Input:

`TERecord` field	Contents
`theFilterRect`	Specifies the rectangle that changed, expressed in local coordinates for the port

Output:

None

Keystroke filter procedure

TextEdit also provides a mechanism for applications to supply custom keystroke processing for a TextEdit record. TextEdit's internal keystroke routine supports the standard keyboard interface described in "Standard TextEdit key sequences" in this chapter. Custom keystroke processing may support different keyboard mappings or macro facilities.

The `keyFilter` field of a `TERecord` can contain a pointer to a keystroke filter procedure. If `keyFilter` is NULL, TextEdit uses its standard keystroke routine. If `keyFilter` is non-NULL, TextEdit calls the routine pointed to by `keyFilter` to process all user keyboard input.

Keystroke filter procedures must follow many of the same rules established for generic filter procedures:

- The procedure must preserve the direct-page and data bank registers, and must return in full native mode.
- Keystroke filter procedures must not issue TextEdit tool calls.

TextEdit invokes the keystroke filter procedure in full native mode by executing a `JSL`. Fields in the `KeyRecord` `TERecord` sub-structure contain information defining the data to be processed:

`KeyRecord` field	Contents
`theChar`	This field contains the keystroke to process.
`theModifiers`	This field contains flags indicating the state of the modifier keys at the time the key was pressed; the keystroke is contained in `theChar` and `theInputHandle`.
`theInputHandle`	Handle to a copy of `theChar`.

TextEdit loads additional control information onto the stack. On entry to the filter procedure, the stack is formatted as follows:

Previous contents	
– *teHandle* –	Long—Handle to the appropriate `TERecord`
type	Word—Type of data to be processed
– *RTL* –	Three bytes—Return address
	<—SP

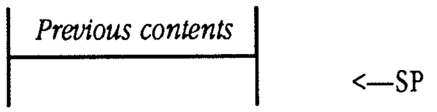
type Indicates the type of data to be processed:

$0001 `KeyRecord theChar` field contains the single character to be processed

$0002 Reserved

The keystroke filter procedure is now free to perform whatever processing is appropriate. For example, it may change the input keystroke value in order to support a different mapping of the standard TextEdit keyboard functions. Or, the routine may expand the keystroke in `theChar` into a block of text to be inserted at the current location. The routine then formats the appropriate return data into the `KeyRecord` fields and returns control to TextEdit by issuing an `RTL` instruction (after removing all input parameters from the stack).

On exit, the filter procedure must format the stack as follows:



The filter procedure indicates what action TextEdit is to take with the `theOpCode` field in the `KeyRecord`. This code in turn governs how TextEdit interprets the remainder of the returned `KeyRecord`. Valid `theOpCode` values are:

`theOpCode`	Value	TextEdit action
`opNormal`	$0000	TextEdit performs its standard processing on the character stored in the location referred to by `theInputHandle`
`opNothing`	$0002	TextEdit ignores the keystroke
`opReplaceText`	$0004	TextEdit inserts the text referred to by `theInputHandle` in place of the current selection in the record; if there is no current selection, TextEdit inserts the text at the current insertion point—if `theInputHandle` has 0 size, TextEdit deletes the current selection and inserts nothing
`opMoveCursor`	$0006	TextEdit moves the cursor to the location specified by `cursorOffset`
`opExtendCursor`	$0008	TextEdit extends the current selection from its anchor point to the location specified by `cursorOffset`
`opCut`	$000A	TextEdit moves the current selection and to the desk scrap
`opCopy`	$000C	TextEdit copies the current selection to the desk scrap
`opPaste`	$000E	TextEdit inserts the contents of the desk scrap in place of the current selection
`opClear`	$0010	TextEdit clears the current selection

The following table summarizes the contents of the KeyRecord on return from the keystroke filter procedure:

KeyRecord field	Contents
theChar	Unchanged
theModifiers	Contains changed modifiers (only for opNormal)
theInputHandle	Handle to replacement text (only for opNormal and opReplaceText), length of text indicated by size of handle;
cursorOffset	If TextEdit is to move the cursor (theOpCode is set to either opMoveCursor or opExtendCursor), this field contains the new cursor location; otherwise, TextEdit ignores this field.
theOpCode	Specifies next action for TextEdit

Word wrap hook

Your program may supply its own routine to handle word wrap. This word wrap hook routine determines whether a character string typed by the user fits on the current line (does not wrap) or needs to begin a new line (does wrap). TextEdit then displays the character string on the appropriate line.

TextEdit determines whether to call a custom word wrap routine by examining the `wordWrapHook` `TERecord` field for the TextEdit record. If that field is NULL, TextEdit uses its standard word wrap routine. If that field is non-NULL, TextEdit calls the routine pointed to by that field whenever a word wrap decision is required. Your program sets this pointer by directly modifying the `wordWrapHook` field of the appropriate `TERecord`.

Word wrap hook routines must follow many of the rules established for generic filter procedures:

- The routine must preserve the direct-page and data bank registers, and must return in full native mode.
- Word wrap routines must not issue TextEdit tool calls, move memory, or cause memory to be moved.

TextEdit invokes the word wrap hook procedure in full native mode by executing a `JSL`. Entry parameters refer the procedure to the correct `TERecord` and character. On exit, the word wrap procedure returns a flag indicating whether the character caused a word wrap.

On entry to the procedure, the stack is formatted as follows:

Previous contents	
Space	Word—Space for result
– *teHandle* –	Long—Handle to the appropriate `TERecord`
theChar	Word—Character to check
– *RTL* –	Three bytes—Return address
	<—SP

On exit, the filter procedure must format the stack as follows:

Previous contents	
wrapFlag	Word—Flag indicating wrap status
	<—SP

wrapFlag	Indicates wrap status for the current character:
$0000	Not a word wrap (TextEdit will leave the word on the current line)
$FFFF	Word wrap (TextEdit will move the word to the next line)

Word break hook

Your program may supply its own routine to determine word breaks when the user is selecting text by words (user has double-clicked on a word and is now extending that selection). This word break hook routine decides whether the cursor resides at a break between words. If so, TextEdit includes the next word in the current selection.

TextEdit determines whether to call a custom word wrap routine by examining the `wordBreakHook` `TERecord` field for the TextEdit record. If that field is NULL, TextEdit uses its standard word break routine. If that field is non-NULL, TextEdit calls the routine pointed to by that field whenever a word break decision is required. Your program sets this pointer by directly modifying the `wordBreakHook` field of the appropriate `TERecord`.

Word break hook routines must follow many of the rules established for generic filter procedures:

- The routine must preserve the direct-page and data bank registers, and must return in full native mode.
- Word break routines must not issue TextEdit tool calls, move memory, or cause memory to be moved.

TextEdit invokes the word break hook procedure in full native mode by executing a `JSL`. Entry parameters refer the procedure to the correct `TERecord` and character. On exit, the word break procedure returns a flag indicating whether the character constitues a word break.

On entry to the procedure, the stack is formatted as follows:

Previous contents	
Space	Word—Space for result
– *teHandle* –	Long—Handle to the appropriate `TERecord`
theChar	Word—Character to check
– *RTL* –	Three bytes—Return address
	<—SP

On exit, the filter procedure must format the stack as follows:

Previous contents	
breakFlag	Word—Flag indicating break status
	<—SP

breakFlag Indicates break status for the current character:

$0000	Not a word break (TextEdit will not extend the selection)
$FFFF	Word break (TextEdit will extend the selection to include next word)

Custom scroll bars

Your program may specify a custom scroll bar for vertical scrolling of a TextEdit record. Use the `vertBar` field of the `TEParamBlock` record to specify a handle to the control record for the custom scroll bar. This scroll bar need not reside in the TextEdit record display port, but it must follow certain rules with respect to the format and content of its control record (see Chapter 28, "Control Manager Update," in this book and Chapter 4, "Control Manager," in the *Toolbox Reference* for details on scroll bar controls and control records):

- Fields corresponding to the `dataSize`, `viewSize`, and `ctlValue` fields of a standard scroll bar control record must be located at the same relative locations within the custom control record.
- TextEdit stores a handle to the appropriate `TERecord` in the `ctlRefCon` field of the scroll bar control record. Do not change the contents of this field. If you need to store additional information in the scroll bar control record, extend the record handle and format that data after the standard control record fields (be sure to extract the size of the returned handle, rather than relying on current record definitions for the size of the control record).
- TextEdit stores a pointer to its internal text scroll routine in the `ctlAction` field ofthe scroll bar control record when the TextEdit record is created (during execution of the `TENew` or `NewControl2` tool call). Your program may change the contents of this field, but should save the pointer, so that you can call the TextEdit text scroll routine when appropriate. For information on the interface to action routines, see"Track Control" in Chapter 4, "Control Manager," of the *Toolbox Reference.*

Refer to Chapter 4, "Control Manager," for background information on writing and invoking control definition procedures.

TextEdit data structures

This section defines and discusses the various data structures used by the TextEdit Tool Set. The TextEdit data structures are divided into high-level data structures and low-level data structures. High-level TextEdit data structures are of interest to all application programmers who use TextEdit facilities. Low-level TextEdit data structures, on the other hand, are not relevant to most application programmers. However, if your program uses the low-level features of TextEdit, or must for some other reason directly access TextEdit data structures, you should familiarize yourself with the information in that section.

In most cases, it is not necessary for your program to directly modify fields in these structures. However, if your program manipulates TextEdit structures, note that many of these data structures refer to or depend upon one another. Whenever your application changes one of these structures, you must be careful to update other affected or dependent structures.

High-level TextEdit structures

TextEdit uses a number of structures to store information about a TextEdit record and to pass that information to TextEdit tool calls. The following sections define the format and content of each of these structures, and describe how your program would use them.

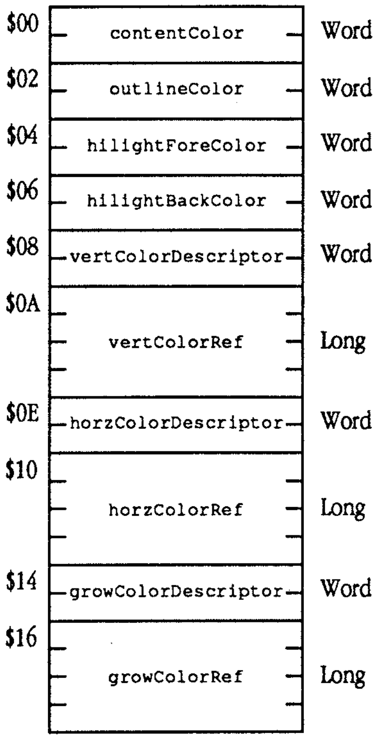
`TEColorTable`

Defines color attributes for a TextEdit record.

The `TEParamBlock` and `TERecord` contain references to color tables stored in `TEColorTable` format.

Note that all bits in TextEdit color words (such as `contentColor`) are significant. TextEdit generates QuickDraw II color patterns by replicating a color word the appropriate number of times for the current resolution (8 times for 640 mode, 16 times for 320 mode). See Chapter 16, "QuickDraw II," for more information on QuickDraw II patterns and dithered colors.

■ **Figure 49-1** `TEColorTable` layout



$00	contentColor	Word
$02	outlineColor	Word
$04	hilightForeColor	Word
$06	hilightBackColor	Word
$08	vertColorDescriptor	Word
$0A	vertColorRef	Long
$0E	horzColorDescriptor	Word
$10	horzColorRef	Long
$14	growColorDescriptor	Word
$16	growColorRef	Long

`contentColor` Defines the color for the entire boundary rectangle (in essence, defining the background color for the text window)

`outlineColor` Defines the color for box that surrounds the text in the display window.

`hilightForeColor`
Defines the foreground color for highlighted text.

`hilightBackColor`
Defines the background color for highlighted text.

vertColorDescriptor

Defines the type of reference stored in vertColorRef:

refIsPointer	$0000	The vertColorRef field contains a pointer to the color table for the vertical scroll bar
refIsHandle	$0004	The vertColorRef field contains a handle to the color table for the vertical scroll bar
refIsResource	$0008	The vertColorRef field contains the resource ID for the color table for the vertical scroll bar

vertColorRef

Reference to the color table for the vertical scroll bar. The vertColorDescriptor field indicates the type of reference stored here. This field must refer to a scroll bar color table, as defined in Chapter 4, "Control Manager," of the *Toolbox Reference.*

horzColorDescriptor

Defines the type of reference stored in horzColorRef:

refIsPointer	$0000	The horzColorRef field contains a pointer to the color table for the horizontal scroll bar
refIsHandle	$0004	The horzColorRef field contains a handle to the color table for the horizontal scroll bar
refIsResource	$0008	The horzColorRef field contains the resource ID for the color table for the horizontal scroll bar

horzColorRef Reference to the color table for the horizontal scroll bar. *horzColorDescriptor* indicates the type of reference stored here. This field must refer to a Scroll bar color table, as defined in Chapter 4, "Control Manager," of the *Toolbox Reference.*

growColorDescriptor

Defines the type of reference stored in growColorRef:

refIsPointer	$0000	The growColorRef field contains a pointer to the color table for the size box
refIsHandle	$0004	The growColorRef field contains a handle to the color table for the size box
refIsResource	$0008	The growColorRef field contains the resource ID for the color table for the size box

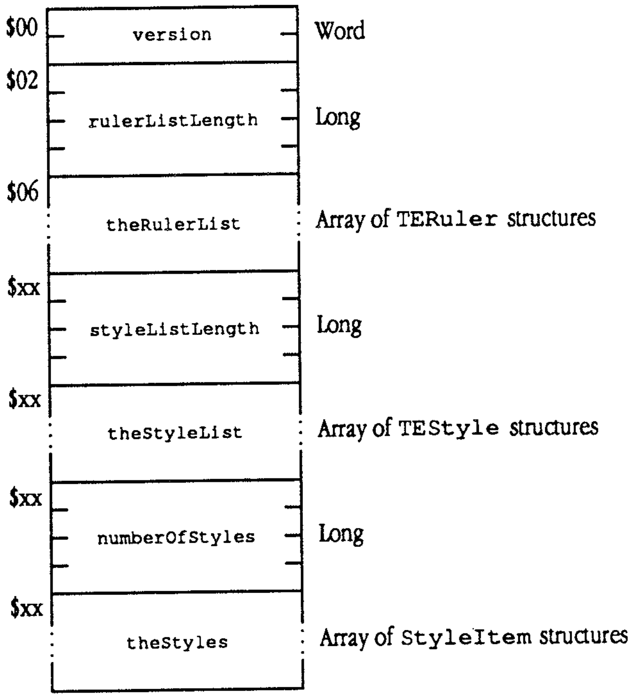
`growColorRef` Reference to the color table for the size box. The `growColorDescriptor` field indicates the type of reference stored here. This field must refer to a size box color table, as defined in Chapter 4, "Control Manager," of the *Toolbox Reference.*

TEFormat

This structure stores text formatting control information. From this structure, you can access the rulers and styles defined for the text.

Many TextEdit tool calls use this structure to accept or return format data for a text record.

- **Figure 49-2** TEFormat layout



`version` Version number corresponding to the layout of this `TEFormat` structure. This version of the structure carries a version number of $0000.

`rulerListLength`
The length of `theRulerList` in bytes.

`theRulerList` Ruler data for the text record. The `TERuler` structure is embedded within the `TEFormat` structure at this location.

`styleListLength`
The length of `theStyleList` in bytes.

`theStyleList` List of all unique styles for the text record. The `TEStyle` structures are embedded within the `TEFormat` structure at this location. Each `TEStyle` structure must define a unique style—there must be no duplicate style entries. In order to apply the same style to multiple blocks of text, you should create additional `StyleItems` for each block of text, and make each item refer to the same style in this array.

`numberOfStyles`
The number of `StyleItems` contained in `theStyles`.

`theStyles` Array of `StyleItems` specifying which actual styles (stored in `theStyleList`) apply to which text within the TextEdit record.

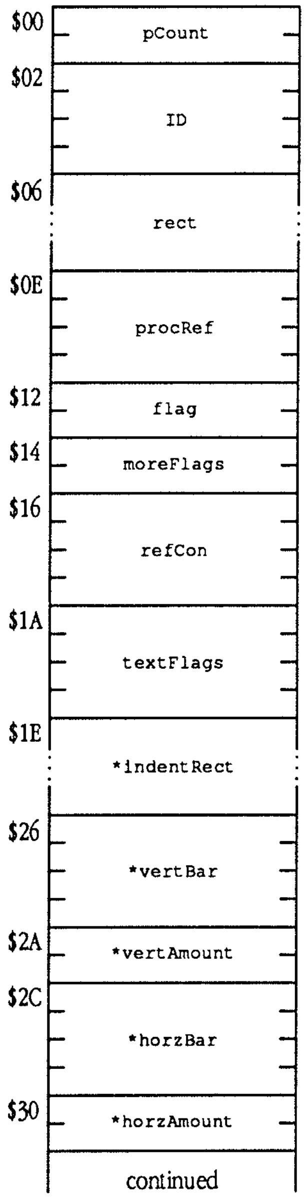
TEParamBlock

This structure contains the parameters necessary to create a new TextEdit record. In it your program defines many of the attributes of the record. The format of this structure directly corresponds to the format of the TextEdit control template accepted by the `NewControl2` Control Manager call when creating TextEdit controls.

The `TENew` tool call requires that its input parameters be specified in a `TEParamBlock` structure. Many of the fields of the `TEParamBlock` directly establish the values for `TERecord` fields.

In the following figure, optional fields are marked with an asterisk (*).

