For Apple IIGS and 1 MB Apple IIGS

Apple IIGS® Toolbox Reference

Volume 3

Addison-Wesley Publishing Company, Inc.

Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan

APPLE COMPUTER, INC.

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

Simultaneously published in the United States and Canada.

ISBN 0-201-55019-9
ABCDEFGHIJ-MU-943210
First printing, MAY 1990

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

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.

For Apple IIGS and 1 MB Apple IIGS

Apple IIGS® Toolbox Reference

Volume 3

The Official Publication from Apple Computer, Inc.

The *Apple IIGS Toolbox Reference* is a comprehensive guide to the Apple IIGS Toolbox, which contains more than 1000 ready-to-use tool set routines. These routines enable programmers and developers to access the powerful capabilities of the Apple IIGS personal computer and write programs that comply with the Apple desktop interface standards. Using the Toolbox also frees programmers from much of the tedious background "bookkeeping" that would otherwise be required to maintain that interface.

The *Apple IIGS Toolbox Reference* consists of three volumes that together provide a complete description of the Toolbox. This volume, Volume 3, contains descriptions of hundreds of changes and additions to the original set of programming tools, including:

- Complete documentation for the new Resource Manager and TextEdit Tool Set
- Descriptions of the new sound-related tool sets (the Audio Compression and Expansion Tool Set, the MIDI Tool Set, the Note Sequencer, and the Note Synthesizer)
- Details on how to use the newly expanded support for controls

Volume 1 begins with a brief overview of the tool sets contained in the Toolbox at the time of publication. Following this introduction, each of the remaining chapters describes one of the tool sets. Arranged alphabetically by tool set name, the chapters include the following information:

- An overview of what routines are in the tool set and how they can be used
- A complete description of each routine, with the parameters for the programming languages, and possible errors. Examples, figures, and tables give additional information about the routines.
- A summary of the constants, data structures, and error codes for the tool set

Volume 2 follows the same format, describing the tool sets not covered in the first volume. It also provides appendixes and a glossary , along with an index covering the first two volumes.

The *Apple IIGS Toolbox Reference* is an indispensable resource for the programmer writing programs that access the full range of capabilities of the Apple IIGS.

Contents

Figures and tables

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. It contains 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 the Apple Programmers and Developers Association (APDA™).

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 *Apple IIGS 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
Appendix G, "Toolbox Code Example"	Presents a sample program, BusyBox, which illustrates the use of many of the new features of the Apple IIGS Toolbox

Typographical conventions

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

Call format

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

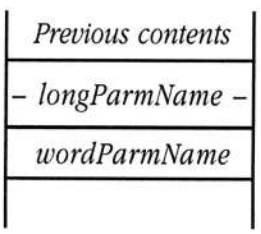
Certain elements of this format may not appear in all calls. For example, stack diagrams are omitted from those calls that do not affect the stack.

`ToolCallName` $callnumber

A description of the call's function.

Parameters

Stack before call

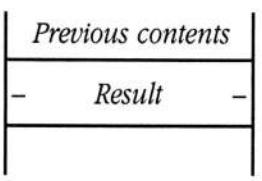


Long—Description of *longParmName* parameter

Word—Description of *wordParmName* parameter

<—SP

Stack after call



Long—Description of call result value (if any)

<—SP

Errors	$xxxx	`Error name`	Description of the error code.
C	`C code.` The C language function declaration for the call.		
stackParameter	Detailed description of stack input or output parameter, where appropriate.		

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 corrections

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 list correctly describes `ReadConfigRec`, the configuration record for the `ReadKeyMicroData` tool call.

Name	Offset	Type	Definition
`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

An error exists in the *Apple IIGS Toolbox Reference Update* (distributed by APDA™). The description of the `ACEExpand` tool call contains 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 require significant processor resources, and therefore take some time.

The following list summarizes the capabilities of the ACE Tool Set. The tool calls are grouped according to function. Later sections of this chapter discuss audio compression and expansion in greater detail and define the precise syntax of the tool calls.

Routine	Description
Housekeeping routines	
`ACEBootInit`	Called only by the Tool Locator—must not be called by an application
`ACEStartUp`	Initializes the ACE Tool Set for use by an application
`ACEShutDown`	Informs the ACE Tool Set that an application is finished using its tool calls
`ACEVersion`	Returns the ACE Tool Set version number

`ACEReset`	Called only when the system is reset—must not be called by an application
`ACEStatus`	Returns the operational status of the ACE Tool Set
`ACEInfo`	Returns information about the ACE Tool Set operating environment

Audio compression and expansion tool calls

`ACECompBegin`	Prepares the ACE tools to compress an audio sequence
`ACECompress`	Compresses an audio sequence
`ACEExpand`	Expands a previously compressed audio sequence
`ACEExpBegin`	Prepares the ACE tools to expand a previously compressed audio sequence

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 is likely to be and a dynamic model of what the next actual change should be, based on the values last observed. ADPCM 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, ADPCM records an erroneous value. In general, a certain statistically predictable amount of error 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 is likely to 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 the values on either side of it. 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, which is specified by the application code. With ADPCM the programmer can specify encoded values either 3 or 4 bits wide. Because 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 status information about the tool set.