- **Figure 49-3** `TEParamBlock` layout



$00	pCount	Word—Parameter count for template: 7 to 23
$02	ID	Long—Application-assigned control ID
$06	rect	Rectangle—Boundary rectangle for control
$0E	procRef	Long—`editTextControl`=$85000000
$12	flag	Word—Highlight and control flags for control
$14	moreFlags	Word—Additional control flags
$16	refCon	Long—Application-defined value
$1A	textFlags	Long—Specific TextEdit control flags (see below)
$1E	*indentRect	Rectangle—Defines text indentation from control rect (optional)
$26	*vertBar	Long—Handle to vertical scroll bar for control (optional)
$2A	*vertAmount	Word—Vertical scroll amount, in pixels (optional)
$2C	*horzBar	Long—Reserved; must be set to NILL (optional)
$30	*horzAmount	Word—Reserved; must be set to 0 (optional)
	continued	

	continued	
$32	`*styleRef`	Long—Reference to initial style information for text (optional)
$36	`*textDescriptor`	Word—Defines format of initial text and `textRef` (optional)
$38	`*textRef`	Long—Reference to initial text for edit window (optional)
$3C	`*textLength`	Long—Length of initial text (optional)
$40	`*maxChars`	Long—Maximum number of characters allowed (optional)
$44	`*maxLines`	Long—Reserved; must be set to 0 (optional)
$48	`*maxCharsPerLines`	Word—Reserved; must be set to 0 (optional)
$4A	`*maxHeight`	Word—Reserved; must be set to 0 (optional)
$4C	`*colorRef`	Long—Reference to TextEdit color table (optional)
$50	`*drawMode`	Word—QuickDraw II text mode for edit window (optional)
$52	`*filterProcPtr`	Long—Pointer to filter routine for this control (optional)

`pCount` Number of parameter fields to follow. Valid values lie in the range from 7 to 23. The value of this field does not include `pCount` itself.

`ID` Unique ID for TextEdit controls. Your application sets this field, and can use it to uniquely identify a particular TextEdit record.

`boundsRect` Boundary rectangle for the TextEdit record. This rectangle contains the entire record, including its scroll bars and outline. If you set the bottom right corner of this rectangle to (0,0), TextEdit uses the bottom right corner of the window that contains the record as a default. Note that this rectangle must be large enough to completely display a single character in the largest allowed font.

`procRef` Specifies the type of control. Must be set to $85000000.

`flag` Control flags for the TextEdit record. Defined bits for `flag` are as follows:

Reserved	bits 8–15	Must be set to 0
`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 0–6	Must be set to 0

`moreFlags` More control flags for TextEdit record. Defined bits for `moreFlags` are as follows:

`fCtlTarget`	bit 15	Indicates whether this TextEdit record is the current target of user actions. Must be set to 0 when creating a TextEdit record.
`fCtlCanBeTarget`	bit 14	Must be set to 1. TextEdit will set this bit to 0 if the `fDisableSelection` flag in `textFlags` is set to1.
`fCtlWantsEvents`	bit 13	Must be set to 1. TextEdit will set this bit to 0 if the `fDisableSelection` flag in `textFlags` is set to1.
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fTellAboutSize`	bit 11	If set to 1, TextEdit will create a size box in the bottom right corner of the window. Whenever the window is resized, the edit text will be resized and redrawn.
`fCtlIsMultiPart`	bit 10	Must be set to 1
Reserved	bits 4–9	Must be set to 0
Color Table Reference	bits 2–3	Defines type of reference in `colorRef`: 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Style Reference	bits 0–1	Defines type of style reference in `styleRef`: 00 - style reference is pointer 01 - style reference is handle 10 - style reference is resource ID 11 - invalid value

△ **Important** Do not set `fTellAboutSize` to 1 unless the window also has a vertical scroll bar. △

`textFlags` Specific TextEdit control flags. Valid values for `textFlags` are as follows:

`fNotControl`	bit 31	Indicates whether the the TextEdit record to be created is for a control: 1 - TextEdit record is not a control 0 - TextEdit record is a control
`fSingleFormat`	bit 30	Must be set to 1
`fSingleStyle`	bit 29	Allows you to restrict the style options available to the user: 1 - Allow only one style in the text 0 - Do not restrict the number of styles in the text
`fNoWordWrap`	bit 28	Allows you to control TextEdit word wrap behavior: 1 - Do not word wrap the text; only break lines on CR ($0D) characters 0 - Perform word wrap to fit the ruler
`fNoScroll`	bit 27	Controls user access to scrolling: 1 - Do not allow either manual or auto-scrolling 0 - Scrolling permitted
`fReadOnly`	bit 26	Restricts the text in the window to read-only operations (copying from the window will still be allowed): 1 - No editing allowed 0 - Editing permitted
`fSmartCutPaste`	bit 25	Controls TextEdit support for smart cut and paste: 1 - Use smart cut and paste 0 - Do not use smart cut and paste
`fTabSwitch`	bit 24	Defines behavior of the Tab key: 1 - Tab to next control in the window 0 - Tab inserted in TextEdit document
`fDrawBounds`	bit 23	Tells TextEdit whether to draw a box around the edit window, just inside `boundsRect`—the pen for this box is two pixels wide and one pixel high 1 - Draw rectangle 0 - Do not draw rectangle
`fColorHilight`	bit 22	Must be set to 0.
`fGrowRuler`	bit 21	Tells TextEdit whether to resize the ruler in response to the user resizing the edit window. If set to 1, TextEdit will automatically adjust the right margin value for the ruler: 1 - Resize the ruler 0 - Do not resize the ruler

`fDisableSelection`
bit 20 Controls whether user can select text:
1 - User cannot select text
0 - User can select text

`fDrawInactiveSelection`
bit 19 Controls how inactive selected text is displayed:
1 - TextEdit draws a box around inactive selections
0 - TextEdit does nothing special when displaying inactive selections

Reserved bits 0–18 Must be set to 0

`indentRect` Each coordinate of this rectangle specifies the amount of white space to leave between the boundary rectangle for the control and the text itself, in pixels. Default values are (2,6,2,4) in 640 mode and (2,4,2,2) in 320 mode. Each indentation coordinate may be specified individually. In order to assert the default for any coordinate, specify its value as $FFFF.

`vertBar` Handle of the vertical scroll bar to use for the TextEdit window. If you do not want a scroll bar at all, then set this field to NIL. If you want TextEdit to create a scroll bar for you, just inside the right edge of the boundary rectangle for the control, then set this field to $FFFFFFFF.

`vertAmount` Specifies the number of pixels to scroll whenever the user presses the up or down arrow on the vertical scroll bar. In order to use the default value (9 pixels), set this field to $0000.

`horzBar` Must be set to NIL.

`horzAmount` Must be set to 0.

`styleRef` Reference to initial style information for the text, specified in a TEFormat structure. Bits 1 and 0 of `moreFlags` define the type of reference (pointer, handle, resource ID) to the structure. In order to use the default style and ruler information, set this field to NULL.

`textDescriptor`
Input text descriptor that defines the reference type for the initial text (which is in `textRef`) and the format of that text.

`textRef` Reference to initial text for the edit window. If you are not supplying any initial text, then set this field to NULL.

`textLength` If `textRef` is a pointer to the initial text, then this field must contain the length of the initial text. For other reference types, this field is ignored—TextEdit can extract the length from the reference itself.

- ◆ *Note:* You must specify or omit the `textDescriptor`, `textRef`, and `textLength` fields as a group.

`maxChars` Maximum number of characters allowed in the text. If you do not want to define any limit to the number of characters, then set this field to NULL.

`maxLines` Must be set to NULL.

`maxCharsPerLines` Must be set to NULL.

`maxHeight` Must be set to NULL.

`colorRef` Reference to the color table for the text, stored in a `TEColorTable` structure. Bits 2 and 3 of `moreFlags` define the type of reference stored here.

`drawMode` This is the text mode used by QuickDraw II for drawing text. See Chapter 16, "QuickDraw II," in Volume 2 of the *Toolbox Reference* for details on valid text modes.

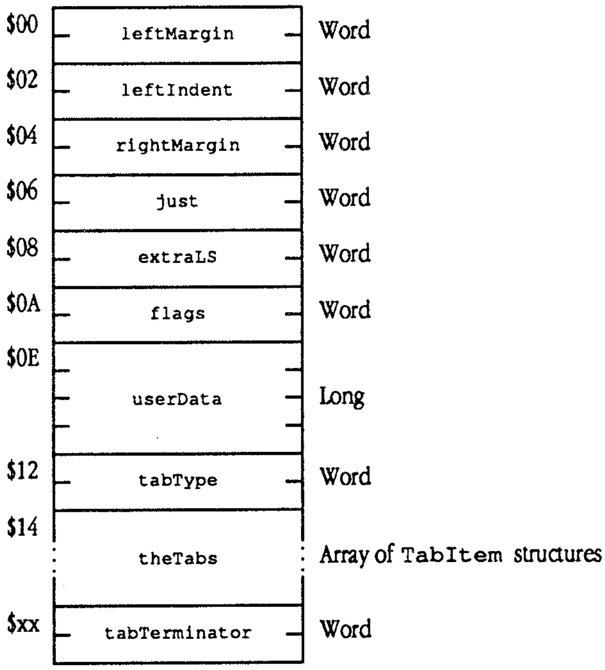
`filterProcPtr` Pointer to a filter routine for the control.For more information about TextEdit filter procedures, see "Generic filter procedure". If you do not want to use a filter routine for the control, set this field to NIL.

`TERuler`

Defines the characteristics of a TextEdit ruler.

The `TEGetRuler` and `TESetRuler` tool calls allow you to obtain and set the ruler information for a TextEdit record.

- **Figure 49-4** `TERuler` layout



$00	leftMargin	Word
$02	leftIndent	Word
$04	rightMargin	Word
$06	just	Word
$08	extraLS	Word
$0A	flags	Word
$0E	userData	Long
$12	tabType	Word
$14	theTabs	Array of `TabItem` structures
$xx	tabTerminator	Word

`leftMargin` Specifies the number of pixels to indent from the left edge of the text rectangle (`viewRect` in `TERecord`) for all text lines except those that start paragraphs.

`leftIndent` Specifies the number of pixels to indent from the left edge of the text rectangle (`viewRect` in `TERecord`) for text lines that start paragraphs.

`rightMargin` Maximum line length, expressed as the number of pixels from the left edge of the text rectangle (`viewRect` in `TERecord`).

`just` Text justification:

0 Left justification—all text lines start flush with left margin
-1 Right justification—all text lines start flush with right margin
1 Center justification—all text lines are centered between left and right margins
2 Full justification—text is blocked flush with both left and right margins; TextEdit pads spaces with extra pixels to fully justify the text

`extraLS` Line spacing, expressed as the number of pixels to add between lines of text. Negative values result in text overlap.

`flags` Control flags:

Reserved	bit 15	Must be set to 0
`fEqualLineSpacing`	bit 14	Must be set to 0
`fShowInvisibles`	bit 13	Must be set to 0
Reserved	bits 0–12	Must be set to 0

`userData` Application-specific data.

`tabType` Specifies the type of tab data that follows:

0 No tabs are set—`tabType` is the last field in the structure
1 Regular tabs—tabs are set at regular pixel intervals, specified by the value of the `tabTerminator` field; `theTabs` is omitted from the structure
2 Absolute tabs—tabs are set at absolute, irregular pixel locations; `theTabs` defines those locations; `tabTerminator` marks the end of `theTabs`

`theTabs` If `tabType` is set to 2, then this is an array of `TabItem` structures defining the absolute pixel positions for the various tab stops. The `tabTerminator` field, with a value of $FFFF, marks the end of this array. For other values of `tabType`, this field is omitted from the structure.

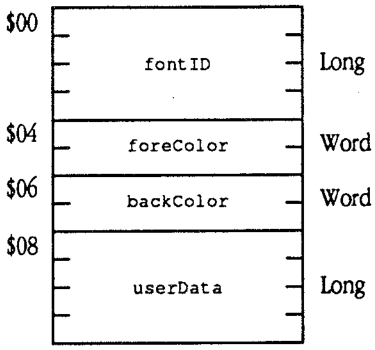
`tabTerminator` If `tabType` is set to 0, then this field is omitted from the structure. If `tabType` is set to 1, then `theTabs` is omitted, and this field contains the number of pixels corresponding to the tab interval for the regular tabs. If `tabType` is set to 2, then `tabTerminator` is set to $FFFF, and marks the end of `theTabs` array.

`TEStyle`

This structure defines the font and color characteristics of a style for text in the TextEdit record.

The `TEFormat` structure contains one or more `TEStyle` structures, each of which defines a unique text style used somewhere in the record text.

■ **Figure 49-5** `TEStyle` layout



`fontID` Font Manager font ID record identifying the font for the text. See Chapter 8, "Font Manager," in the *Toolbox Reference* for more information about font IDs.

`foreColor` Defines the foreground color for the text. This is a TextEdit color word, as described in "`TEColorTable`" in this chapter.

`backColor` Defines the background color for the text. This is a TextEdit color word, as described in "`TEColorTable`" in this chapter.

`userData` Application-specific data.

Low-level TextEdit structures

TextEdit uses several other structures for its internal processing. Typically, your application should not manipulate these structures. In addition, if your program does modify data in these structurers, you should be careful to maintain the correct relationships between fields that affect other TextEdit structures.

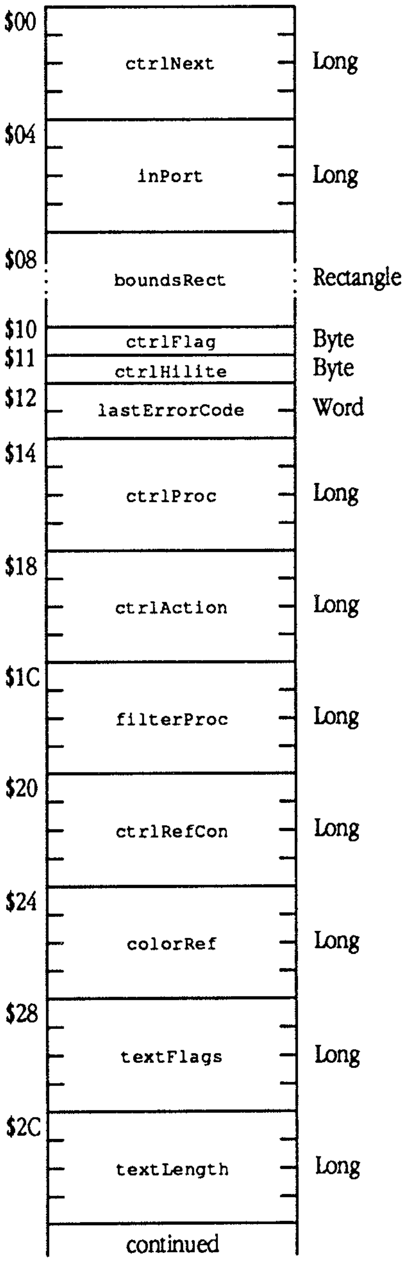
`TERecord`

This is the main structure for a TextEdit record. The `TENew` tool call creates this structure based partially on the information specified in the `TEParamBlock` you supply to that call. In most cases, your program does not need to directly access fields in this structure. However, in order to use such advanced features as custom word-wrap routines, your application will have to modify the `TERecord`.

Note that this section only describes those `TERecord` fields that are currently defined and available to application programs. Your program should assume that there are more fields beyond those described here, and should not try to directly move or copy a `TERecord`.

Most TextEdit tool calls require a handle to a `TERecord` as an entry parameter.

■ **Figure 49-6** TERecord layout



$00	ctrlNext	Long
$04	inPort	Long
$08	boundsRect	Rectangle
$10	ctrlFlag	Byte
$11	ctrlHilite	Byte
$12	lastErrorCode	Word
$14	ctrlProc	Long
$18	ctrlAction	Long
$1C	filterProc	Long
$20	ctrlRefCon	Long
$24	colorRef	Long
$28	textFlags	Long
$2C	textLength	Long
	continued	

	continued	
$30	blockList	TextList
$38	ctrlID	Long
$3C	ctrlMoreFlags	Word
$3E	ctrlVersion	Word
$40	viewRect	Rectangle
$48	totalHeight	Long
$4C	lineSuper	SuperHandle
$58	styleSuper	SuperHandle
$64	styleList	Long
$68	rulerList	Long
$6C	lineAtEndFlag	Word
$6E	selectionStart	Long
$72	selectionEnd	Long
	continued	

	continued	
$76	selectionActive	Word
$78	selectionState	Word
$7A	caretTime	Long
$7E	nullStyleActive	Word
$80	nullStyle	TEStyle
$8C	topTextOffset	Long
$90	topTextVPos	Word
$92	vertScrollBar	Long
$96	vertScrollPos	Long
$9A	vertScrollMax	Long
$9E	vertScrollAmount	Word
$A0	horzScrollBar	Long
$A4	horzScrollPos	Long
	continued	

	continued	
$A8	`horzScrollMax`	Long
$AC	`horzScrollAmount`	Word
$AE	`growBoxHandle`	Long
$B2	`maximumChars`	Long
$B6	`maximumLines`	Long
$BA	`maxCharsPerLine`	Word
$BC	`maximumHeight`	Word
$BE	`textDrawMode`	Word
$C0	`wordBreakHook`	Long
$C4	`wordWrapHook`	Long
$C8	`keyFilter`	Long
$CC	`theFilterRect`	Rectangle
	continued	

	continued	
$D4	`theBufferVPos`	Word
$D6	`theBufferHPos`	Word
$D8	`theKeyRecord`	`KeyRecord`
$E6	`cachedSelcOffset`	Long
$EA	`cachedSelcVPos`	Word
$EC	`cachedSelcHPos`	Word
$EE	`mouseRect`	Rectangle
$F6	`mouseTime`	Long
$FA	`mouseKind`	Word
$FC	`lastClick`	Long
$100	`savedHPos`	Word
$102	`anchorPoint`	Long

`ctrlNext` Handle of next control in the system-maintained control list.

`inPort` Pointer to the GrafPort for this TextEdit record.

`boundsRect` Boundary rectangle for the record, which surrounds the text window as well as its scroll bars and outline.

`ctrlFlag` Control flags for the TextEdit record. TextEdit obtains this field from the low-order byte of the `flags` field in the `TEParamBlock` passed to `TENew`. The following flags are defined:

`ctlInvis`	bit 7	1=invisible, 0=visible
Reserved	bits 0–6	Must be set to 0

`ctrlHilite` Reserved

`lastErrorCode` The last error code generated by TextEdit. Note that this code may have been returned by either a control definition procedure or from a TextEdit tool call.

`ctrlProc` Must be set to $85000000. Identifies this as a TextEdit control to the system.

`ctrlAction` Reserved

`filterProc` Pointer to the generic filter procedure for the record. If there is no filter procedure, this field is set to NULL. See "Generic filter procedure" for information about generic filter procedures.

`ctrlRefCon` Application-defined value.

`colorRef` Reference to the `TEColorTable` for the record. Bits 2 and 3 in `ctrlMoreFlags` define the type of reference stored here.