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

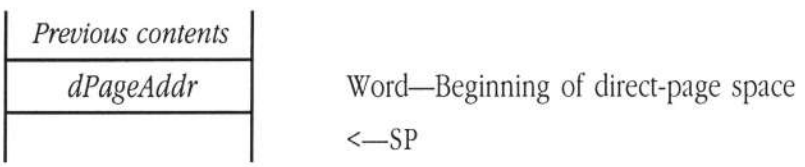
```
extern pascal void ACEBootInit();
```

ACEStartUp $021D

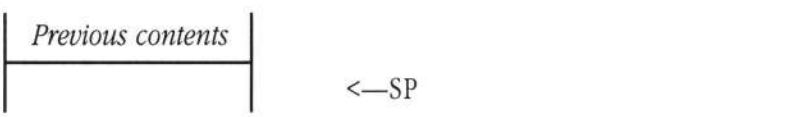
Initializes the ACE tools for use by an application. The `ACEStartUp` routine sets aside a region of bank $00, specified by *dPageAddr,* for use as the ACE tools' direct page. At present, ACE uses one 256-byte page of bank $00 memory as its direct page. Because future versions of the ACE tools may use a different amount of memory for the direct page, 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



Stack after call



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

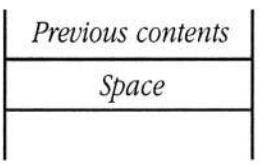
```
extern pascal void ACEShutDown();
```

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

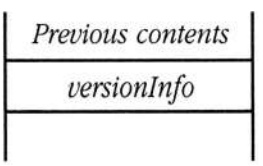
Stack before call



Word—Space for result

<—SP

Stack after call



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();
```

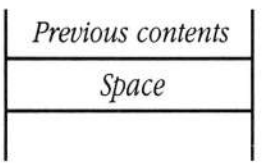
ACEStatus $061D

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

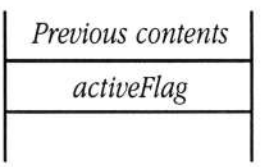
Stack before call



Word—Space for result

<—SP

Stack after call



Word—Boolean value indicating whether tool set is active

<—SP

Errors None

C

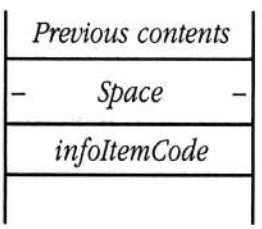
```
extern pascal Boolean ACEStatus();
```

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

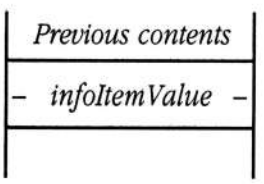


Long—Space for result

Word—What type of information to return

<—SP

Stack after call



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 *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

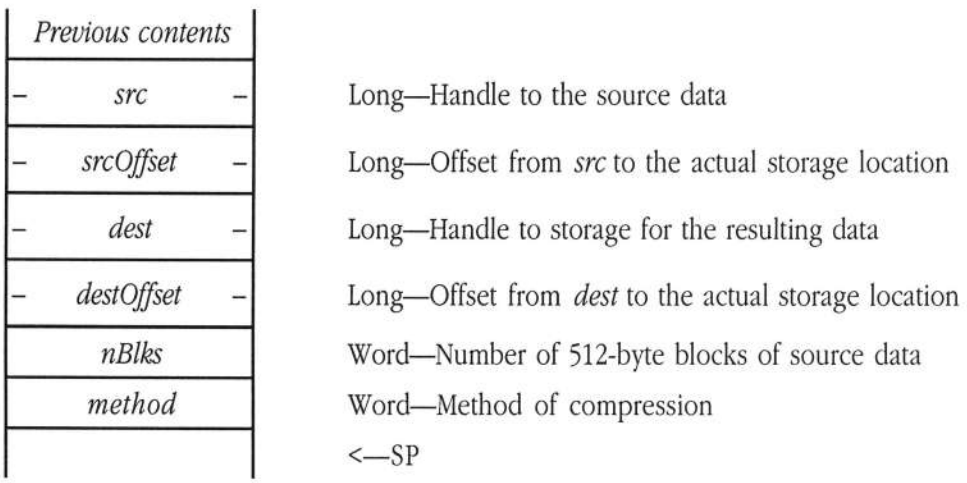
Compresses a number of blocks of digital audio data and stores the compressed data at a specified location. Each input block contains 512 bytes of data to be compressed. Your program also specifies the compression method, using the *method* parameter.

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



Stack after call

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 set a starting location 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), as in this formula:

((*nBlks* * 512) * 4) / 8

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

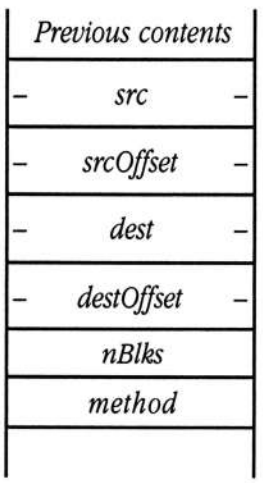
((*nBlks* * 512) * 3) / 8

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 because the resulting data is 2 to 2.67 times as large as the source.

Parameters

Stack before call



Long—Handle to the source data

Long—Offset from *src* to the actual storage location

Long—Handle to storage for the resulting data

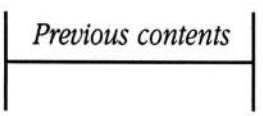
Long—Offset from *dest* to the actual storage location

Word—Number of 512-byte blocks to be stored at *dest*

Word—Method of compression

<—SP

Stack after call



<—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 set a starting location 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 before expanding 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, however, 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

Table 27-1 lists the error codes that may be returned by Audio Compression and Expansion Tool Set calls.

■ **Table 27-1** ACE Tool Set error codes

Value	Name	Definition
$0000	aceNoError	No error
$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
$1DFF	aceNotImplemented	The requested function has not been implemented

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 *Apple IIGS 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	Outline color	
		bits 15–8	= zero
		bits 7–4	= outline color
		bits 3–0	= zero
`growNorBack`	word	Color of interior when not highlighted	
		bits 15–8	= zero
		bits 7–4	= background color
		bits 3–0	= icon color
`growSelBack`	**word**	**Color of interior when highlighted**	
		bits 15–8	**= zero**
		bits 7–4	**= background color**
		bits 3–0	**= icon color**

- A statement on page 4-76 of the *Toolbox Reference,* in the section that covers the `SetCtlParams` call, is not strictly accurate. The statement that the call "sets new parameters to the control's definition procedure" is misleading; the call does not set the parameters directly. 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 to the definition procedure.

Clarifications

The following items provide additional information about features previously described in Volume 1 of the *Toolbox Reference*.

- The `barArrowBack` entry in the scroll bar color table was never implemented as first intended and is 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 of 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 of 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 allows dithered colors to 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 allowable value of `ctlValue` 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 the 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, TaskMaster will return 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 available only to controls that have been created with the `NewControl2` tool call, and for which the `fCtlWantEvents` 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 more than one edit field control can be used 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 the edit field control that 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. Note further that the type of reference you use when you specify data for the Control Manager governs how that data is later accessed. For example, if you originally specify the color table for a control with a handle, then anytime the system returns a reference to that color table, the reference is a handle; similarly, your application must always refer to that color table with a handle.

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 loads the code when it is needed.