`textFlags` Control flags specific to TextEdit. The system derives this field from the `textFlags` field in the `TEParamTable` structure passed to `TENew` when a new TextEdit record is created. The following flags are defined:

`fNotControl` bit 31 Indicates whether the the TextEdit record to be created is for a control:
1 - TextEdit record is not a control
0 - TextEdit record is a control

`fSingleFormat` bit 30 Must be set to 1

`fSingleStyle` bit 29 Allows you to restrict the style options available to the user:
1 - Allow only one style in the text
0 - Do not restrict the number of styles in the text

`fNoWordWrap` bit 28 Allows you to control TextEdit word wrap behavior:
1 - Do not word wrap the text; only break lines on CR ($0D) characters
0 - Perform word wrap to fit the ruler

`fNoScroll` bit 27 Controls user access to scrolling:
1 - Do not allow either manual or auto-scrolling
0 - Scrolling permitted

fReadOnly bit 26 Restricts the text in the window to read-only operations (copying from the window will still be allowed):
1 - No editing allowed
0 - Editing permitted

fSmartCutPaste bit 25 Controls TextEdit support for smart cut and paste:
1 - Use smart cut and paste
0 - Do not use smart cut and paste

fTabSwitch bit 24 Defines behavior of the Tab key:
1 - Tab to next control in the window
0 - Tab inserted in TextEdit document

fDrawBounds bit 23 Tells TextEdit whether to draw a box around the edit window, just inside boundsRect. The pen for this rectangle is two pixels wide and one pixel high
1 - Draw rectangle
0 - Do not draw rectangle

fColorHilight bit 22 Must be set to 0.

fGrowRuler bit 21 Tells TextEdit whether to resize the ruler in response to the user resizing the edit window. If set to 1, TextEdit will automatically adjust the right margin value for the ruler:
1 - Resize the ruler
0 - Do not resize the ruler

fDisableSelection
bit 20 Controls whether user can select text:
1 - User cannot select text
0 - User can select text

fDrawInactiveSelection
bit 19 Controls how inactive selected text is displayed:
1 - TextEdit draws a box around inactive selections
0 - TextEdit does not display inactive selections

Reserved bits 0–18 Must be set to 0

textLength Number of bytes of text in the record. Your program must not modify this field.

blockList Cached link into the linked list of TextBlock structures, which contain the actual text for the record. The actual TextList structure resides here. Your program must not modify this field.

ctrlID Application-assigned ID for the TextEdit control.

`ctrlMoreFlags` More control flags. TextEdit obtains the data for this field from the `moreFlags` field of the `TEParamBlock` structure passed to `TENew` when a new TextEdit record is created. The following flags are defined:

`fCtlTarget`	bit 15	Indicates whether this TextEdit record is the current target of user actions. Must be set to 0 when creating a TextEdit record.
`fCtlCanBeTarget`	bit 14	Must be set to 1
`fCtlWantsEvents`	bit 13	Must be set to 1.
`fCtlProcNotPtr`	bit 12	Must be set to 1
`fTellAboutSize`	bit 11	If set to 1, TextEdit will create a size box in the bottom right corner of the window. Whenever the window is resized, the edit text will be resized and redrawn.
`fCtlIsMultiPart`	bit 10	Must be set to 1
Reserved	bits 4–9	Must be set to 0
Color Table Reference	bits 2–3	Defines type of reference in `colorRef`: 00 - color table reference is pointer 01 - color table reference is handle 10 - color table reference is resource ID 11 - invalid value
Reserved	bits 0–1	Must be set to 0

`ctrlVersion` Reserved

`viewRect` Boundary rectangle for the text, within the rectangle defined in `boundsRect`, which surrounds the entire record, including its associated scroll bars and outline.

`totalHeight` Total height of the text in the TextEdit record, in pixels.

`lineSuper` Cached link into the linked list of `SuperBlock` structures that define the text lines in the record.

`styleSuper` Cached link into the linked list of `SuperBlock` structures that define the styles for the record.

`styleList` Handle to array of `TEStyle` structures, containing style information for the record. The array is terminated with a long set to $FFFFFFFF.

`rulerList` Handle to array of `TERuler` structures, defining the format rulers for the record. Note that only the first ruler is currently used by TextEdit. The array is terminated with a long set to $FFFFFFFF.

`lineAtEndFlag` Indicates whether the last character was a line break. If so, this field is set to $FFFF.

`selectionStart`
Starting text offset for the current selection. Must always be less than `selectionEnd`.

`selectionEnd` Ending text offset for the current selection.Must always be greater than `selectionStart`.

`selectionActive`
Indicates whether the current selection (defined by `selectionStart` and `selectionEnd`) is active:

$0000	Active
$FFFF	Inactive

`selectionState` Contains state information about the current selection range:

$0000	Off screen
$FFFF	On screen

`caretTime` Blink interval for caret, expressed in system ticks.

`nullStyleActive`
Indicates whether the null style is active for the current selection:

$0000	Do not use null style when inserting text
$FFFF	Use null style when inserting text

`nullStyle` `TEStyle` structure defining the null style. This may be the default style for newly inserted text, depending upon the value of `nullStyleActive`.

`topTextOffset` Text offset into the record corresponding to the top line displayed on the screen.

`topTextVPos` Difference,in pixels, between the topmost vertical scroll position (corresponding to the top of the vertical scroll bar) and the top line currently displayed on the screen. This is, essentially, the vertical position of the display window in the text for the record.

`vertScrollBar` Handle to the vertical scroll bar control record.

`vertScrollPos` Current position of the vertical scroll bar, in units defined by `vertScrollAmount`.

◆ *Note:* that while TextEdit supports `vertScrollPos` as a long, standard Apple IIGS scroll bars support only the low-order word. This leads to unpredictable scroll bar behavior when editing large documents.

`vertScrollMax` Maximum allowable vertical scroll, in units defined by `vertScrollAmount`.

`vertScrollAmount`
Number of pixels to scroll on each vertical arrow click.

`horzScrollBar` Handle to the horizontal scroll bar control record. Currently not supported

`horzScrollPos` Current position of the horizontal scroll bar, in units defined by `horzScrollAmount`. Currently not supported

`horzScrollMax` Maximum allowable horizontal scroll, in units defined by `horzScrollAmount`. Currently not supported

`horzScrollAmount`
Number of pixels to scroll on each horizontal arrow click. Currently not supported.

`growBoxHandle` Handle of size box control record.

`maximumChars` Maximum number of characters allowed in the text.

`maximumLines` Maximum number of lines allowed in the text. Currently not supported.

`maxCharsPerLine`
Currently not supported.

`maximumHeight` Maximum text height, in pixels. This value allows applications to easily constrain text to a display window of a known height. Currently not supported.

`textDrawMode` QuickDraw II drawing mode for the text. See Chapter 16, "QuickDraw II," in the *Toolbox Reference* for more information on QuickDraw II drawing modes.

`wordBreakHook` Pointer to the routine that handles word breaks. See "Word break hook" earlier in this chapter for information about word break routines. Your program may modify this field.

`wordWrapHook` Pointer to the routine that handles word wrap. See "Word wrap hook" earlier in this chapter for information about word wrap routines. Your program may modify this field.

`keyFilter` Pointer to the keystroke filter routine. See "Keystroke filter procedure" earlier in this chapter for information about keystroke filter routines. Your program may modify this field.

`theFilterRect` Defines a rectangle used by the generic filter procedure for some of its routines. See "Generic filter procedure" earlier in this chapter for information about generic filter procedures and their routines. Your program may modify this field.

`theBufferVPos` Vertical component of the current position of the buffer within the port for the TextEdit record, expressed in the local coordinates appropriate for that port. This value is used by some generic filter procedure routines. See "Generic filter procedure" for information about generic filter procedures and their routines. Your program may modify this field.

`theBufferHPos` Horizontal component of the current position of the buffer within the port for the TextEdit record, expressed in the local coordinates appropriate for that port. This value is used by some generic filter procedure routines. See "Generic filter procedure" earlier in this chapter for information about generic filter procedures and their routines. Your program may modify this field.

`theKeyRecord` Parameter block, in `KeyRecord` format, for the keystroke filter routine. Your program may modify this field.

`cachedSelcOffset`

Cached selection text offset. If this field is set to $FFFFFFFF, then the cache is invalid and will be recalculated when appropriate.

`cachedSelcVPos`

Vertical component of the cached buffer position, expressed in local coordinates for the output port.

`cachedSelcHPos`

Horizontal component of the cached buffer position, expressed in local coordinates for the output port.

`mouseRect` Boundary rectangle for multiclick mouse commands. If the user clicks the mouse more than once within the region defined by this rectangle within the time period defined for multiclicks, then TextEdit interprets those clicks as multiclick sequences (double- or triple-clicks). The user sets the time period with the Control Panel.

`mouseTime` System tickcount when the user last released the mouse button.

`mouseKind` Type of last mouse click:

0	Single click
1	Double-click
2	Triple-click

`lastClick` Location of last user mouse click.

`savedHPos` Cached horizontal character position. TextEdit uses this value to manage where it should display the caret on a line when the user presses the up or down arrow.

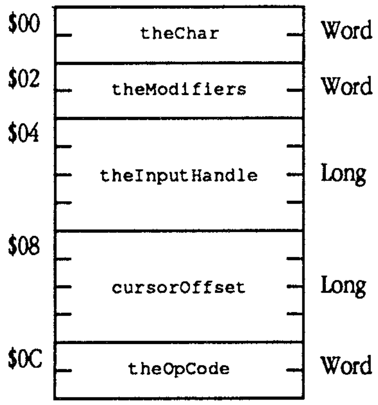
`anchorPoint` Defines the character from which the user began to select text for the current selection. When TextEdit expands the current selection (as a result of user keyboard or mouse commands, or at the direction of a custom keystroke filter procedure), it always does so from the `anchorPoint`, not `selectionStart` or `selectionEnd`.

`KeyRecord`

Defines the entry and exit parameter block for the keystroke filter procedure for a TextEdit record. On entry to the filter procedure, TextEdit sets up this structure with the information necessary to process the keystroke. On exit, the filter procedure returns the processed keystroke and any other status information in this same structure. For complete information about the TextEdit keystroke filter procedure and the use of these fields, see "Keystroke filter procedure" earlier in this chapter.

The `KeyRecord` for a TextEdit record resides in the appropriate `TERecord`.

■ **Figure 49-7** `KeyRecord` layout



`theChar` Character code of the character to translate. The low-order byte of `theChar`_contains the key code for the character; the high-order byte is ignored.

`theModifiers` On input, contains the state of the modifier keys when the character in `theChar` was generated. This is an Event Manager modifier word, as described in Chapter 7, "Event Manager," of the *Toolbox Reference.* On output, the keystroke filter procedure may change the setting of these flags.

`theInputHandle`
On input, contains a handle to a copy of the keystroke in `theChar`. On output, the keystroke filter procedure may modify the size and content of the data referred to by this handle.

`cursorOffset` For some operations, the keystroke filter routine sets this field with a new cursor text offset.

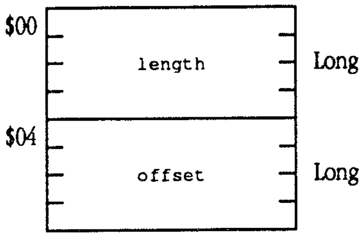
`theOpCode` On return from the filter routine, this field contains an operation code indicating what TextEdit is to do next and how it is to interpret the `KeyRecord`:

`opNormal`	$0000	TextEdit performs its standard processing on the character stored in the location referred to by `theInputHandle`
`opNothing`	$0001	TextEdit ignores the keystroke
`opReplaceText`	$0002	TextEdit inserts the text referred to by `theInputHandle` in place of the current selection in the record; if there is no current selection, TextEdit inserts the text at the current insertion point—if `theInputHandle` has 0 size, TextEdit deletes the current selection and inserts nothing
`opMoveCursor`	$0003	TextEdit moves the cursor to the location specified by `cursorOffset`
`opExtendCursor`	$0004	TextEdit extends the current selection from its anchor point to the location specified by `cursorOffset`
`opCut`	$0005	TextEdit copies the current selection to the desk scrap then clears the selection
`opCopy`	$0006	TextEdit copies the current selection to the desk scrap
`opPaste`	$0007	TextEdit inserts the contents of the desk scrap in place of the current selection
`opClear`	$0008	TextEdit clears the current selection

StyleItem

The TEFormat structure contains an array of StyleItem substructures, which define the text characters that use a particular style. Each element of this array refers to the style information for a run of characters. Taken consecutively, the array of StyleItem structures completely defines the styles for the entire record.

- **Figure 49-8** StyleItem layout



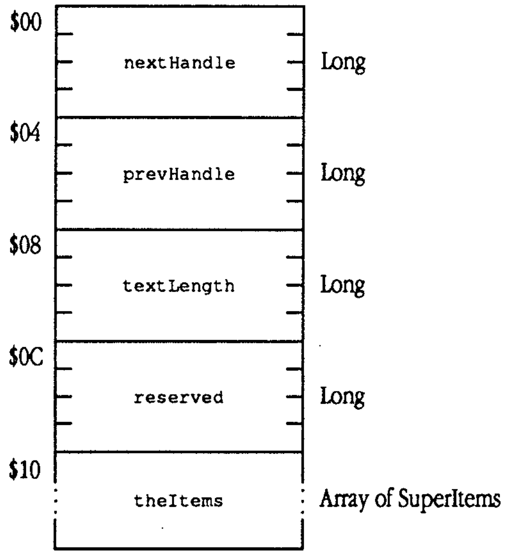
length — The total number of text characters that use this style. These characters begin where the previous StyleItem left off. Value of $FFFFFFFF indicates an unused entry.

offset — Offset, in bytes, into theStyleList array in the TEFormat record to the TEStyle record which defines the characteristics of the style in question.

SuperBlock

SuperBlock structures define linked-lists of TextEdit control information items. These control information items may relate to styles or to line start locations, and are defined by the SuperItem substructure. A SuperHandle substructure provides address information into a chain of SuperBlock structures. The TERecord contains a number of SuperHandles.

■ **Figure 49-9** SuperBlock layout



nextHandle Handle to the next SuperBlock in this chain of blocks. A value of NIL indicates the end of the chain.

prevHandle Handle to the previous SuperBlock in this chain of blocks. A value of NIL indicates the beginning of the chain.

textLength The number of characters of text referred to by theItems.

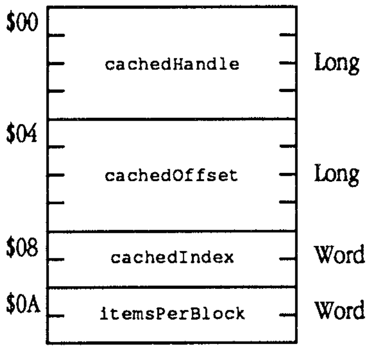
Reserved Reserved

theItems Array of SuperItems for this SuperBlock. The textLength field contains the total length of the characters defined by these items.

SuperHandle

Identifies the current position within a chain of SuperBlocks. This substructure contains both byte offset and index information. The cachedOffset field contains the offset into the text identified by the cached SuperItem. The cachedIndex field contains the byte offset to the SuperItem within its SuperBlock. The TERecord contains several SuperHandles.

■ **Figure 49-10** SuperHandle layout



cachedHandle Handle to the SuperBlock containing the current SuperItem.

cachedOffset Byte offset to the current character within the text identified by the cached SuperItem.

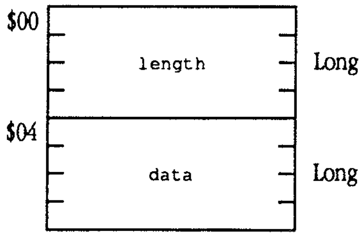
cachedIndex Byte offset to the start of the current SuperItem within the array of SuperItems stored in the SuperBlock referred to by cachedHandle.

itemsPerBlock Indicates the number of SuperItems stored in each SuperBlock.

SuperItem

Defines an individual item within a `SuperBlock`.

- **Figure 49-11** `SuperItem` layout



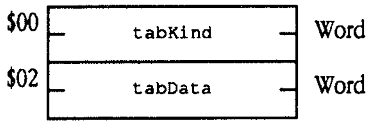
`length` Specifies the number of text characters in the TextEdit record that are affected by this `SuperItem`. A value of $FFFFFFFF indicates that this item is not currently used.

`data` Contains the actual data for the item.

TabItem

Contains information defining an absolute tab position, expressed as a pixel offset from the left margin of the text rectangle (viewRect of the TERecord). The TERuler structure contains an array of TabItems whenever the user has defined absolute tabs.

■ **Figure 49-12** TabItem layout



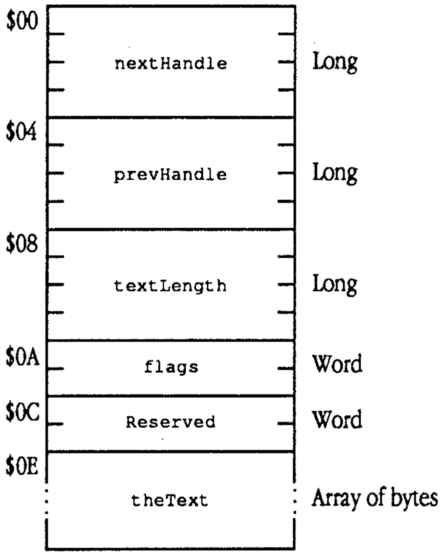
tabKind — Must be set to $0000.

tabData — Location of the absolute tab, expressed as the number of pixels to indent from the left edge of the text rectangle (viewRect of TERecord).

TextBlock

Contains the actual text for the record. The TextBlock substructure defines a linked-list which stores the text. A TextList substructure within the TERecord contains access information into the chain of TextBlocks for the TextEdit record. The TextBlock chain stores the text for the TextEdit record in sequential order. That is, the first TextBlock contains the first block of text, the second TextBlock contains the next block of text, and so on.

■ **Figure 49-13** TextBlock layout



nextHandle Handle to the next TextBlock in the chain of blocks for this text record. A value of NIL indicates the end of the chain.

prevHandle Handle to the previous TextBlock in the chain of blocks for this text record. A value of NIL indicates the beginning of the chain.

textLength The number of text bytes stored at theText.

flags Reserved.

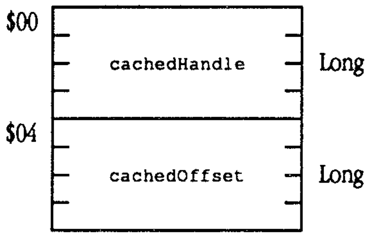
Reserved Reserved

theText Text for the record. The textLength field specifies the length of this array.

`TextList`

Identifies the current position within the text for the record, which is stored in `TextBlocks`. The `TERecord` contains a `TextList` substructure.

- **Figure 49-14** `TextList` layout



`cachedHandle` Handle to the `TextBlock` containing the text corresponding to the current location.

`cachedOffset` Byte offset within the `TextBlock` referred to by `cachedHandle` corresponding to the current location.

TextEdit housekeeping routines

The following sections describe the standard housekeeping calls in the TextEdit Tool Set.

`TEBootInit` $0122

Initializes TextEdit; called only by the Tool Locator.

Parameters None

Errors None

`TEStartUp` $0222

Starts up the TextEdit tool set, and prepares TextEdit for use by an application by allocating memory and formatting direct-page space. Application programs must issue this call before any other TextEdit tool calls. Before exiting, applications that issue the `TEStartUp` call must call `TEShutDown` to de-initialized TextEdit.

Parameters

Stack before call

Previous contents	
userID	Word—Application's User ID (obtained at program start time)
directPage	Word—Address of one page of direct-page memory
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2201	`teAlreadyStarted`	TextEdit has already been started
Memory Manager errors		Returned unchanged

C

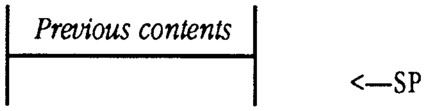
```
extern pascal void TEStartUp(userID, directPage);

Word      userID, directPage;
```

`TEShutDown` $0322

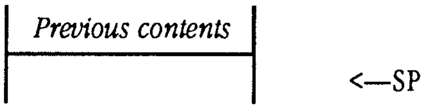
Frees memory used by TextEdit on behalf of an application, not including individual TextEdit records—it is the application's responsibility to issue the `TEKill` tool call at the end of each TextEdit record. Every application that uses TextEdit must issue this call before exiting. During application initialization, applications that use TextEdit must issue the `TEStartUp` tool call before any other TextEdit calls.

Parameters

Stack before call



Stack after call



Errors	$2202	`teNotStarted`	TextEdit has not been started
C	`extern pascal void TEShutDown();`		

TEVersion $0422

Retrieves the Resource Manager version number. This call returns valid information if TextEdit has been loaded; the tool set need not be active. The *versionInfo* result will contain the information in the standard format defined in Appendix A, "Writing Your Own Tool Set," in Volume 2 of the *Toolbox Reference.*

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
	<—SP

Stack after call

Previous contents	
versionInfo	Word—TextEdit version number
	<—SP

Errors None

C

```
extern pascal Word TEVersion();
```

TEReset $0522

Resets TextEdit; called only when the system is reset.

Parameters None

Errors None

`TEStatus` $0622

Returns a flag indicating whether TextEdit is active. If TextEdit has not been loaded, your program receives a Tool Locator error (`toolNotFoundErr`).

- ◆ *Note:* If your program issues this call in assembly language, initialize the result space on the stack to NIL. Upon return from `TEStatus`, your program need only check the value of the returned flag. If TextEdit is not active, the returned value will be FALSE (NIL).

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
	<—SP

Stack after call

Previous contents	
activeFlag	Word—Boolean; TRUE if TextEdit is active
	<—SP

Errors None

C `extern pascal Word TEStatus();`

TextEdit tool calls

The following sections describe the TextEdit tool calls, in order by call name.

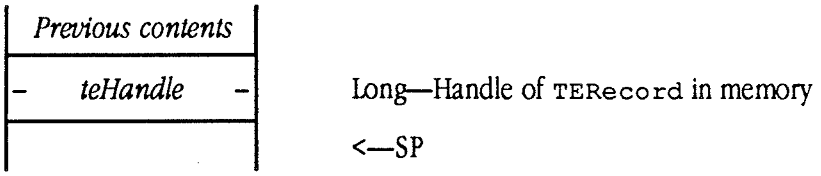
`TEActivate` $0F22

Makes the specified TextEdit record active—that is, makes that record the target of user keystrokes. TextEdit highlights the current selection or displays the caret, as appropriate. User editing activity now applies to this TextEdit record.

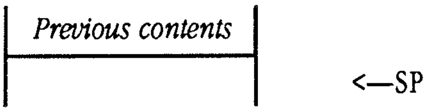
Your application need issue this call only if it is managing its own TextEdit records. If your program uses TextEdit controls with TaskMaster, it should not issue this call; TaskMaster manages the control automatically.

Parameters

Stack before call



Stack after call



Errors	$2202	`teNotStarted`	TextEdit has not been started
	$2203	`teInvalidHandle`	*teHandle* does not refer to a valid `TERecord`

C

```
extern pascal void TEActivate(teHandle);

Long      teHandle;
```

`TEClear` $1922

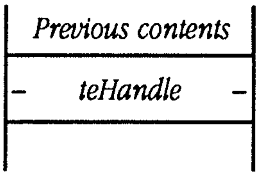
Clears the current selection in the active TextEdit record and redraws the screen. If there is no current selection, then this call does nothing and returns immediately. This call does not affect the desk scrap.

Note that this call does not generate any update events; it directly redraws the active record.

Your application need issue this call only if it is managing its own TextEdit records. If your program uses TextEdit controls and TaskMaster, it should not issue this call; TaskMaster manages the control automatically.

Parameters

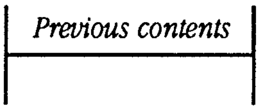
Stack before call



Long—Handle of `TERecord` in memory; NULL for active record

<—SP

Stack after call



<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started

C

```
extern pascal void TEClear(teHandle);

Long        teHandle;
```

teHandle Specifies the TextEdit record for the operation. If your program specifies a NULL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

`TEClick` $1122

Tracks the mouse within a TextEdit record, selecting all text that it passes over until the user releases the mouse button. If the user holds down the Shift key, this call extends the current selection to include the new text. TextEdit automatically scrolls the text in the proper direction if the user drags the mouse outside the view rectangle.

This call handles double- and triple-clicks as follows: double-clicks select a word, dragging lengthens or shortens the selection in word increments; triple-clicks select a line, dragging lengthens or shortens the selection in line increments.

If your program issues this call for a TextEdit record that is not currently active, TextEdit first makes that record active, then proceeds to track the mouse.

Your application need issue this call only if it is managing its own TextEdit records. If your program uses TextEdit controls with TaskMaster, it should not issue this call; TaskMaster manages the control automatically.

Parameters

Stack before call

Previous contents	
– *eventRecordPtr* –	Long—Pointer to event record for the mouse click
– *teHandle* –	Long—Handle of `TERecord` in memory
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started
	$2203	`teInvalidHandle`	*teHandle* does not refer to a valid `TERecord`
	Memory Manager errors		Returned unchanged

C

```
extern pascal void TEClick(eventRecordPtr,
          teHandle);

Pointer   eventRecordPtr;
Long      teHandle;
```

eventRecordPtr Points to the event record describing the mouse click. The `what`, `when`, `where`, and `modifiers` fields of the event record must be set. TextEdit ignores the `message` field. For information on the format and content of event records, see Chapter 7, "Event Manager," in the *Toolbox Reference.*

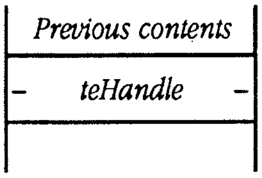
`TECompactRecord` $1728

Compacts all the TextEdit data structures in a specified TextEdit record. TECompactRecord reclaims space used for deleted lines and style items, and for styles that are no longer referenced from the text. While this call may be issued by any application that uses TextEdit, it is intended to be used from within an out-of-memory routine (see Chapter 36, "Memory Manager Update," in this book for information about out-of-memory routines and the out-of-memory queue).

Note that your program may not pass a NULL TextEdit record handle to this tool call.

Parameters

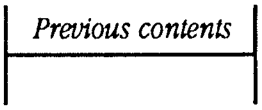
Stack before call



Long—Handle of `TERecord` to compact

<—SP

Stack after call



<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started
	Memory Manager errors		Returned unchanged

C

```
extern pascal void TECompactRecord(teHandle);

Long      teHandle;
```

teHandle Specifies the TextEdit record for the operation.

`TECopy` $1722

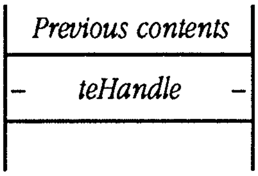
Copies the current selection from the active TextEdit record to the desk scrap, destroying the previous desk scrap contents. This call copies both the text and style information to the scrap. Note, however, that if there is no current selection, then this call does nothing, and does not affect the desk scrap.

This call does not automatically scroll to the current selection.

Your application need issue this call only if it is managing its own TextEdit records. If your program uses TextEdit controls, it should not issue this call; TaskMaster manages the control automatically.

Parameters

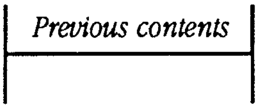
Stack before call



Long—Handle of `TERecord` in memory; NULL for active record

<—SP

Stack after call



<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started
	Memory Manager errors		Returned unchanged

C

```
extern pascal void TECopy(teHandle);

Long       teHandle;
```

teHandle Specifies the TextEdit record for the operation. If your program specifies a NULL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

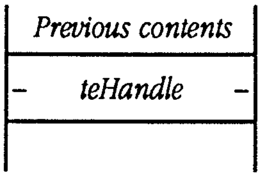
`TECut` $1622

Copies the current selection from the active TextEdit record to the desk scrap, destroying the previous desk scrap contents. `TECut` then scrolls to the beginning of the selection, deletes it, and then redraws the screen. This call copies both the text and style information to the scrap. Note, however, that if there is no current selection, then this call does nothing, and does not affect the desk scrap.

Your application need issue this call only if it is managing its own TextEdit records. If your program uses TextEdit controls, it should not issue this call; TaskMaster manages the control automatically.

Parameters

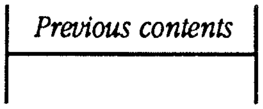
Stack before call



Long—Handle of `TERecord` in memory; NULL for active record

<—SP

Stack after call



<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started
	Memory Manager errors		Returned unchanged

C

```
extern pascal void TECut(teHandle);

Long        teHandle;
```

teHandle Specifies the TextEdit record for the operation. If your program specifies a NULL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

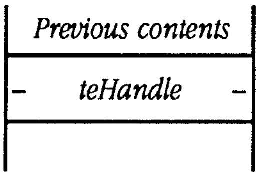
TEDeactivate $1022

Deactivates a TextEdit record. Your program specifies the `TERecord` for the record in question. `TEDeactivate` changes the highlighting for the current selection in that record to show that it is inactive. Any user editing actions (keystrokes, cut and paste) have no effect on the deactivated record.

Your application need issue this call only if it is managing its own TextEdit records. If your program uses TextEdit controls, it should not issue this call; TaskMaster manages the control automatically.

Parameters

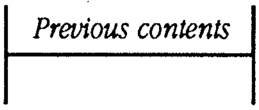
Stack before call



Long—Handle of `TERecord` in memory

<—SP

Stack after call



<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started
	$2203	`teInvalidHandle`	*teHandle* does not refer to a valid TERecord

C

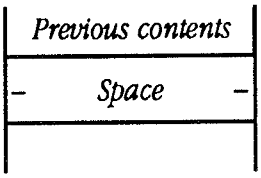
```
extern pascal void TEDeactivate(teHandle);

Long      teHandle;
```

`TEGetDefProc` $2222

Returns the address of the TextEdit control definition procedure. The system issues this call when the Control Manager starts up in order to obtain the address of the TextEdit control definition procedure. This call is not intended for application use.

Parameters

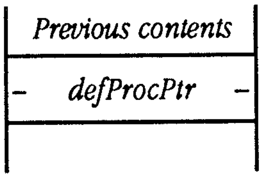
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to control definition procedure

<—SP

Errors None

C `extern pascal Pointer TEGetDefProc();`

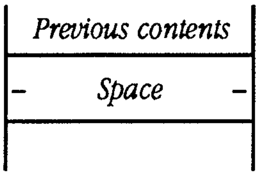
`TEGetInternalProc` $2622

Returns a pointer to the low-level procedure routine for TextEdit.

This call is reserved for future use by application programs to access certain low-level TextEdit routines.

Parameters

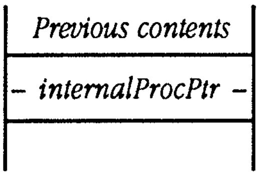
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to internal low-level procedure routine

<—SP

Errors None

C `extern pascal Pointer TEGetInternalProc();`

`TEGetLastError` $2722

Returns the last error code generated for a TextEdit record. Your program specifies the `TERecord` for the appropriate record and a flag indicating whether to clear the last error code after the call. TextEdit then returns the last error code for that record, and, if requested, clears the last error field.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
clearFlag	Word—Flag controlling disposition of last error field for record
– *teHandle* –	Long—Handle of `TERecord` in memory; NULL for active record
	<—SP

Stack after call

Previous contents	
lastError	Word—Last error code generated for the record
	<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started

C

```
extern pascal Word TEGetLastError(clearFlag,
          teHandle);

Word      clearFlag;
Long      teHandle;
```

clearFlag Controls what TextEdit does with the last error field after servicing the call:

$0000	Leaves the last error code intact
$FFFF	Clears the last error code to $0000

TEGetRuler $2322

Returns the ruler definition for a TextEdit record. Your program specifies the destination for the ruler information and the `TERecord` corresponding to the appropriate record. `TEGetRuler` returns the `TERuler` record defining the ruler for the record in question.

Parameters

Stack before call

Previous contents	
rulerDescriptor	Word—Defines type of reference in *rulerRef*
– *rulerRef* –	Long—Reference to buffer to receive `TERuler` record
– *teHandle* –	Long—Handle of `TERecord` in memory; NULL for active record
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started

C

```
extern pascal void TEGetRuler(rulerDescriptor,
                    rulerRef, teHandle);

Word        rulerDescriptor;
Long        rulerRef, teHandle;
```

rulerDescriptor Defines the type of reference stored in *rulerRef*:

`refIsPointer`	$0000	*rulerRef* contains a pointer to a buffer to receive the `TERuler` structure
`refIsHandle`	$0001	*rulerRef* contains a handle to a buffer to receive the `TERuler` structure
`refIsResource`	$0002	*rulerRef* contains a resource ID which can be used to access a buffer to receive the `TERuler` structure
`refIsNewHandle`	$0003	*rulerRef* contains a pointer to a four-byte buffer to receive a handle to the `TERuler` structure; `TEGetRuler` allocates the new handle and returns it in the specified buffer

teHandle Specifies the TextEdit record for the operation. If your program specifies a NULL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

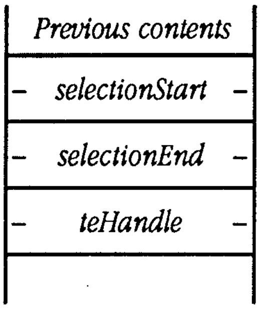
`TEGetSelection` $1C22

Returns information defining the current selection for a TextEdit record. Your program specifies the `TERecord` for the record in question. `TEGetSelection` then determines the starting and ending byte offsets for the current selection, and returns those values into locations specified by your program.

Both offset values are stored as four-byte long values. If there is no current selection for the specified record, both the starting and ending offsets will contain the current caret position.

Parameters

Stack before call



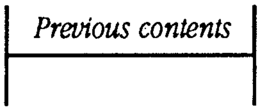
Long—Pointer to buffer to receive starting offset value

Long—Pointer to buffer to receive ending offset value

Long—Handle of `TERecord` in memory; NULL for active record

<—SP

Stack after call



<—SP

Errors $2202 `teNotStarted` TextEdit has not been started

C

```
extern pascal void TEGetSelection(selectionStart,
                    selectionEnd, teHandle);

Pointer   selectionStart, selectionEnd;
Long      teHandle;
```

teHandle Specifies the TextEdit record for the operation. If your program specifies a NULL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

TEGetSelectionStyle $1E22

Returns all style information for the text in the current selection in a TextEdit record. Your program specifies the `TERecord` for the record in question and the addresses of buffers to receive the style data. `TEGetSelectionStyle` then loads the main output buffer with `TEStyle` structures describing all styles affecting text in the current selection. The first word in the buffer contains a counter indicating the number of `TEStyle` structures returned.

`TEGetSelectionStyle` also builds a common style record, which contains all style elements which are common to all text in the selection. A flag word directs your program to the relevant portions of the common style record, which is also in `TEStyle` format.

If there is no current selection, `TEGetSelectionStyle` returns the null style record, which defines the style that will be applied to any text inserted at the current caret position.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
– *commonStylePt* –	Long—Pointer to `TEStyle` buffer for common style record
– *styleHandle* –	Long—Handle to buffer for style information
– *teHandle* –	Long—Handle of `TERecord` in memory; NULL for active record
	<—SP

Stack after call

Previous contents	
commonFlag	Word—Bit flag describing common style record contents
	<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started

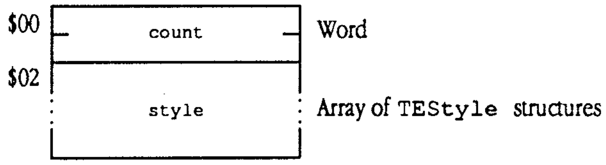
C

```
extern pascal Word
          TEGetSelectionStyle(commonStylePtr,
          styleHandle, teHandle);

Pointer   commonStylePtr;
Long      styleHandle, teHandle;
```

commonStylePtr Points to a buffer to receive a formatted `TEStyle` structure containing the style elements that are common to all text in the current selection. *commonFlag* indicates which portions of this `TEStyle` structure contain valid data.

styleHandle Handle to a buffer to receive the style information for the current selection. `TEGetSelectionStyle` returns as many `TEStyle` structures as are required to fully specify all the styles in the selection. If the buffer referenced by *styleHandle* cannot store enough `TEStyle` structures, `TEGetSelectionStyle` automatically resizes the handle memory.

On return from `TEGetSelectionStyle`, the buffer referenced by *styleHandle* will be formatted as follows:



`count` Indicates the number of `TEStyle` structures in the `styles` array.