The Control Manager handles resources differently according to the relative permanence of the data. For temporary information, the 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 system may store other definition procedures 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," 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 Tool Set," 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.

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 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 all controls created by `NewControl2`, not just the new control types, are extended controls. 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 call `NewControl2` directly 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 in Controls" 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 in Controls" 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 as simple button controls do. 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 in Controls" 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, the portion of the icon that lay outside the rectangle is not erased when you erase the control.

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, so your application should ensure that the system disk is online before defining an icon button control. Your application can prompt the user to insert the system disk if it is not already online.

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 (see Chapter 34, "LineEdit Tool Set Update," for information about the new features in the LineEdit Tool Set).

LineEdit controls respond to both mouse and keyboard events. If your application uses TaskMaster, the system handles 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 of TaskMaster).

If your application does not use TaskMaster, your application must call `TrackControl` to track the mouse and perform appropriate text selection when the user presses the mouse button in a LineEdit control. TaskMaster does 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 does all this for you if you have set the `tmControlKey` flag to 1 in the `taskMask` field.

To keep the insertion point blinking, your application must send idle events to the LineEdit control. 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 does 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. To understand how to use this control in your application, see Chapter 35, "List Manager Update," in this book.

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 dimensions of the original picture frame are not the same as those of 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 of TaskMaster).

If your application does not use TaskMaster, your application must call `TrackControl` to track the mouse and present the pop-up menu to the user when the user presses the mouse button inside a pop-up control. TaskMaster does 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 does 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 select a button by pressing an assigned key on the keyboard. See "Keystroke Processing in Controls" 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 that automatically invoke `GrowWindow` and `SizeWindow` if you create the control with the `NewControl2` tool call. When the user drags the size box, the Control Manager calls `GrowWindow` and `SizeWindow` to track the control and resize the window rectangle 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). 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. Static text controls accept initial text in the same format as the `LETextBox2` LineEdit tool call does. Consequently, you can place font, style, size, and color changes into the displayed text, affording you great freedom to create a distinctive text display (see "LETextBox2" in Chapter 11, "List Manager," in Volume 1 of the *Toolbox Reference* for information on the embedded change codes accepted by `LETextBox2`). In addition, static text 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 create TextEdit controls only with the `NewControl2` tool call.

The TextEdit control uses the TextEdit tool set. This new tool set is completely described in Chapter 49, "TextEdit Tool Set." 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 must support these 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 size has changed
17	`ctlHandleTab`	By pressing the Tab key, the user has moved to a control that can be the target
18	`ctlNotifyMultiPart`	A multipart control (a control that owns separate visible items) must be hidden, drawn, or shown
19	`ctlWindStateChange`	Window state has changed

In addition, the `initCtl`, `dragCtl`, 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 control definition procedure is finished drawing, restore the previous GrafPort. Note that saving the current GrafPort includes saving the pen state, 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 32,768).

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 disable 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 definition procedure 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.

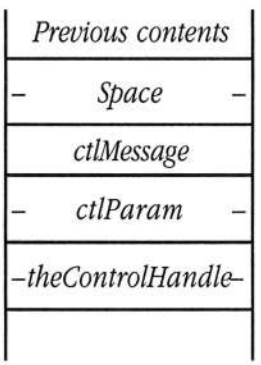
Event routine

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 `fCtlWantEvents` 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 has 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 control definition procedure handles the event, TaskMaster passes the event to the application.