`styles` Array of `count` `TEStyle` structures.

teHandle Specifies the TextEdit record for the operation. If your program specifies a NULL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

commonFlag Indicates which portions of the common style record pointed to by *commonStylePtr* are relevant:

Reserved	bits 6–15	Will be set to 0
`fUseFont`	bit 5	The font family defined by the `fontID` field of the common style record is valid: 1 - Font family valid 0 - Font family not valid
`fUseSize`	bit 4	The font size defined by the `fontID` field of the common style record is valid: 1 - Font size valid 0 - Font size not valid
`fUseForeColor`	bit 3	The `foreColor` field of the common style record is valid: 1 - `foreColor` valid 0 - `foreColor` not valid
`fUseBackColor`	bit 2	The `backColor` field of the common style record is valid: 1 - `backColor` valid 0 - `backColor` not valid
`fUseUserData`	bit 1	The `userData` field of the common style record is valid: 1 - `userData` valid 0 - `userData` not valid
`fUseAttributes`	bit 0	The attributes defined by the `fontID` field of the common style record is valid: 1 - Font attributes valid 0 - Font attributes not valid

TEGetText $0C22

Returns the text from a TextEdit record, including the style information associated with that text. Your program specifies the TERecord for the record in question, the format for the returned text, and buffers to receive the text and style data. TEGetText places the text in the return buffer in the format requested by your program; style information is returned in a TEFormat structure.

In addition, TEGetText returns a value indicating the total length of the text in the TextEdit record. This value represents the number of bytes of text in the record, not the number of bytes loaded into the return buffer. If the return buffer is too small to receive all the record text, TEGetText returns a teBufferOverflow error. This error is also returned if the text is too large to be returned in the specified format (for example, the record contains 300 text characters and your program requested an output Pascal string).

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
bufferDescriptor	Word—Defines the format for text returned at *bufferRef*
– *bufferRef* –	Long—Reference to the output text buffer
– *bufferLength* –	Long—Length of the buffer referred to by *bufferRef*
styleDescriptor	Word—Defines the type of reference stored in *styleRef*
– *styleRef* –	Long—Reference to buffer for TEFormat structure defining style
– *teHandle* –	Long—Handle of TERecord in memory; NULL for active record
	<—SP

Stack after call

Previous contents	
– *textLength* –	Long—Total length of all text in record
	<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started
	$2203	`teInvalidHandle`	*teHandle* does not refer to a valid `TERecord`
	$2204	`teInvalidDescriptor`	Invalid descriptor value specified
	$2208	`teBufferOverflow`	The output buffer was too small to accept all data
	Memory Manager errors		Returned unchanged

C

```
extern pascal Long TEGetText(bufferDescriptor,
bufferRef, bufferLength, styleDescriptor, styleRef,
teHandle);

Long      bufferRef, bufferLength, styleRef,
          teHandle;
Word      bufferDescriptor, styleDescriptor;
```

bufferDescriptor Defines the format in which `TEGetText` should return the record text, and the type of reference stored in *bufferRef*:

Reserved	bits 5–16	Must be set to 0
`refFormat`	bits 3–4	Defines the type of reference stored in *bufferRef*: 00 - *bufferRef* is a pointer to the output buffer; *bufferLength* contains the length of the buffer (in bytes) 01 - *bufferRef* is a handle to the output buffer; *bufferLength* is ignored 10 - *bufferRef* is a resource ID for the output buffer (TextEdit will create the resource if it does not already exist); *bufferLength* is ignored 11 - *bufferRef* is a pointer to a four-byte buffer to receive a handle to the output text; `TEGetText` allocates the handle; *bufferLength* is ignored

`dataFormat`	bits 0–2	Defines the format for the output text: 000 - Output data will be formatted as a Pascal string (resource type of `rPString`) 001 - Output data will be formatted as a C string (resource type of `rCString`) 010 - Output data will be formatted as a Class 1 GS/OS input string (resource type of `rC1InputString`) 011 - Output data will be formatted as a Class 1 GS/OS output string (resource type of `rC1OutputString`); application need not set the buffer size field 100 - Output data will be formatted for input to LineEdit `TextBox2` tool call (resource type of `rTextBox2`)—see Chapter 10, "LineEdit Tool Set," in the *Toolbox Reference* for details 101 - Output data will be returned as an unformatted text block (resource type of `rText`) 110 - Invalid 111 - Invalid

bufferLength Defines the length of the output buffer referenced by *bufferRef*, if `refFormat` indicates that *bufferRef* contains a pointer. For other types of references, this field is ignored.

styleDescriptor Defines the type of reference stored in *styleRef*:

`refIsPointer`	$0000	*styleRef* contains a pointer to a buffer to receive the `TEFormat` structure
`refIsHandle`	$0001	*styleRef* contains a handle to a buffer to receive the `TEFormat` structure
`refIsResource`	$0002	*styleRef* contains a resource ID which can be used to access a buffer to receive the `TEFormat` structure
`refIsNewHandle`	$0003	*styleRef* contains a pointer to a four-byte buffer to receive a handle to the `TEFormat` structure; `TEGetText` allocates the new handle and returns it in the specified buffer

styleRef Reference to buffer to receive style information, in `TEFormat` structure form. If this field is set to NULL, then `TEGetText` will not return any style information, and will ignore *styleDescriptor.*

teHandle Specifies the TextEdit record for the operation. If your program specifies a NULL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

textLength Indicates the number of bytes of text in the record. Note that this value may exceed the number of bytes returned at *bufferRef*, if the referenced buffer is too small to receive all the text. In this case, `TEGetText` also returns a `teBufferOverflow` error code.

TEGetTextInfo $0D22

Returns a variable-sized information record describing a TextEdit record. Your program specifies the `TERecord` for the TextEdit record in question, the address of a buffer to receive the information record, and a value indicating how much data `TEGetTextInfo` should return. The system returns the appropriate data at the specified location.

Parameters

Stack before call

Previous contents	
– *infoRecPtr* –	Long—Pointer to buffer for information record
parameterCount	Word—Specifies the number of fields to return
– *teHandle* –	Long—Handle of `TERecord` in memory; NULL for active record
	<—SP

Stack after call

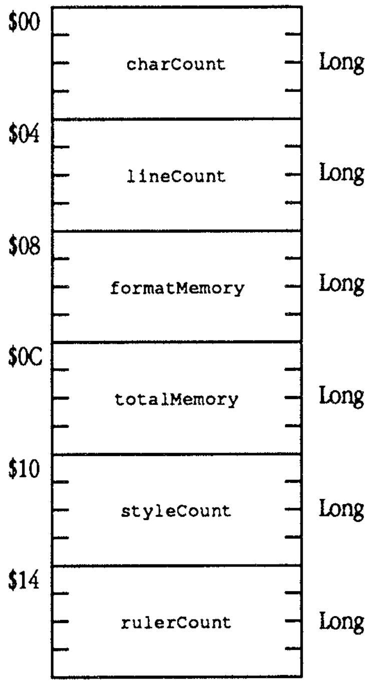
Previous contents	
	<—SP

Errors			
	$2202	`teNotStarted`	TextEdit has not been started
	$2203	`teInvalidHandle`	*teHandle* does not refer to a valid `TERecord`
	$2206	`teInvalidPCount`	Invalid parameter count value specified

C

```
extern pascal void TEGetTextInfo(infoRecPtr,
                    parameterCount, teHandle);

Pointer   infoRecPtr;
Long      teHandle;
Word      parameterCount;
```

infoRecPtr Points to a buffer to receive a partial or complete information record, depending upon the value of *parameterCount.* The information record is formatted as follows (future versions of TextEdit may add fields to the end of this record):



`charCount` Contains the number of text characters in the record.

`lineCount` Contains the number of lines in the record. A line is defined as all text that is displayed on a single line of the screen, based upon the current display options.

`formatMemory` The amount of memory (in bytes) required to store the style information for the record.

`totalMemory` The amount of memory (in bytes) required for the record, including both text and style data.

`styleCount` The number of unique styles defined for the record.

`rulerCount` The number of rulers defined for the record.

parameterCount Specifies the number of information record fields to be returned by `TEGetTextInfo`. Valid values lie in the range from 1 to 6. Values outside this range yield a `teInvalidPCount` error code. The returned data always begin with the `charCount` field, and continue until the specified number of fields have been formatted.

teHandle Specifies the TextEdit record for the operation. If your program specifies a NULL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

`TEIdle` $0E22

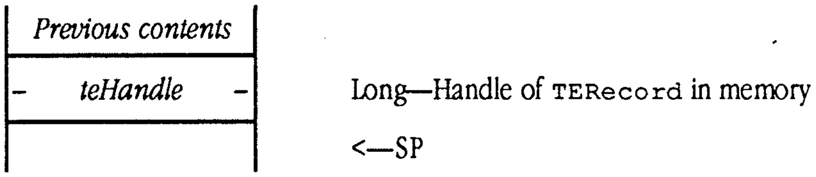
Provides processor time to TextEdit so that it can blink the cursor and perform background tasks. Your program specifies the `TERecord` for the record. TextEdit then determines whether enough time has elapsed to require a cursor blink, and, if so, blinks the cursor. In addition, TextEdit performs any necessary background processing for the record.

Your application need issue this call only if it is managing its own TextEdit records. If your program uses TextEdit controls, it should not issue this call; TaskMaster manages the control automatically.

Your program should call `TEIdle` often—usually every time through the main event loop, and periodically during time-consuming operations. If your program does not call `TEIdle` often enough, the cursor will blink irregularly. TextEdit ensures that the cursor blink rate does not exceed that specified by the user's control panel setting.

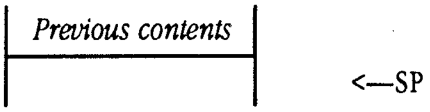
Parameters

Stack before call



Long—Handle of `TERecord` in memory

Stack after call



Errors $2202 `teNotStarted` TextEdit has not been started

C

```
extern pascal void TEIdle(teHandle);

Long      teHandle;
```

`TEInsert` $1A22

Inserts a block of text before the current selection in a TextEdit record, and then redraws the text screen. Your program specifies the text and style data to be inserted and the `TERecord` for the record. `TEInsert` then inserts the text and style data at the current selection.

This call does not affect the desk scrap.

Parameters

Stack before call

Previous contents	
textDescriptor	Word—Defines the format for text stored at *textRef*
– *textRef* –	Long—Reference to the input text buffer
– *textLength* –	Long—Length of the buffer referred to by *textRef*
styleDescriptor	Word—Defines the type of reference stored in *styleRef*
– *styleRef* –	Long—Reference to `TEFormat` structure defining style for text
– *teHandle* –	Long—Handle of `TERecord` in memory; NULL for active record
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started
	$220C	`teInvalidTextBox2`	The `TextBox2` format codes were inconsistent
	Memory Manager errors		Returned unchanged

C

```
extern pascal void TEInsert(textDescriptor, textRef,
            textLength, styleDescriptor, styleRef,
            teHandle);

Long        textRef, textLength, styleRef, teHandle;
Word        textDescriptor, styleDescriptor;
```

textDescriptor Defines the format of the text to be inserted, and the type of reference stored in *textRef*:

Reserved	bits 5–16	Must be set to 0
`refFormat`	bits 3–4	Defines the type of reference stored in *textRef*: 00 - *textRef* is a pointer to the text buffer; *textLength* contains the length of the buffer (in bytes) 01 - *textRef* is a handle to the text buffer; *textLength* is ignored 10 - *textRef* is a resource ID for the text buffer; *textLength* is ignored 11 - Invalid
`dataFormat`	bits 0–2	Defines the format of the text: 000 - Text is a Pascal string (resource type of `rPString`) 001 - Text is a C string (resource type of `rCString`) 010 - Text is a Class 1 GS/OS input string (resource type of `rC1InputString`) 011 - Text is a Class 1 GS/OS output string (resource type of `rC1OutputString`) 100 - Text is formatted for input to LineEdit `TextBox2` tool call (resource type of `rTextBox2`)—see Chapter 10, "LineEdit Tool Set," in the *Toolbox Reference* for details; style data in the text overrides that specified by *styleRef* 101 - Text is an unformatted text block (resource type of `rText`) 110 - Invalid 111 - Invalid

textLength Defines the length of the buffer referenced by *textRef*. This field is only valid for reference types that do not contain length data (see *textDescriptor*). For other types of references, this field is ignored.

styleDescriptor Defines the type of reference stored in *styleRef*:

`refIsPointer`	$0000	*styleRef* contains a pointer to a TEFormat structure
`refIsHandle`	$0001	*styleRef* contains a handle to a TEFormat structure
`refIsResource`	$0002	*styleRef* contains a resource ID which can be used to access a buffer containing the TEFormat structure

styleRef Reference to buffer containing style information, in `TEFormat` structure form. If this field is set to NULL, then `TEInsert` will use the style of the first character in the current selection for the record, and will ignore *styleDescriptor.*

teHandle Specifies the TextEdit record for the operation. If your program specifies a NULL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

TEKey $1422

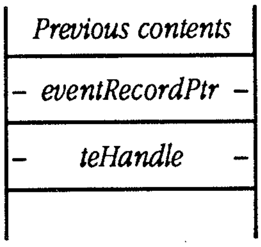
Processes a keystroke for a TextEdit record. Your program specifies the TERecord for the record and the event record for the keystroke. TEKey then processes the key. If the keystroke is a control key (one that requires special processing, as outlined in "Standard TextEdit key sequences" earlier in this chapter), TEKey performs the appropriate TextEdit action. If the keystroke is not a control key, TEKey inserts the corresponding character into the text of the target TextEdit record at the current selection.

Your application need issue this call only if it is managing its own TextEdit records. If your program uses TextEdit controls, it should not issue this call; TaskMaster manages the control automatically.

Your program should issue this call in response to KeyDown or AutoKey events.

Parameters

Stack before call

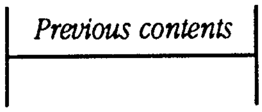


Long—Pointer to event record for the key

Long—Handle of TERecord in memory

<—SP

Stack after call



<—SP

Errors	$2202	teNotStarted	TextEdit has not been started
	Memory Manager errors		Returned unchanged

C

```
extern pascal void TEKey(eventRecordPtr, teHandle);

Pointer    eventRecordPtr;
Long       teHandle;
```

eventRecordPtr Points to the event record describing the keystroke. For information on the format and content of event records, see Chapter 7, "Event Manager," in the *Toolbox Reference.*

`TEKill` $0A22

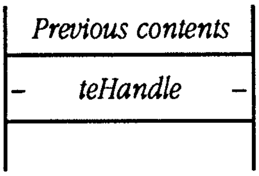
Deallocates a `TERecord` and all associated memory. Your program specifies the `TERecord` to be freed. `TEKill` then releases the record and any memory supporting it. `TEKill` does not erase or invalidate the screen, nor does it make another record the target if the target record is killed. Your program must take care of these duties.

Your program should issue this call only when it is completely through with the `TERecord` and its TextEdit record—all text associated with the record is lost after this call.

If your program uses TextEdit controls it may issue the `KillControls` or `DisposeControl` Control Manager tool calls instead of `TEKill`.

Parameters

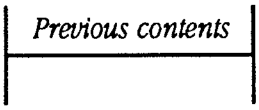
Stack before call



Long—Handle of `TERecord` in memory

<—SP

Stack after call



<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started
	$2203	`teInvalidHandle`	*teHandle* does not refer to a valid TERecord

C

```
extern pascal void TEKill(teHandle);

Long        teHandle;
```

`TENew` $0922

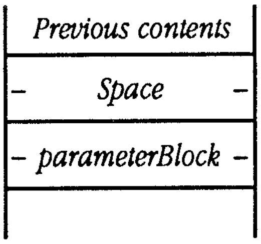
Allocates a new TextEdit record in the current port, and returns the `TERecord` defining that record. Your program specifies the parameters for that record in a `TEParamBlock` structure (see "TextEdit data structures" earlier in this chapter for information on the format and content of the `TEParamBlock`). TextEdit then allocates and formats the `TERecord` for the record.

The bounds rectangle specified in the `TEParamBlock` must be large enough to completely enclose a single character in the largest allowable font for the record.

Your program should issue this call only if it is not using TextEdit controls. In order to create a TextEdit control, use the `NewControl2` Control Manager tool call (see Chapter 28, "Control Manager Update," in this book). Note that `NewControl2` may be used to create several controls at once.

Parameters

Stack before call

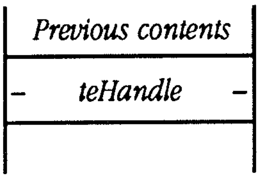


Long—Space for result

Long—Pointer to formatted `TEParamBlock`

<—SP

Stack after call



Long—Handle to new `TERecord`

<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started
	$2204	`teInvalidDescriptor`	Invalid descriptor value specified
	$2205	`teInvalidFlag`	*textFlags* field of `TEParamBlock` is invalid
	$2206	`teInvalidPCount`	Invalid parameter count value specified
	Memory Manager errors		Returned unchanged

C

```
extern pascal Long TENew(parameterBlock);

Pointer   parameterBlock;
```

`TEOffsetToPoint` $2022

Converts a text byte offset into a pixel position expressed in the local coordinates of the GrafPort containing the TextEdit record. Your program specifies the byte offset to the character in question, the addresses of buffers to receive the pixel position information, and the `TERecord` for the record. `TEOffsetToPoint` then determines the pixel position for the character.

The returned pixel position is expressed as two signed long integers. If the specified offset is beyond the end of the text for the record, `TEOffsetToPoint` returns the position of the last character. Note that if the specified character lies above the display rectangle, the vertical position component will be a negative value. The pixel position is not expressed as a QuickDraw II point, because the TextEdit drawing space is larger than the QuickDraw II drawing space.

The `TEPointToOffset` call performs the inverse operation, converting a pixel position into a text offset.