If your custom control can be the target control, your event routine should issue the `MakeNextCtlTarget` tool call whenever the user presses the Tab key. When your routine regains control after that call, it should check whether another control became the target control. If so, your routine should send a `ctlHandleTab` control message to that control definition procedure. In either case, your routine must indicate that it handled the Tab key event by setting *retValue* to $FFFFFFFF on return from the Event routine.

Parameters

Stack before call



Long—Space for result

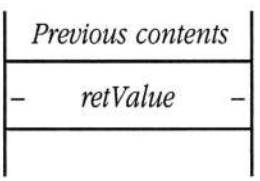
Word—`ctlHandleEvent` message

Long—Pointer to task record containing event information

Long—Handle to control

<—SP

Stack after call



Long—$FFFFFFFF if control took the event; $0 if control did not

<—SP

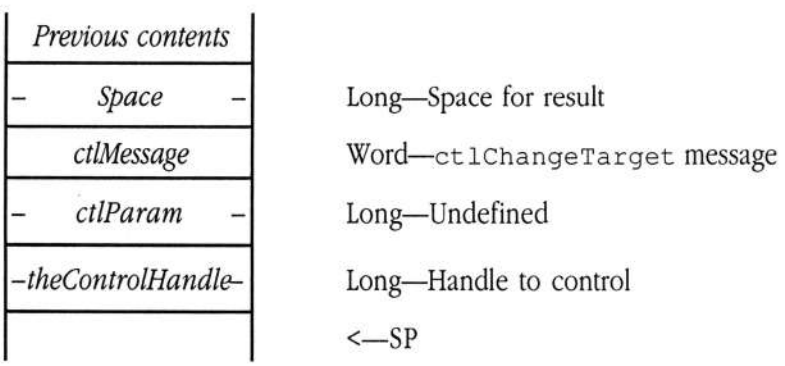
Target routine

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. This bit is set to 1 in the control record of the new target control. In the previous target, the bit is set to 0.

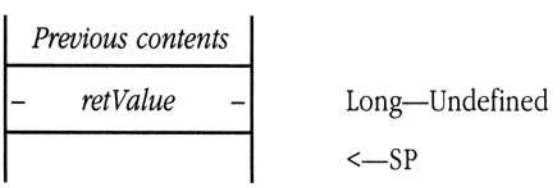
In response to the `ctlChangeTarget` message, some control definition procedures change the appearance of their control on the screen or perform other actions as appropriate. For example, LineEdit and TextEdit controls display an insertion point or a text selection only when they are the target.

Parameters

Stack before call



Stack after call



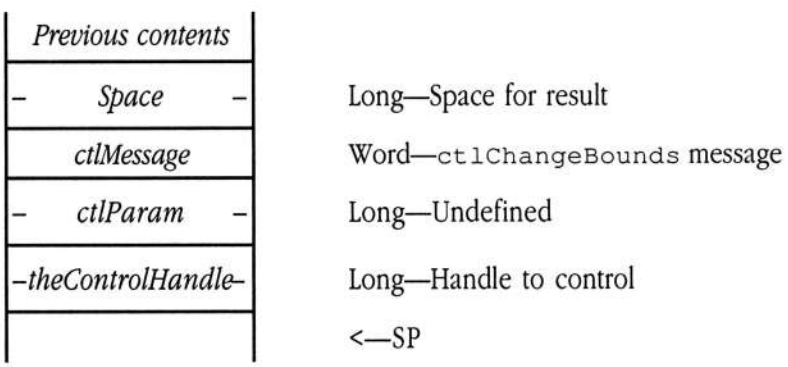
Bounds routine

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.

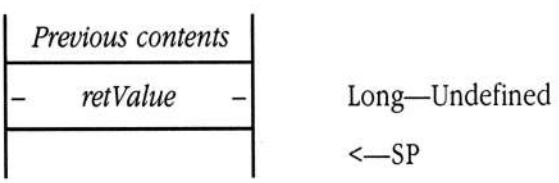
◆ *Note:* This message is not supported by control definition procedures currently provided by Apple Computer, Inc.; however, you should handle this message in any custom controls you create.

Parameters

Stack before call



Stack after call

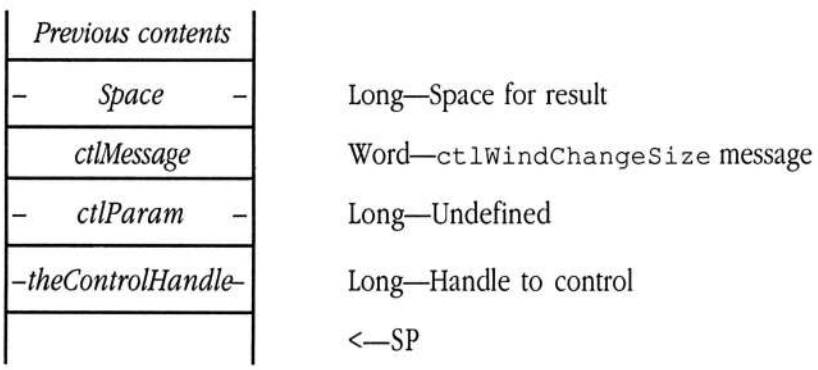


Window size routine

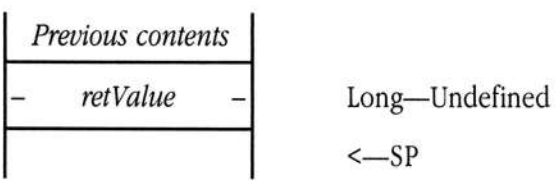
The Control Manager calls the control definition procedure with the `ctlWindChangeSize` message whenever the user changes the size of the control window. In response to this message, your control definition procedure should do what is necessary to maintain a consistent screen presentation. This may entail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

Parameters

Stack before call



Stack after call

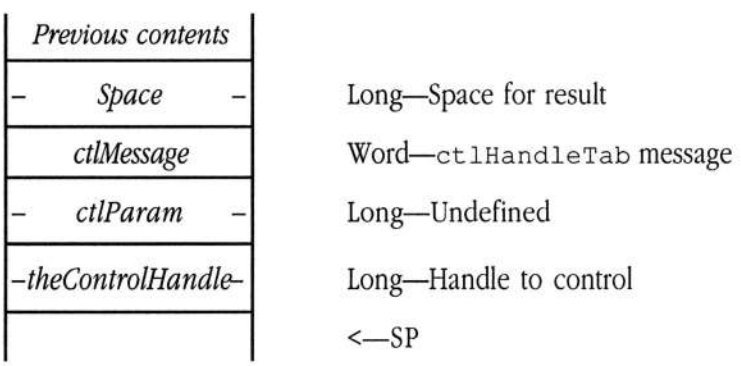


Tab routine

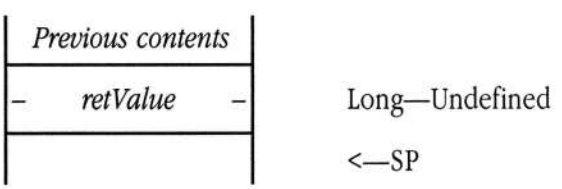
Your control definition procedure receives the `ctlHandleTab` message when the user presses the Tab key while another control is the target. That control's definition procedure issues the `MakeNextCtlTarget` tool call before sending this control message (see "Event Routine" earlier in this chapter). Your definition procedure receives the `ctlChangeTarget` control message before it receives the `ctlHandleTab` message. 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



Stack after call



Notify multipart routine

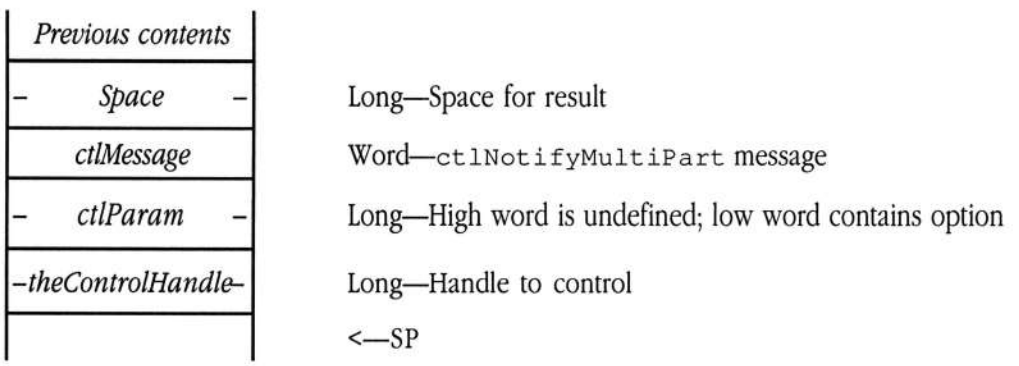
The Control Manager calls the control definition procedure with the `ctlNotifyMultiPart` message to signal that a multipart control needs to be hidden, shown, or drawn. This message is relevant only to multipart controls, which include other displayable entities that do not fit within the boundary rectangle. For example, list controls consist of the list itself and a scroll control, which is separate, and are therefore multipart controls. By contrast, the scroll control itself is *not* a multipart control because its component parts (arrows, page regions, and thumb) are fully contained in the scroll control boundary rectangle, and are not separate functional entities. 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 definition procedure must do what is needed to hide or show the control completely.

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

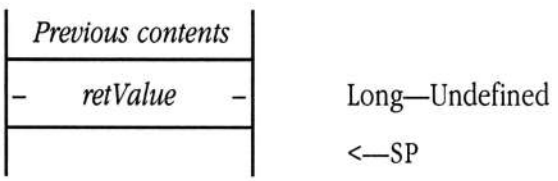
0	Hide the entire control
1	Erase the entire control
2	Show the entire control
3	Show one control

Parameter

Stack before call



Stack after call



Window change routine

The Control Manager calls the control definition procedure with the `ctlWindStateChange` message to signal that the state of the window containing the control has changed. For example, a control definition procedure receives 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.

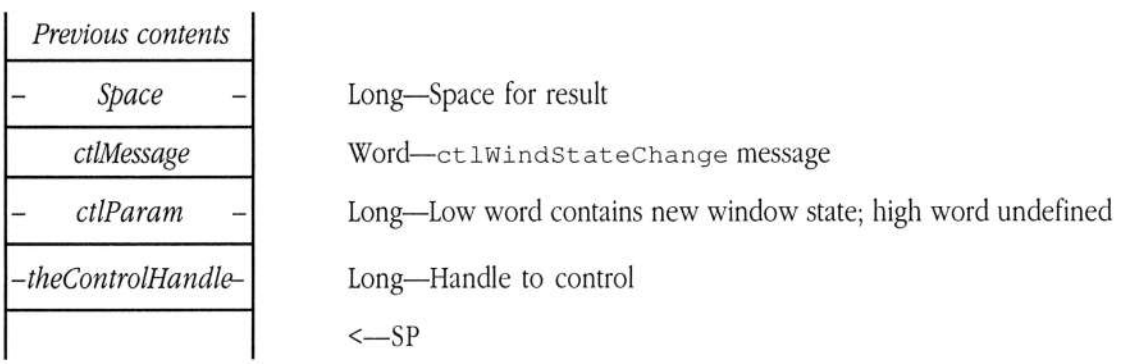
The low-order word of the *ctlParam* parameter contains the new state of the window.

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

The high-order word is undefined.

Parameter

Stack before call



Stack after call

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 *ctlParam* 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

Previous contents	
– *Space* –	Long—Space for result from control definition procedure
– *ctlHandle* –	Long—Handle of control to be called
ctlMessage	Word—Control message to send to control definition procedure
– *ctlParam* –	Long—Parameter to pass to control definition procedure
	<—SP

Stack after call

Previous contents	
– *Result* –	Long—Result value from control definition procedure
	<—SP

Errors None

C

```
extern pascal Long CallCtlDefProc(ctlHandle,
          ctlMessage, ctlParam);

Handle    ctlHandle;
Word      ctlMessage;
Long      ctlParam;
```

CMLoadResource $3210

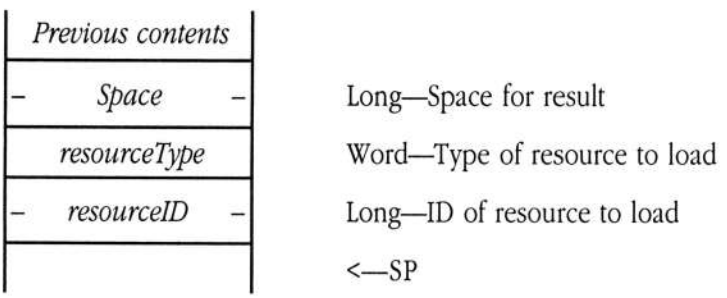
This is an entry point to the internal Control Manager routine that loads resources. You specify the resource type and ID of the resource to be loaded. See Chapter 45, "Resource Manager," for more information on resources.

Any errors during resource load result in system death.

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

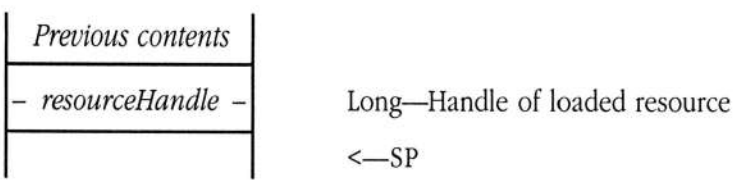
Parameters

Stack before call



Long—Space for result
Word—Type of resource to load
Long—ID of resource to load
<—SP

Stack after call



Long—Handle of loaded resource
<—SP

Errors None

C

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

Word      resourceType;
Long      resourceID;
```

CMReleaseResource $3310

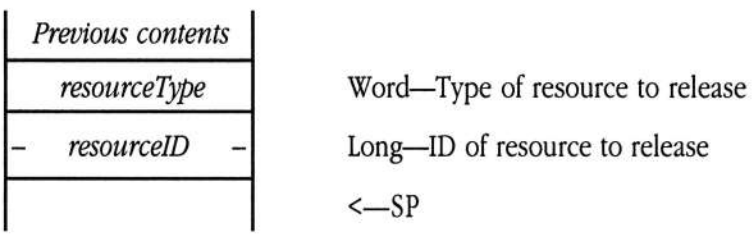
This is an entry point to the internal Control Manager routine that releases resources. You specify the resource type and ID of the resource to be released. The resource is released by marking it purgeable. See Chapter 45, "Resource Manager," for more information on resources.

Any errors result in system death.

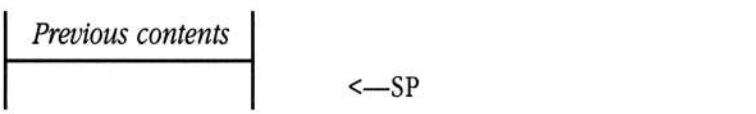
▲ **Warning** Applications must never issue this call. ▲