Parameters

Stack before call

Previous contents	
– *textOffset* –	Long—Byte offset to text
– *vertPosPtr* –	Long—Pointer to four-byte buffer to receive vertical position
– *horzPosPtr* –	Long—Pointer to four-byte buffer to receive horizontal position
– *teHandle* –	Long—Handle of `TERecord` in memory; NULL for active record
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started

C

```
extern pascal void TEOffsetToPoint(textOffset,
          vertPosPtr, horzPosPtr, teHandle);

Long      textOffset, teHandle;
Pointer   vertPosPtr, horzPosPtr;
```

teHandle Specifies the TextEdit record for the operation. If your program specifies a NULL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

`TEPaintText` $1322

Prints the text from a TextEdit record. Your program specifies the destination rectangle and GrafPort, print control information, and the TextEdit record from which `TEPaintText` is to print. TextEdit then draws the appropriate record text into the specified rectangle and GrafPort. `TEPaintText` begins printing at a line number you specify, and continues until the destination rectangle has been filled. The routine then returns the next line number to be printed so that your program can issue the next call correctly.

Your program issues this tool call within a Print Manager job, which you start by calling `PrOpenDoc`. The Print Manager returns the GrafPort pointer when you initiate the job. Refer to Chapter 15, "Print Manager," of the *Toolbox Reference* for complete information on starting, managing, and ending a print job.

Note that this call is not limited to printing; your application can use this tool call to paint into any GrafPort.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
– *grafPort* –	Long—Pointer to destination GrafPort
– *startingLine* –	Long—Starting line number for print operation
– *rectPtr* –	Long—pointer to the destination rectangle in GrafPort
flags	Word—Control flags for the print operation
– *teHandle* –	Long—Handle of `TERecord` in memory; NULL for active record
	<—SP

Stack after call

Previous contents	
– *nextLine* –	Long—Next line number to print
	<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started
	$2209	`teInvalidLine`	Starting line value is greater than the number of lines in the text (end-of-file)

C

```
extern pascal Long TEPaintText(grafPort,
          startingLine, rectPtr, flags, teHandle);

Pointer   grafPort, rectPtr;
Long      startingLine, teHandle;
Word      flags;
```

startingLine Specifies the first line to be printed. A line is defined as all text that is displayed on a single line of the screen, based upon the current display options.

grafPort Points to a QuickDraw II GrafPort definition which describes the destination for the print operation. For more information on the format, content, and use of the GrafPort structure, see Chapter 16, "QuickDraw II," in the *Toolbox Reference.*

rectPtr Points to a structure defining the destination rectangle for the print operation. This rectangle essentially defines the output page size, and must lie in the output GrafPort specified by *grafPort.* Each print operation initiated by `TEPaintText` ends when the rectangle described by the structure pointed to by *rectPtr* fills. Refer to the description of the `PrOpenPage` tool call in Chapter 15, "Print Manager," of the *Toolbox Reference* for more information on this frame rectangle.

flags Controls the print operation:

`fPartialLines`	bit 15	Reserved; must be set to 0
`fDontDraw`	bit 14	Controls printing: 0 - Print data 1 - Calculate the number of lines that will fit in *rectPtr*, but do not print—*nextLine* still contains next line to print just as if text had been printed (supports page skip)
Reserved	bits 0–13	Must be set to 0

teHandle Specifies the TextEdit record for the operation. If your program specifies a NULL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

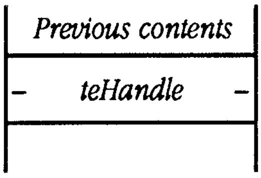
`TEPaste` $1822

Replaces the current selection with the contents of the desk scrap, including both text and style information. Your program specifies the `TERecord` for the TextEdit record in which the operation is to take place. `TEPaste` then pastes the data from the desk scrap into the record text. If the desk scrap is empty, the current selection is untouched.

Your application need issue this call only if it is managing its own TextEdit records. If your program uses TextEdit controls, it should not issue this call; TaskMaster manages the control automatically.

Parameters

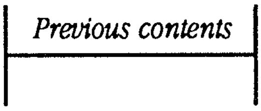
Stack before call



Long—Handle of `TERecord` in memory; NULL for active record

<—SP

Stack after call



<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started
	Memory Manager errors		Returned unchanged

C

```
extern pascal void TEPaste(teHandle);

Long        teHandle;
```

teHandle Specifies the TextEdit record for the operation. If your program specifies a NULL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

`TEPointToOffset` $2122

Converts a pixel position expressed in the local coordinates of the GrafPort containing the TextEdit record into a text byte offset into the text for the record. Your program specifies the pixel position in terms of its relative horizontal and vertical location in the GrafPort, but not as a QuickDraw II point. `TEPointToOffset` then generates the appropriate text offset within the record.

The vertical and horizontal components of the pixel position are represented as signed `Long` integers. If the specified position lies before the first text character in the record, then the returned offset will be $00000000. If the position is after the last text character, the call returns the offset of the last character in the record. If your program specifies a horizontal position beyond the last character in a line, `TEPointToOffset` returns the offset of the last character in the line.

The `TEOffsetToPoint` call performs the inverse operation, converting a text offset into a pixel position.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
– *vertPos* –	Long—Vertical position component
– *horzPos* –	Long—Horizontal position component
– *teHandle* –	Long—Handle of `TERecord` in memory; NULL for active record
	<—SP

Stack after call

Previous contents	
– *textOffset* –	Long—Byte offset to text corresponding to pixel position
	<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started

C

```
extern pascal Long TEPointToOffset(vertPos, horzPos,
          teHandle);

Long      vertPos, horzPos, teHandle;
```

teHandle Specifies the TextEdit record for the operation. If your program specifies a NULL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

`TEReplace` $1B22

Replaces the current selection in a TextEdit record with a specified block of text, and then redraws `the` text screen. Your program specifies the text and style data to be inserted and the TERecord for the record. `TEReplace` then inserts the text and style data in place of the current selection.

This call does not affect the desk scrap.

Parameters

Stack before call

Stack	
Previous contents	
textDescriptor	Word—Defines the format for text stored at *textRef*
– *textRef* –	Long—Reference to the input text buffer
– *textLength* –	Long—Length of the buffer referred to by *textRef*
styleDescriptor	Word—Defines the type of reference stored in *styleRef*
– *styleRef* –	Long—Reference to `TEFormat` structure defining style for text
– *teHandle* –	Long—Handle of `TERecord` in memory; NULL for active record
	<—SP

Stack after call

Stack	
Previous contents	
	<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started
	$220C	`teInvalidTextBox2`	The `TextBox2` format codes were inconsistent
	Memory Manager errors		Returned unchanged

C

```
extern pascal void TEReplace(textDescriptor,
        textRef, textLength, styleDescriptor,
        styleRef, teHandle);

Long    textRef, textLength, styleRef, teHandle;
Word    textDescriptor, styleDescriptor;
```

textDescriptor Defines the format of the text to be inserted, and the type of reference stored in *textRef*.

Reserved	bits 5–16	Must be set to 0
`refFormat`	bits 3–4	Defines the type of reference stored in *textRef*: 00 - *textRef* is a pointer to the text buffer; *textLength* contains the length of the buffer (in bytes) 01 - *textRef* is a handle to the text buffer; *textLength* is ignored 10 - *textRef* is a resource ID for the text buffer; *textLength* is ignored 11 - Invalid
`dataFormat`	bits 0–2	Defines the format of the text: 000 - Text is a Pascal string (resource type of `rPString`) 001 - Text is a C string (resource type of `rCString`) 010 - Text is a Class 1 GS/OS input string (resource type of `rC1InputString`) 011 - Text is a Class 1 GS/OS output string (resource type of `rC1OutputString`) 100 - Text is formatted for input to LineEdit TextBox2 tool call (resource type of `rTextBox2`)—see Chapter 10, "LineEdit Tool Set," in the *Toolbox Reference* for details; style data in the text overrides that specified by *styleRef* 101 - Text is an unformatted text block (resource type of `rText`) 110 - Invalid 111 - Invalid

textLength Defines the length of the buffer referenced by *textRef*. This field is only valid for reference types that do not contain length data (see *textDescriptor*). For other types of references, this field is ignored.

styleDescriptor Defines the type of reference stored in *styleRef*:

`refIsPointer`	$0000	*styleRef* contains a pointer to a `TEFormat` structure
`refIsHandle`	$0001	*styleRef* contains a handle to a `TEFormat` structure
`refIsResource`	$0002	*styleRef* contains a resource ID which can be used to access a buffer containing the `TEFormat` structure

styleRef Reference to buffer containing style information, in `TEFormat` structure form. If this field is set to NULL, then `TEInsert` will use the first defined style in the current selection for the record, and will ignore *styleDescriptor.*

teHandle Specifies the TextEdit record for the operation. If your program specifies a NULL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

`TEScroll` $2522

Scrolls the text in a TextEdit record. Your program specifies control information for the scroll and the `TERecord` for the record. `TEScroll` then updates the current position for the record accordingly.

Parameters

Stack before call

Previous contents	
scrollDescriptor	Word—Control information for the scroll operation
– *vertAmount* –	Long—Vertical amount to scroll (this is a signed value)
– *horzAmount* –	Long—Horizontal amount to scroll (Must be set to 0)
– *teHandle* –	Long—Handle of `TERecord` in memory; NULL for active record
	<—SP

Stack after call

Previous contents	
	<—SP

Errors $2202 `teNotStarted` TextEdit has not been started

C

```
extern pascal void TEScroll(vertAmount, horzAmount,
                teHandle);

Long        vertAmount, horzAmount, teHandle;
```

scrollDescriptor Defines the nature of the scroll operation, and the use of and units for *vertAmount* and *horzAmount*:

$0000 Scroll to absolute text position, place at top of window. The *vertAmount* parameter contains the offset value for the text character to place at the top of the TextEdit display window. The *horzAmount* parameter is ignored.

$0001 Scroll to absolute text position, center in window. The *vertAmount* parameter contains the offset value for the text character to place in the center of the TextEdit display window. The *horzAmount* parameter is ignored.

$0002 Scroll to line, place at top of window. The *vertAmount* parameter contains a line number specifying which text line to place at the top of the TextEdit display window. The *horzAmount* parameter is ignored.

$0003 Scroll to line, center in window. The *vertAmount* parameter contains a line number specifying which text line to center in the TextEdit display window. The *horzAmount* parameter is ignored.

$0004 Scroll to absolute unit position, place at top of window. The *vertAmount* parameter contains a value to which to scroll the top of the TextEdit window, in units defined by the value of the *vertScrollAmount* field of the `TERecord` for the record. The *horzAmount* parameter must be set to 0.

$0005 Scroll to relative unit position, place at top of window. The *vertAmount* parameter contains a value to add to contents of the *vertScrollPos* field of the `TERecord` for the record, in units defined by the value of the *vertScrollAmount* field of that `TERecord`.The *horzAmount* parameter must be set to 0.

teHandle Specifies the TextEdit record for the operation. If your program specifies a NULL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

`TESetRuler` $2422

Sets the ruler for a TextEdit record. Your program specifies the new ruler definition, in `TERuler` format, and the `TERecord` for the record. `TESetRuler` then sets the ruler as specified, and reformats all text in the record. For TextEdit controls, `TESetRuler` invalidates the entire display rectangle (the screen will be redrawn on the next update event). For TextEdit records that are not controls, `TESetRuler` redraws the screen.

Parameters

Stack before call

Previous contents	
rulerDescriptor	Word—Defines type of reference in *rulerRef*
– *rulerRef* –	Long—Reference to buffer containing new `TERuler` structure
– *teHandle* –	Long—Handle of `TERecord` in memory; NULL for active record
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started

C

```
extern pascal void TESetRuler(rulerDescriptor,
                    rulerRef, teHandle);

Word        rulerDescriptor;
Long        rulerRef, teHandle;
```

rulerDescriptor Defines the type of reference stored in *rulerRef*:

`refIsPointer`	$0000	*rulerRef* contains a pointer to a buffer containing the `TERuler` structure
`refIsHandle`	$0001	*rulerRef* contains a handle to a buffer containing the `TERuler` structure
`refIsResource`	$0002	*rulerRef* contains a resource ID which can be used to access a buffer containing the `TERuler` structure

teHandle Specifies the TextEdit record for the operation. If your program specifies a NULL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

TESetSelection $1D22

Sets the current selection for a TextEdit record. Your program specifies the starting and ending text byte offsets for the selection and the TERecord for the record. TESetSelection then updates the record accordingly.

If the ending offset value is less than the starting value, TESetSelection automatically swaps them. If either offset is beyond the end of the text for the record, it is reset to the offset for the last character.

Parameters

Stack before call

Previous contents	
– *selectionStart* –	Long—Starting offset value
– *selectionEnd* –	Long—Ending offset value
– *teHandle* –	Long—Handle of TERecord in memory; NULL for active record
	<—SP

Stack after call

Previous contents	
	<—SP

Errors			
	$2202	teNotStarted	TextEdit has not been started
	$2203	teInvalidHandle	*teHandle* does not refer to a valid TERecord

C

```
extern pascal void TESetSelection(selectionStart,
                    selectionEnd, teHandle);

Pointer    selectionStart, selectionEnd;
Long       teHandle;
```

teHandle Specifies the TextEdit record for the operation. If your program specifies a NULL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

`TESetText` $0B22

Replaces the text in a TextEdit record, including style information, with supplied text and style data. Your program supplies the text and style information, along with the `TERecord` for the TextEdit record. `TESetText` then replaces any existing text and style information in the record with the supplied data. For TextEdit controls, `TESetText` then invalidates the entire display rectangle (the screen will be redrawn on the next update event). For TextEdit records that are not controls, `TESetText` redraws the screen immediately.

Supplied style information must be formatted in a `TEFormat` structure.

Parameters

Stack before call

Previous contents	
textDescriptor	Word—Defines the format for text stored at *textRef*
– *textRef* –	Long—Reference to the input text
– *textLength* –	Long—Length of the buffer referred to by *textRef*
styleDescriptor	Word—Defines the type of reference stored in *styleRef*
– *styleRef* –	Long—Reference to TEFormat structure defining style
– *teHandle* –	Long—Handle of `TERecord` in memory; NULL for active record
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started
	$2203	`teInvalidHandle`	*teHandle* does not refer to a valid `TERecord`
	$2204	`teInvalidDescriptor`	Invalid descriptor value specified
	Memory Manager errors		Returned unchanged

C

```
extern pascal void TESetText(textDescriptor,
          textRef, textLength, styleDescriptor,
          styleRef, teHandle);

Long      textRef, textLength, styleRef, teHandle;
Word      textDescriptor, styleDescriptor;
```

textDescriptor Defines the format of the new text for the record, and the type of reference stored in *textRef*:

Reserved bits 5–16 Must be set to 0

refFormat bits 3–4 Defines the type of reference stored in *textRef*:
00 - *textRef* is a pointer to the text; *textLength* contains the length of the buffer (in bytes)
01 - *textRef* is a handle to the text; *textLength* is ignored
10 - *textRef* is a resource ID for the text; *textLength* is ignored
11 - Invalid value

dataFormat bits 0–2 Defines the format of the text:
000 - Text is a Pascal string (resource type of `rPString`)
001 - Text is a C string (resource type of `rCString`)
010 - Text is a Class 1 GS/OS input string (resource type of `rC1InputString`)
011 - Text is a Class 1 GS/OS output string (resource type of `rC1OutputString`)
100 - Text is formatted for input to LineEdit `TextBox2` tool call (resource type of `rTextBox2`)—see Chapter 10, "LineEdit Tool Set," in the *Toolbox Reference* for details; style data in the text overrides that specified by *styleRef*
101 - Text is an unformatted text block (resource type of `rText`)
110 - Invalid
111 - Invalid

textLength Defines the length of the output buffer referenced by *textRef*. This field is only valid for reference types that do not contain length data (see *textDescriptor*). For other types of references, this field is ignored.

styleDescriptor Defines the type of reference stored in *styleRef*:

`refIsPointer`	$0000	*styleRef* contains a pointer to the `TEFormat` structure
`refIsHandle`	$0001	*styleRef* contains a handle to the `TEFormat` structure
`refIsResource`	$0002	*styleRef* contains a resource ID which can be used to access the `TEFormat` structure

styleRef Reference to style information for the new text, in `TEFormat` structure form. If this field is set to NULL, then `TESetText` uses the first style encountered in the existing text for the record.

teHandle Specifies the TextEdit record for the operation. If your program specifies a NULL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

`TEStyleChange` $1F22

Changes the style information for the current selection in a TextEdit `record`. Your program specifies the style information and the `TERecord` for the record. TEStyleChange then applies that new information to all the styles in the current selection. If there is no current selection, then the new style applies to the null style record, which defines style information for newly inserted text.

Parameters

Stack before call

Previous contents	
flags	Word—Control flag for applying style data frcm `TEStyle`
– *stylePtr* –	Long—Pointer to `TEStyle` structure
– *teHandle* –	Long—Handle of `TERecord` in memory; NULL for active record
	<—SP

Stack after call

Previous contents	
	<—SP

Errors	$2202	`teNotStarted`	TextEdit has not been started
	$2205	`teInvalidFlag`	Specified flag word is invalid

C

```
extern pascal void TEStyleChange(flags, stylePtr,
                    teHandle);

Word        flags;
Pointer     stylePtr;
Long        teHandle;
```

flags Indicates which portions of the `TEStyle` structure pointed to by *stylePtr* are relevant:

Reserved	bits7–15	Must be set to 0
`fReplaceFont`	bit 6	Controls use of font family defined by `fontID` field of `TEStyle` structure: 1 - Replace the font family for all styles in the current selection 0 - Do not change font family
`fReplaceSize`	bit 5	Controls use of font size defined by `fontID` field of `TEStyle` structure 1 - Replace the font size for all styles in the current selection 0 - Do not change font size
`fReplaceForeColor`	bit 4	Controls use of `foreColor` field of `TEStyle` structure: 1 - Replace the foreground color for all styles in the current selection 0 - Do not change the foreground color
`fReplaceBackColor`	bit 3	Controls use of `backColor` field of `TEStyle` structure: 1 - Replace the background color for all styles in the current selection 0 - Do not change the background color
`fReplaceUserData`	bit 2	Controls the use of the `userData` field of the `TEStyle` structure: 1 - Replace the `userData` field for all styles in the current selection with that in the supplied `TEStyle` structure 0 - Do not change `userData` field
`fReplaceAttributes`	bit 1	Controls use of font attributes defined by the `fontID` field of `TEStyle` structure: 1 - Replace the font attributes for all styles in the current selection 0 - Do not change font attributes

`fSwitchAttributes`		
	bit 0	Controls attribute switching: 1 - If the entire selection contains the font attributes specified by the `TEStyle` structure `fontID` field, then these attributes are removed from the selection; otherwise, the specified attributes are added to those already defined for the selection (note that the attributes are considered together, not individually) 0 - Perform no attribute switching

◆ *Note:* The `fReplaceAttributes` and `fSwitchAttributes` flags are mutually exclusive. If both flags are set to 1, `TEStyleChange` returns a `teInvalidFlag` error code.

stylePtr Points to a formatted `TEStyle` structure containing the style elements that are to be applied to the current selection. The *flags* parameter indicates which portions of this `TEStyle` structure contain valid data.

teHandle Specifies the TextEdit record for the operation. If your program specifies a NULL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

`TEUpdate` $1222

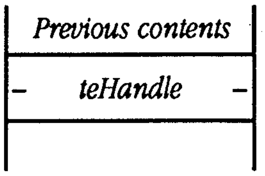
Redraws the screen for a TextEdit record. Your program specifies the `TERecord` for the record. `TEUpdate` then redraws the text for the record. Only that portion of the screen that must be redrawn is affected.