Parameters

Stack before call



Stack after call



Errors None

C

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

Word        resourceType;
Long        resourceID;
```

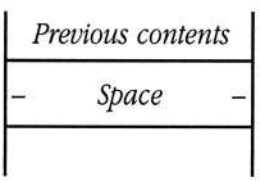
FindTargetCtl $2610

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 an undefined value.

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

Parameters

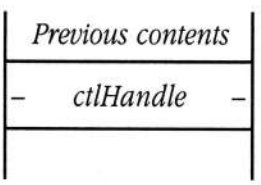
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Handle of target control; undefined if none or error

<—SP

Errors

$1004	`noCtlError`	No controls in window.
$1005	`noExtendedCtlError`	No extended controls in window.
$1006	`noCtlTargetError`	No target extended control.
$100C	`noFrontWindowError`	There is no front window.

C

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

GetCtlHandleFromID $3010

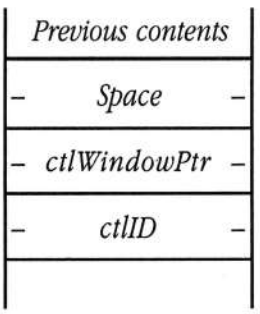
Retrieves the handle to the control record for a control with a specified `ctlID` field value. The `ctlID` field is an application-defined tag for a control. Set the `ctlID` field with the `SetCtlID` or `NewControl2` tool call; read the contents of the `ctlID` field with `GetCtlID`.

If an error occurs, the returned handle is undefined.

This call is valid only for extended controls.

Parameters

Stack before call



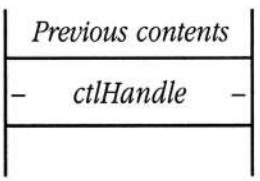
Long—Space for result

Long—Pointer to window for control list search; NIL = top window

Long—ID value for desired control

<—SP

Stack after call



Long—Handle for specified control

<—SP

Errors

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

C

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

Pointer   ctlWindowPtr;
Long      ctlID;
```

GetCtlID $2A10

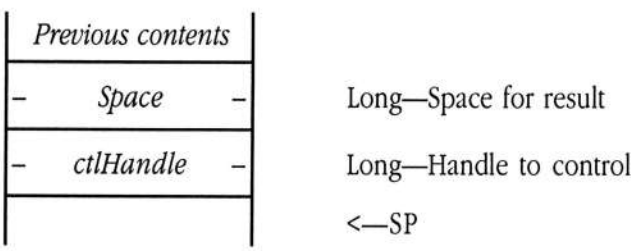
Returns the `ctlID` field from the control record of a specified control. The `ctlID` field is an application-defined tag for a control. Your application can use this field in many ways. For example, since the value of `ctlID` is known at compile time, you can construct efficient routing code for handling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

Use the `SetCtlID` or `NewControl2` Control Manager tool call to set the `ctlID` field.

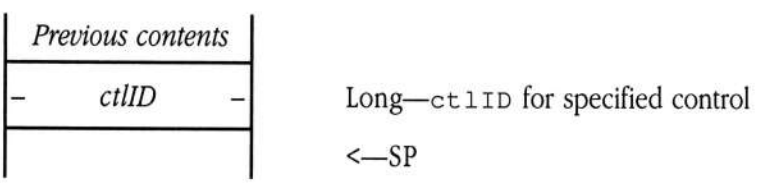
If the specified control is not an extended control, the resulting ID is undefined, and an error is returned.

Parameters

Stack before call



Stack after call



Errors

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

C

```
extern pascal Long GetCtlID(ctlHandle);

Handle    ctlHandle;
```

GetCtlMoreFlags $2E10

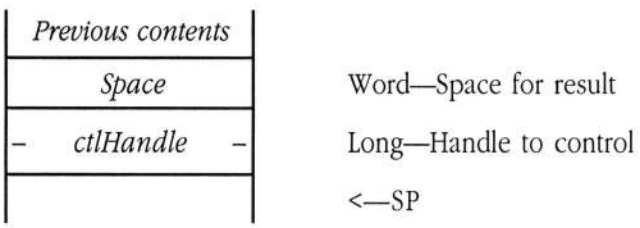
Gets the contents of the `ctlMoreFlags` field of the control record for a specified control. The `ctlMoreFlags` field contains flags governing target status, event processing, and other aspects of the control.

Use the `SetCtlMoreFlags` or `NewControl2` Control Manager tool call to set the `ctlMoreFlags` field.

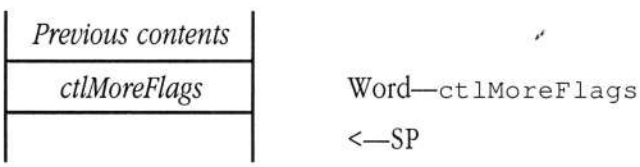
If the specified control is not an extended control, the result is undefined, and an error is returned.

Parameters

Stack before call



Stack after call



Errors

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

C

```
extern pascal Word GetCtlMoreFlags(ctlHandle);

Handle    ctlHandle;
```

GetCtlParamPtr $3510

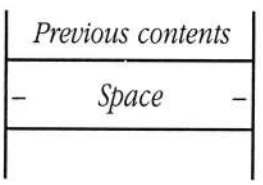
Retrieves the pointer to the current text substitution array for the Control Manager. This array contains the information used for text substitution in static text controls (see "Static Text Control" earlier in this chapter for details).

Set the contents of this field with the `SetCtlParamPtr` or `NewControl2` Control Manager tool call.