Your program should issue this call after the Window Manager `BeginUpdate` call, and before an `EndUpdate` call. Issue this call separately for each TextEdit record in the window. TextEdit returns very quickly if no redraw is required.

If your program uses TextEdit controls, use the ControlManager `DrawControls` tool call, rather than this one.

Parameters

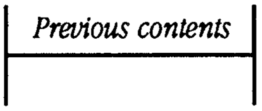
Stack before call



Long—Handle of `TERecord` in memory

<—SP

Stack after call



<—SP

Errors			
	$2202	`teNotStarted`	TextEdit has not been started
	$2203	`teInvalidHandle`	*teHandle* does not refer to a valid TERecord

C

```
extern pascal void TEUpdate(teHandle);

Long        teHandle;
```

TextEdit summary

This section summarizes the constants, data structures, and error codes used by the Resource Manager.

- **Table 49-1** TextEdit constants

Name	Value	Description
Justification values		
`leftJust`	$0000	Left justify all text
`rightJust`	$FFFF	Right justify all text
`centerJust`	$0001	Center all text
`fullJust`	$0002	Fully justify all text (both left and right margins)
`TERuler flags` field values		
`fShowInvisibles`	$2000	Show invisible characters (space, tab, return)
`fEqualLineSpacing`	$4000	TextEdit uses maximum line spacing for every line in the block
`TERuler tabType` field values		
`noTabs`	$0000	No tabs defined—`tabType` is last field in `TERuler` structure
`stdTabs`	$0001	Tabs every `tabTerminator` pixels—`tabTerminator` is last field in the `TERuler` structure; `theTabs` is omitted
`absTabs`	$0002	Tabs at absolute locations specified by `theTabs` array
`TEParamBlock flags` field values		
`fCtlInvis`	$0080	Controls visibility of the record
`fRecordDirty`	$0040	Indicates whether text or style data have changed since the last save

`TEParamBlock moreFlags` field values

`fCtlTarget`	$8000	Record is target of user keystrokes
`fCtlCanBeTarget`	$4000	Record can be target of user keystrokes—must be set to 1
`fCtlWantsEvents`	$2000	Must be set to 1
`fCtlProcRefNotPtr`	$1000	Must be set to 1
`fTellAboutGrow`	$0800	Record wants to know when the user changes the window size
`fCtlIsMultiPart`	$0400	Must be set to 1
`colorDescriptor`	$000C	Indicates type of reference in `colorRef`
`styleDescriptor`	$0003	Indicates type of reference in `styleRef`

`TEParamBlock textFlags` field values

`fNotControl`	$80000000	TextEdit Record is not a control
`fSingleFormat`	$40000000	Only one ruler is allowed for record—must be set to 1
`fSingleStyle`	$20000000	Only one style is allowed for record
`fNoWordWrap`	$10000000	No word wrap is performed
`fNoScroll`	$08000000	The text cannot scroll
`fReadOnly`	$04000000	Text cannot be edited
`fSmartCutPaste`	$02000000	Record supports intelligent cut and paste
`fTabSwitch`	$01000000	Tab key switches user to next TextEdit record on the screen
`fDrawBounds`	$00800000	TextEdit draws a box around text, inside the bounding rectangle
`fColorHilight`	$00400000	Use color table for highlighting
`fGrowRuler`	$00200000	Adjust right margin whenever the user changes the window size
`fDisableSelection`	$00100000	User cannot select or edit text
`fDrawInactiveSelection`	$00080000	TextEdit displays a box around an inactive selection

■ **Table 49-2** TextEdit data structures

Name	Offset/Value	Type	Description
`TEColorTable`			
`contentColor`	$0000	Word	Color for inside of bounds rectangle
`outlineColor`	$0002	Word	Color for outline drawn around text
`hilightForeColor`	$0004	Word	Foreground color for highlighted text
`hilightBackColor`	$0006	Word	Background color for highlighted text; also used for caret
`vertColorDescriptor`	$0008	Word	Specifies type of reference in `vertColorRef`
`vertColorRef`	$000A	LongWord	Reference to color table for vertical scroll bar
`horzColorDescriptor`	$000E	Word	Specifies type of reference in `horzColorRef`
`horzColorRef`	$0010	LongWord	Reference to color table for horizontal scroll bar
`growColorDescriptor`	$0014	Word	Specifies type of reference in `growColorRef`
`growColorRef`	$0016	LongWord	Reference to color table for size box

◆ *Note:* All of the bits in each `TEColorTable` color word are significant. TextEdit forms color patterns by replicating the appropriate color word the appropriate number of times to form a QuickDraw II pattern.

`TEFormat` (format structure)

`version`	$0000	Word	Version number for format structure—value must be $0000
`rulerListLength`	$0002	LongWord	Size, in bytes, of `theRulerList` array
`theRulerList`	$0006	`TERuler`	Array of `TERuler` structures
`styleListLength`		LongWord	Size, in bytes, of `theStyleList` array
`theStyleList`		`TEStyle`	Array of `TEStyle` structures
`numberOfStyles`		LongWord	Number of entries in `theStyles` array
`theStyles`		`StyleItem`	Array of `StyleItem` structures

`TEParamBlock` (parameter block for creating TextEdit structures)

`pCount`	$0000	Word	Number of parameters to follow—values range from 7 through 23
`ID`	$0002	LongWord	Application assigned ID for record
`rect`	$0006	Rect	Bounding rectangle for entire TextEdit record, including scroll bars and outlines
`procRef`	$000E	LongWord	Must be set to $85000000
`flags`	$0012	Word	Control flags
`moreFlags`	$0014	Word	More control flags
`refCon`	$0016	LongWord	Reserved for application use
`textFlags`	$001A	LongWord	TextEdit control flags
`indentRect`	$001E	Rect	Specifies bounding rectangle for text in TextEdit record
`vertBar`	$0026	Handle	Handle to vertical scroll bar
`vertAmount`	$002A	Word	Number of pixels to scroll per click on vertical scroll arrows
`horzBar`	$002C	Handle	Reserved—must be set to NULL
`horzAmount`	$0030	Word	Reserved—must be set to $0000
`styleRef`	$0032	LongWord	Reference to initial style information for record
`textDescriptor`	$0036	Word	Defines format of `textRef`
`textRef`	$0038	LongWord	Reference to initial text for record
`textLength`	$003C	LongWord	Length of text referred to by `textRef`
`maxChars`	$0040	LongWord	Maximum number of characters allowed in record
`maxLines`	$0044	LongWord	Must be set to NULL
`maxCharsPerLine`	$0048	Word	Must be set to NULL
`maxHeight`	$004A	Word	Must be set to NULL
`colorRef`	$004C	LongWord	Reference to the `TEColorTable` for the record
`drawMode`	$0050	Word	QuickDraw II text mode
`filterProcPtr`	$0052	Pointer	Pointer to filter routine for the record

`TERecord` (control structure for TextEdit records)

`ctrlNext`	$0000	Handle	Handle to next control in control list
`inPort`	$0004	Pointer	Pointer to GrafPort for TextEdit record
`boundsRect`	$0008	Rect	Bounding rectangle for record

ctrlFlag	$0010	Byte	Low-order byte from TEParamBlock flags field
ctrlHilite	$0011	Byte	Reserved
lastErrorCode	$0012	Word	Last error generated by TextEdit
ctrlProc	$0014	LongWord	Always set to $85000000
ctrlAction	$0018	LongWord	Reserved
filterProc	$001C	Pointer	Pointer to filter procedure for the record
ctrlRefCon	$0020	LongWord	Reserved for application
colorRef	$0024	LongWord	Reference to TEColorTable for record
textFlags	$0028	LongWord	TEParamBlock textFlags field
textLength	$002C	LongWord	Length, in bytes, of text in record
blockList	$0030	TextList	TextList structure describing text for the record
ctrlID	$0038	Long	Application-assigned ID for the record
ctrlMoreFlags	$003C	Word	TEParamBlock moreFlags field
ctrlVersion	$003E	Word	Reserved
viewRect	$0040	Rect	Bounding rectangle for text on screen
totalHeight	$0048	LongWord	Total height of the text for the record, in pixels
lineSuper	$004C	SuperHandle	Root reference for text in record
styleSuper	$0058	SuperHandle	Root reference for styles in record
styleList	$0064	Handle	Handle to list of unique styles
rulerList	$0068	Handle	Handle to list of rulers
lineAtEndFlag	$006C	Word	Indicates whether last character was a line break
selectionStart	$006E	LongWord	Starting text offset for current selection
selectionEnd	$0072	LongWord	Ending text offset for current selection
selectionActive	$0076	Word	Indicates whether selection is active
selectionState	$0078	Word	Indicates whether selection is on screen
caretTime	$007A	LongWord	Tick count for caret blink
nullStyleActive	$007E	Word	Indicates whether null style is to be used
nullStyle	$0080	TEStyle	Style definition for null style
topTextOffset	$008C	LongWord	Offset into record text corresponding to top of screen

`topTextVPos`	$0090	Word	Difference between top of text and topmost scroll position
`vertScrollBar`	$0092	Handle	Handle of vertical scroll bar
`vertScrollPos`	$0096	LongWord	Current vertical scroll position
`vertScrollMax`	$009A	LongWord	Maximum allowable vertical scroll from top of text
`vertScrollAmount`	$009E	Word	Number of pixels to scroll on each vertical arrow click
`horzScrollBar`	$00A0	Handle	Handle of horizontal scroll bar (not supported)
`horzScrollPos`	$00A4	LongWord	Current horizontal scroll position (not supported)
`horzScrollMax`	$00A8	LongWord	Maximum allowable horizontal scroll from left boundary (not supported)
`horzScrollAmount`	$00AC	Word	Number of pixels to scroll on each horizontal arrow click (not supported)
`growBoxHandle`	$00AE	Handle	Handle to the Size box control
`maximumChars`	$00B2	LongWord	Maximum number of characters allowed in the text
`maximumLines`	$00B6	LongWord	Maximum number of lines allowed in the text (not supported)
`maxCharsPerLine`	$00BA	Word	Maximum number of characters allowed in a line (not supported)
`maximumHeight`	$00BC	Word	Maximum text height, in pixels (not supported)
`textDrawMode`	$00BE	Word	QuickDraw II drawing mode for the text
`wordBreakHook`	$00C0	Pointer	Pointer to routine to handle word breaks
`wordWrapHook`	$00C4	Pointer	Pointer to routine to handle word wrap
`keyFilter`	$00C8	Pointer	Pointer to keystroke filter routine
`theFilterRect`	$00CC	Rect	Rectangle for generic filter procedure
`theBufferVPos`	$00D4	Word	Vertical component of current position for generic filter procedure
`theBufferHPos`	$00D6	Word	Horizontal component of current position for generic filter procedure
`theKeyRecord`	$00D8	`KeyRecord`	Parameters for keystroke filter routine
`cachedSelcOffset`	$00E6	LongWord	Text offset for cached caret position

`cachedSelcVPos`	$00EA	Word	Vertical component of cached caret position
`cachedSelcHPos`	$00EC	Word	Horizontal component of cached caret position
`mouseRect`	$00EE	Rect	Bounding rectangle for mouse events
`mouseTime`	$00F6	LongWord	Tick count value when mouse button was last released
`mouseKind`	$00FA	Word	Kind of last click
`lastClick`	$00FC	Point	Location of last mouse click
`savedHPos`	$0100	Word	Saved horizontal position for up and down arrows
`anchorPoint`	$0102	Long	Anchor point for current selection

◆ *Note:* TextEdit maintains fields beyond `anchorPoint`. Applications should never access these fields or attempt to save the state of a TextEdit record by writing and reading the public fields documented here.

`TERuler` (ruler structure)

`leftMargin`	$0000	Word	Left indent pixel count for all lines except those that start paragraphs
`leftIndent`	$0002	Word	Left indent pixel count for lines that start paragraphs
`rightMargin`	$0004	Word	Right text boundary, measured from left edge of text rectangle
`just`	$0006	Word	Text justification flag
`extraLS`	$0008	Word	Spacing between lines (in pixels)
`flags`	$000A	Word	Control flags for the ruler
`userData`	$000C	LongWord	Reserved for application use
`tabType`	$0010	Word	Indicates type of tabs used
`theTabs`	$0012	`TabItem`	Array of `TabItems`, one for each absolute tab stop
`tabTerminator`		Word	Either the spacing for standard tabs or a flag terminating `theTabs` array

`TEStyle` (style description structure)

`fontID`	$0000	LongWord	FontID for text using this style
`foreColor`	$0004	Word	Foreground color for the style
`backColor`	$0006	Word	Background color for the style
`userData`	$0008	LongWord	Reserved for application use

KeyRecord

theChar	$0000	Word	Character value to translate
theModifiers	$0002	Word	Modifier key state bit flag (see Chapter 7, "Event Manager," for information on key modifiers)
theInputHandle	$0004	Handle	Handle to character in text
cursorOffset	$0008	LongWord	New cursor location
theOpCode	$000C	Word	Operation code for key filter routine

StyleItem (style reference structure)

length	$0000	LongWord	Number of text characters using the style
offset	$0004	LongWord	Byte offset into theStyleList to entry that defines the style for this text

SuperBlock

nextHandle	$0000	Handle	Handle to next SuperBlock in list
prevHandle	$0004	Handle	Handle to previous SuperBlock in list
textLength	$0008	LongWord	Number of bytes of text for this SuperBlock
	$000C	LongWord	Reserved
theItems	$0010	SuperItem	Array of SuperItems for this block

SuperHandle

cachedHandle	$0000	Handle	Handle to the current SuperBlock
cachedOffset	$0004	LongWord	Text offset to the start of the current SuperBlock
cachedIndex	$0008	Word	Index value for current SuperBlock
itemsPerBlock	$000A	Word	Number of SuperItems per SuperBlock

SuperItem

length	$0000	LongWord	Number of bytes of text for this SuperItem
data	$0004	LongWord	Data for the SuperItem

`TabItem` (tab stop descriptor)

`tabKind`	$0000	Word	Must be set to $0000
`tabData`	$0002	Word	Pixel offset to the tab stop from left boundary of text rectangle

`TextBlock`

`nextHandle`	$0000	Handle	Handle to next `TextBlock` in list
`prevHandle`	$0004	Handle	Handle to previous `TextBlock` in list
`textLength`	$0008	LongWord	Number of bytes of text in `theText`
`flags`	$000C	Word	Reserved
	$000E	Word	Reserved
`theText`	$0010	Byte	`textLength` bytes of text

`TextList`

`cachedHandle`	$0000	Handle	Handle to the current `TextBlock`
`cachedOffset`	$0004	LongWord	Text offset to the start of the current `TextBlock`

■ **Table 49-3** TextEdit error codes

Code	Name	Description
$2201	`teAlreadyStarted`	TextEdit has already been started
$2202	`teNotStarted`	TextEdit has not been started
$2203	`teInvalidHandle`	The *teHandle* parameter does not refer to a valid `TERecord`
$2204	`teInvalidDescriptor`	Invalid descriptor value specified
$2205	`teInvalidFlag`	Specified flag word is invalid
$2206	`teInvalidPCount`	Invalid parameter count value specified
$2207	Reserved	Reserved
$2208	`teBufferOverflow`	The output buffer was too small to accept all data
$2209	`teInvalidLine`	Starting line value is greater than the number of lines in the text (can be interpreted as end-of-file in some circumstances)
$220A	`teInvalidCall`	This call is not allowed for the supplied `TERecord`
$220B	`teInvalidParameter`	A passed parameter was invalid
$220C	`teInvalidTextBox2`	The `TextBox2` format codes were inconsistent

Chapter 50 **Text Tool Set Update**

This chapter documents new features of the Text Tool Set. The complete reference to the Text Tool Set is in Volume 2, Chapter 23 of the *Apple IIGS Toolbox Reference.*

New features in the Text Tool Set

The following information describes new features in this version of the Text Tool Set.

The Text Tool Set now supports the Slot Arbiter. All set device calls (such as `SetOutputDevice`, `SetInputDevice`, and so forth) accept slot numbers 1 through 7 or 9 through 15. Previously, the external slots, slots 9 through 15, were not valid for these calls. If your application specifies an external slot, the Text Tool Set will route the calls as appropriate. If your application specifies a slot from 1 through 7, the Text Tool Set determines whether the slot is internal or external, and routes the calls to the appropriate firmware.

Note that all get device calls still return slot numbers in the range from 1 through 7, in order to maintain compatability with existing code.

Chapter 51 **Tool Locator Update**

This chapter documents new features of the Tool Locator. The complete reference to the Tool Locator is in Volume 2, Chapter 24 of the *Apple IIGS Toolbox Reference.*

New features in the Tool Locator

This section explains new features of the Tool Locator.

- The Tool Locator uses a new algorithm to load tools from disk. It will load tools from disk only if it cannot find a tool in ROM with a version number as high as the requested version. The Tool Locator makes no assumptions about which tools are in ROM and which are on the system disk.

  For every tool that is to be loaded, the Tool Locator makes a version call. If the version call returns an error because the tool is not present, or the resulting version number is too low, then the tool is loaded from the system disk.
- The Tool Locator no longer unloads all RAM-based tools every time `TLShutDown` is called. Instead, it returns the system to a default state, set by a new call in the Tool Locator, `SetDefaultTPT`. This call can make any collection of RAM and ROM tools the default state. The system returns to the default state when `TLShutdown` is called.

Tool set startup and shutdown

The Tool Locator now provides calls that automatically start and stop specified tool sets in the correct order. These calls, `StartUpTools` and `ShutDownTools`, are documented in "New Tool Locator calls" later in this chapter.

The `StartUpTools` call performs the following steps during startup processing:

1) Starts the Resource Manager
2) Opens the resource fork for the current application
3) Obtains memory for the application's direct page
4) Starts the tools specified in the input `StartStop` record; updates the `StartStop` record as appropriate
5) Returns the `StartStop` record reference to the calling program

Your application must pass this returned `StartStop` record reference to `ShutDownTools` at tool shutdown time.

`StartUpTools` sets some tool set default values for you. If these value are not appropriate for your application, you should change them by issuing the appropriate tool calls after `StartUpTools` has returned:

QuickDraw II

Started with the video mode from the input `StartStop` record—the `QDStartUp` *maxWidth* parameter is set to 160 bytes.

QuickDraw II Auxiliary

System calls `WaitCursor`; your application must change the cursor to an arrow before accepting user input.

Event Manager

Queue size set to 20, maximum mouse clamp set to either 320 or 640, depending upon the video mode specified in the `StartStop` record.

Note Sequencer

Update rate set to 0 (use default Note Synthesizer rate), increment set to 20, and interrupts have been disabled (the system calls `StopInts`). Your program must enable interrupts with `StartInts`. The Note Sequencer will automatically start the Sound Tool Set and the Note Synthesizer, if you have not included them in your `StartStop` record.

`ShutDownTools` performs the following steps during tool set shutdown:

1) Shuts down tools specified in input `StartStop` record

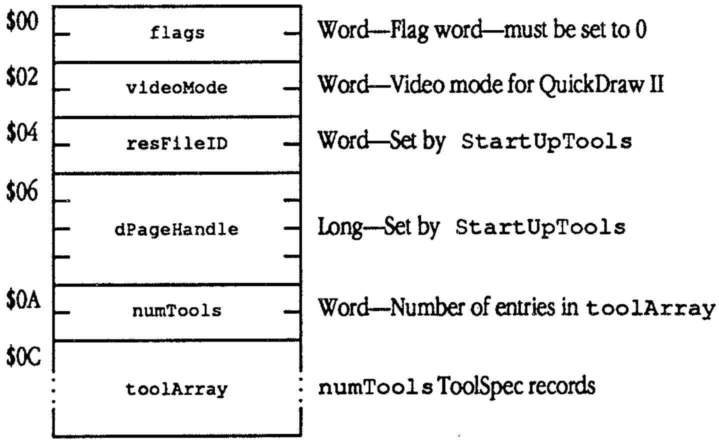
2) Disposes of handle to direct page

3) Disposes of handle to `StartStop` record (unless pointer was passed)

4) Shutsdown the Resource Manager

Both these calls require that your application format a tool `StartStop` record. That record is defined as follows:

- **Figure 51-1** Tool set StartStop record



$00	`flags`	Word—Flag word—must be set to 0
$02	`videoMode`	Word—Video mode for QuickDraw II
$04	`resFileID`	Word—Set by `StartUpTools`
$06	`dPageHandle`	Long—Set by `StartUpTools`
$0A	`numTools`	Word—Number of entries in `toolArray`
$0C	`toolArray`	`numTools` ToolSpec records

`videoMode` Defines the video mode for QuickDraw II. See Chapter 16, "QuickDraw II," in the *Toolbox Reference* for valid values.

`resFileID` The `StartUpTools` call sets this field. `ShutDownTools` requires it as input.

`dPageHandle` The `StartUpTools` call sets this field. `ShutDownTools` requires it as input.

`toolArray` Each entry defines a tool set to be started. The `numTools` field specifies the number of entries in this array. Each entry is formatted as follows:

$00	`toolNumber`	Word—Tool set identifier
$02	`minVersion`	Word—Minimum acceptable tool set version

`toolNumber` Specifies the tool set to be loaded Valid tools set numbers are listed in Table 51-1.

`toolVersion` Specifies the minimum acceptable version for the tool set. See Chapter 24, "Tool Locator," in the *Toolbox Reference* for the format of this field.

Tool set numbers

Table 51-1 lists the tool set numbers for all tool sets supported by the `StartUpTools` and `ShutDownTools` calls.

- **Table 51-1** Tool set numbers

Tool set number		Tool set name
$01	#01	Tool Locator
$02	#02	Memory Manager
$03	#03	Miscellaneous Tool Set
$04	#04	QuickDraw II
$05	#05	Desk Manager
$06	#06	Event Manager
$07	#07	Scheduler
$08	#08	Sound Tool Set
$09	#09	Apple Desktop Bus Tool Set
$0A	#10	SANE Tool Set
$0B	#11	Integer Math Tool Set
$0C	#12	Text Tool Set
$0D	#13	Reserved for internal use
$0E	#14	Window Manager
$0F	#15	Menu Manager
$10	#16	Control Manager
$11	#17	System Loader
$12	#18	QuickDraw II Auxiliary
$13	#19	Print Manager
$14	#20	LineEdit Tool Set
$15	#21	Dialog Manager
$16	#22	Scrap Manager
$17	#23	Standard File Operations Tool Set
$18	#24	Not available
$19	#25	Note Synthesizer
$1A	#26	Note Sequencer
$1B	#27	Font Manager
$1C	#28	List Manager

$1D	#29	Audio Compression (ACE)
$1E	#30	Resource Manager
$20	#32	MIDI Tool Set
$22	#34	TextEdit Tool Set

Tool set dependencies

Although `StartUpTools` handles the order of tool startup for you, it does not manage tool set dependencies. It is your responsibility to specify all required tool sets in order to ensure correct system operation. Table 51–2 documents current tool set dependencies.

■ **Table 51-2** Tool set dependencies

Tool set and number	Depends upon
Tool Locator(#01)	No dependencies; always started first
Memory Manager (#02)	Tool Locator (#01)
Miscellaneous Tool Set (#03)	Tool Locator (#01) Memory Manager (#02)
QuickDraw II (#04)	Tool Locator (#01) Memory Manager (#02) Miscellaneous Tool Set (#03)
Desk Manager (#05)	Tool Locator (#01) Memory Manager (#02) Miscellaneous Tool Set (#03) QuickDraw II (#04) Event Manager (#06) Window Manager (#14) Control Manager (#16) Menu Manager (#15) LineEdit (#20) Dialog Manager (#21) Scrap Manager (#22)
Event Manager (#06)	Tool Locator (#01) Memory Manager (#02) Miscellaneous Tool Set (#03)
Scheduler (#07)	Tool Locator (#01) Memory Manager (#02) Miscellaneous Tool Set (#03)
Sound Tool Set (#08)	Tool Locator (#01) Memory Manager (#02) Miscellaneous Tool Set (#03)
Apple Desktop Bus (#09)	Tool Locator (#01)
SANE Tool Set (#10)	Tool Locator (#01) Memory Manager (#02)

Integer Math Tool Set (#11)	Tool Locator (#01)	
Text Tool Set (#12)	Tool Locator (#01)	
Window Manager (#14)	Tool Locator (#01)	
	Memory Manager (#02)	
	Miscellaneous Tool Set (#03)	
	QuickDraw II (#04)	
	Event Manager (#06)	
	Control Manager (#16)	
	Menu Manager (#15)	
	LineEdit (#20)	For `AlertWindow` call
	Font Manager (#27)	For `AlertWindow` call
	Resource Manager (#30)	Only if you use resources
Menu Manager (#15)	Tool Locator (#01)	
	Memory Manager (#02)	
	Miscellaneous Tool Set (#03)	
	QuickDraw II (#04)	
	Event Manager (#06)	
	Window Manager (#14)	
	Control Manager (#16)	
	Resource Manager (#30)	Only if you use resources
Control Manager (#16)	Tool Locator (#01)	
	Memory Manager (#02)	
	Miscellaneous Tool Set (#03)	
	QuickDraw II (#04)	
	Event Manager (#06)	
	Window Manager (#14)	
	Menu Manager (#15)	
	Resource Manager (#30)	Only if you use resources or icon buttons

◆ *Note*: You should consider the Window, Control, and Menu managers as one unit, and always start them together and in that order.

System Loader (#17)	Tool Locator (#01)
	Memory Manager (#02)
	Miscellaneous Tool Set (#03)

QuickDraw II Auxiliary (#18)	Tool Locator (#01)	
	Memory Manager (#02)	
	Miscellaneous Tool Set (#03)	
	QuickDraw II (#04)	
	Font Manager (#27)	

◆ *Note*: QuickDraw II Auxiliary uses the Font Manager in the picture drawing routines. For proper operation, you should start the Font Manager before using the QuickDraw II Auxiliary picture routines; however, the picture routines will not fail if the Font Manager is not present.

Print Manager (#19)	Tool Locator (#01)	
	Memory Manager (#02)	
	Miscellaneous Tool Set (#03)	
	QuickDraw II (#04)	
	QuickDraw II Auxiliary (#18)	
	Event Manager (#06)	
	Window Manager (#14)	
	Control Manager (#16)	
	Menu Manager (#15)	
	LineEdit (#20)	
	Dialog Manager (#21)	
	List Manager (#28)	
	Font Manager (#27)	
LineEdit Tool Set (#20)	Tool Locator (#01)	
	Memory Manager (#02)	
	Miscellaneous Tool Set (#03)	
	QuickDraw II (#04)	
	Event Manager (#06)	
	QuickDraw II Auxiliary (#18)	For `Text2` items only
	Scrap Manager (#22)	
	Font Manager (#27)	For `Text2` items only

Dialog Manager (#21)	Tool Locator (#01)	
	Memory Manager (#02)	
	Miscellaneous Tool Set (#03)	
	QuickDraw II (#04)	
	Event Manager (#06)	
	Window Manager (#14)	
	Control Manager (#16)	
	Menu Manager (#15)	
	QuickDraw II Auxiliary (#18)	For `Text2` items only
	LineEdit (#20)	
	Font Manager (#27)	For `Text2` items only

◆ *Note:* LineEdit and Dialog Manager require Font Manager and QuickDraw II Auxiliary if you use `LETextBox2` or `LongStatText2` which require any font styling (for example, outline, boldface, and so on).

Scrap Manager (#22)	Tool Locator (#01)
	Memory Manager (#02)
Standard File Tool Set (#23)	Tool Locator (#01)
	Memory Manager (#02)
	Miscellaneous Tool Set (#03)
	QuickDraw II (#04)
	Event Manager (#06)
	Window Manager (#14)
	Control Manager (#16)
	Menu Manager (#15)
	LineEdit (#20)
	Dialog Manager (#21)
Note Synthesizer (#25)	Tool Locator (#01)
	Memory Manager (#02)
	Sound Tool Set (#08)
Note Sequencer (#26)	Tool Locator (#01)
	Memory Manager (#02)
	Sound Tool Set (#08)
	Note Synthesizer (#25)

◆ *Note:* The Note Sequencer automatically handles the startup and shutdown of the Sound Tool Set (#8) and the Note Synthesizer (#25).

Font Manager	Tool Locator (#1)	
	Memory Manager (#2)	
	Miscellaneous Tool Set (#3)	For `ChooseFont` only
	QuickDraw II (#4)	
	Integer Math (#11)	For `ChooseFont` only
	Window Manager (#14)	For `ChooseFont` only
	Control Manager (#16)	For `ChooseFont` only
	Menu Manager (#15)	For `FixFontMenu` only
	List Manager (#28)	For `FixFontMenu` and `ChooseFont` only
	LineEdit (#20)	For `ChooseFont` only
	Dialog Manager (#21)	For `ChooseFont` only
List Manager (#28)	Tool Locator (#01)	
	Memory Manager (#02)	
	Miscellaneous Tool Set (#03)	
	QuickDraw II (#04)	
	Event Manager (#06)	
	Window Manager (#14)	
	Control Manager (#16)	
	Menu Manager (#15)	
Audio Compression (ACE) (#29)	Tool Locator (#01)	
	Memory Manager (#02)	
MIDI Tool Set (#32)	Tool Locator (#01)	
	Memory Manager (#02)	
	Miscellaneous Tool Set (#03)	
	Sound Tool Set (#8)	
	Note Synthesizer (#25)	For time-stamping only

◆ *Note:* The MIDI Tool Set requires the Note Synthesizer in order to support the MIDI clock feature. If you are not using MIDI clock, the Note Synthesizer is not required.

Resource Manager (#30)	Tool Locator (#01)

TextEdit Tool Set (#34)	Tool Locator (#01)	Version $0300
	Miscellaneous Tool Set (#03)	Version $0300
	QuickDraw II (#04)	Version $0300
	QuickDraw II Auxiliary (#18)	Version $0206
	Event Manager (#06)	Version $0300
	Window Manager (#14)	Version $0300
	Control Manager (#16)	Version $0300
	Menu Manager (#15)	Version $0300
	Scrap Manager (#22)	Version $0104
	Font Manager (#27)	Version $0204
	Resource Manager (#30)	Version $0100

New Tool Locator calls

The call `SetDefaultTPT` has been added to the Tool Locator to facilitate permanent tool patches. `StartUpTools` and `ShutDownTools` provide automatic services for bringing up or removing tool sets. The `MessageByName` tool call provides facilities for your application to use the message center.

`MessageByName` $1701

Creates and associates a name with a new message, providing a convenient and extensible mechanism for creating, tracking, and passing messages between programs. Your application can then use the other Message Center Tool Locator calls to manipulate or delete the message.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
createItFlag	Word—Boolean; create message?
– *recordPointer* –	Long—Pointer to input record
	<—SP

Stack after call

Previous contents	
– *responseRecord* –	Long—Response record from call
	<—SP

Errors

$0111	`messageNotFound`	No message found with specified name
$0112	`messageOvfl`	No message numbers available
$0113	`nameTooLong`	Message name too long
Memory Manager errors		Errors from `NewHandle` returned unchanged

C

```
extern pascal responseRecord
          MessageByName(createItFlag,
          recordPointer);

Boolean   createItFlag;
Pointer   recordPointer;
```

createItFlag Determines whether to create a message containing the information from the input record. If there is no existing message with the specified name, then the setting of *createItFlag* governs whether to create a message. If there is already a message with the specified name, then the setting of *createItFlag* determines whether to replace that existing message with a new one based upon the input record.

recordPointer Pointer to an input record, which defines the content and characteristics of the new message:

$00	`blockLen`	Word—Length of record (including `blockLen`)
$02	`nameString`	n Bytes—Identifier string for the message
$02+n	`dataBlock`	m Bytes—Optional data for message

`blockLen` Contains the length, in bytes, of the input record. Note that the value for this field includes the length of `blockLen`.

`nameString` The identifier for the new message. This is a standard Pascal string (length byte followed by ASCII data), with a maximum length of 64 bytes (not including the length byte). In order to prevent message name conflict, this name string should contain the the manufacturer's name, followed by the product name and/or code, followed by a unique identifying string. You may set the high-order bits of each byte; note, however, that the system does not include these bits in name comparisons.

`dataBlock` Application-defined data that are copied into a created message. Use of this field is optional.

`responseRecord` Contains the response information from the call:

$00	`createFlag`	Word—Boolean; was message created?
$02	`messageID`	Word—ID number for created message

`createFlag` Indicates whether `MessageByName` created a message. Note that if you set *createItFlag* to TRUE on input to `MessageByName`, and there was already a message with the specified `nameString` in the Message Center, then this flag will be FALSE, and `messageID` will identify the message into which your `dataBlock` was copied.

`messageID`` Message ID for new mesage, if MessageByName created one.

`SetDefaultTPT` $1601

Sets the default Tool Pointer Table (TPT) to the current TPT. Used to permanently install a tool patch.

▲ **Warning** An application should not make this call.▲

Parameters This call has no input or output parameters. The stack is unaffected.

C

```
extern pascal void SetDefaultTPT();
```

`ShutDownTools` $1901

Shuts down the tools specified in the input `StartStop` record.

Your program must pass the `StartStop` record reference that was returned by `StartUpTools`.

Parameters

Stack before call

Previous contents	
startStopDesc	Word—Defines the type of reference stored in *startStopRecRef*
– *startStopRecRef* –	Long—Reference to the `StartStop` record
	<—SP

Stack after call

Previous contents	
	<—SP

Errors None

C

```
extern pascal void ShutDownTools(startStopDesc,
                    startStopRecRef);

Word        startStopDesc;
Long        startStopRecRef;
```

startStopDesc Defines the type of reference stored in *startStopRecRef*:

0	Reference is a pointer
1	Reference is a handle

startStopRecRef Reference to the updated `StartStop` record returned by `StartUpTools`.

`StartUpTools` $1801

Starts and loads the tools specified in the input `StartStop` record. Upon successful return from `StartUpTools`, the specified tools will have been started and the cursor will be represented by the watch image (if QuickDraw II Auxiliary was loaded). Your program should change the cursor image before accepting user input.

Your program must pass the `StartStop` record reference that was returned by `StartUpTools` to the `ShutDownTools` call at tool shutdown time.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
userID	Word—Application User ID for system calls
startStopDesc	Word—Defines the type of reference stored in *startStopRecRef*
– *startStopRecRef* –	Long—Reference to the `StartStop` record
	<—SP

Stack after call

Previous contents	
–*startStopRecRefRet*–	Long—Reference to resulting `StartStop` record
	<—SP

Errors	$0103	`TLBadRecFlag`	`StartStop` record invalid
	$0104	`TLCantLoad`	A tool cannot be loaded–check input `StartStop` record for valid tool numbers, and versions, and for correct `numTools` value
	System Loader errors		Returned unchanged
	Memory Manager errors		Returned unchanged
	GS/OS errors		Returned unchanged

C

```
extern pascal long StartUpTools(userID,
          startStopDesc, startStopRecRef);

Word      userID, startStopDesc;
Long      startStopRecRef;
```

startStopDesc Defines the type of reference stored in *startStopRecRef*:

0 Reference is a pointer
1 Reference is a handle
2 Reference is a resource ID

startStopRecRefRet Reference to the updated `startStop` record. Your application must pass this record to the `ShutDownTools` tool call. If the input record reference to `StartUpTools` was a pointer, then this reference is also a pointer. If the input reference was either a handle or a resource ID, then `StartUpTools` returns a handle.