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

Parameters

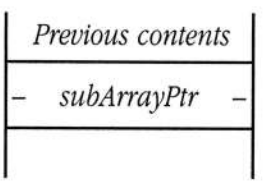
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to text substitution array

<—SP

Errors None

C

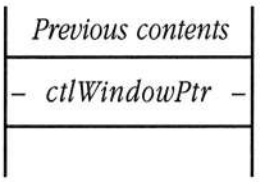
```
extern pascal Pointer GetCtlParamPtr();
```

InvalCtls $3710

Invalidates all rectangles for all controls in a specified window.

Parameters

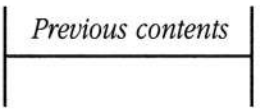
Stack before call



Long—Pointer to window for operation

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void InvalCtls(ctlWindowPtr);

Pointer   ctlWindowPtr;
```

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 set to 1 in the `ctlMoreFlags` field of the control record). It then clears the target flag for this control, searches the control list for the next control 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. If no new target control is found, the Control Manager returns the handle of the current target control.

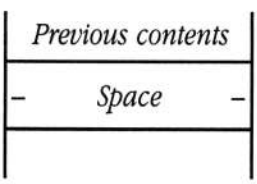
Both affected controls (the old and new target) receive `ctlChangeTarget` messages from the Control Manager.

If an error occurs, the returned handle is undefined.

This call is valid only for extended controls.

Parameters

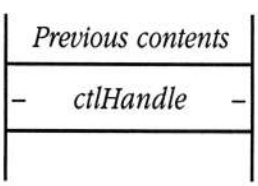
Stack before call



Long—Space for result (handle)

<—SP

Stack after call



Long—Handle of new target control; undefined if error

<—SP

Errors

$1004	`noCtlError`	No controls in window.
$1005	`noExtendedCtlError`	No extended controls in window.
$100B	`noCtlToBeTargetError`	No control could be made the target.

C

```
extern pascal Handle MakeNextCtlTarget();
```

MakeThisCtlTarget $2810

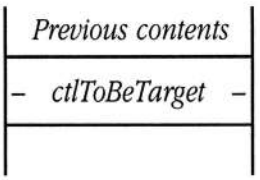
Makes the specified control the target. You specify the control that is to become the target control by passing its handle to this routine. This call will work for both active and inactive windows.

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

This call is valid only for extended controls.

Parameters

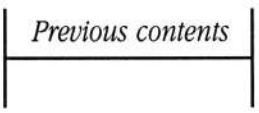
Stack before call



Long—Handle to control to be made the target

<—SP

Stack after call



<—SP

Errors

$1007	`notExtendedCtlError`	Action valid only for extended controls.
$1008	`canNotBeTargetError`	Specified control cannot be made the 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. 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)
inputDesc	Word—Describes contents of *inputRef*
– *inputRef* –	Long—Reference of a type defined by *inputDesc*
	<—SP

Stack after call

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

Errors None

C

```
extern pascal Handle NewControl2(ownerPtr,
          inputDesc, inputRef);

Pointer   ownerPtr;
Word      inputDesc;
Long      inputRef;
```

inputDesc Defines the contents and type of item referenced by *inputRef*. Possible values for *inputDesc* are

`singlePtr`	0	*inputRef* is a pointer to a single-item template.
`singleHandle`	1	*inputRef* is a handle for a single-item template.
`singleResource`	2	*inputRef* is a resource ID of a single-item template (resource type of `rControlTemplate`, $8004).
`ptrToPtr`	3	*inputRef* is a pointer to a list of pointers to item templates.
`ptrToHandle`	4	*inputRef* is a pointer to a list of handles for item templates.
`ptrToResource`	5	*inputRef* is a pointer to a list of resource IDs of item templates (resource type of `rControlTemplate`, $8004).
`handleToPtr`	6	*inputRef* is a handle to a list of pointers to item templates.
`handleToHandle`	7	*inputRef* is a handle to a list of handles for item templates.
`handleToResource`	8	*inputRef* is a handle to a list of resource IDs of item templates (resource type of `rControlTemplate`, $8004).
`resourceToResource`	9	*inputRef* is a resource ID of a list of resource IDs of item templates (the list reference is a resource of type `rControlList`, $8003; each entry in that list is a resource of type `rControlTemplate`, $8004).

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

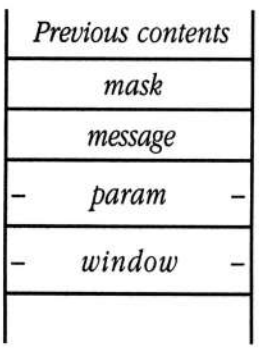
NotifyCtls $2D10

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* parameter. 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 (the system performs a bitwise AND operation with *mask* and `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 definition procedure messages).

Parameters

Stack before call



Word—Bit mask to be compared with `ctlMoreFlags`

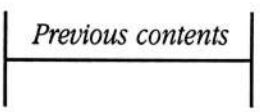
Word—Control message to send to control definition procedures

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 value indicating whether the event was fielded by a control and returns the handle of the control that serviced the event. That handle is returned in `taskData2` of the task record for the event.

The *targetOnlyFlag* parameter governs how 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, the 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, the 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, the result is FALSE and `taskData2` is undefined. Otherwise, the result is TRUE and `taskData2` contains the handle of the accepting control.

This call is valid only for extended controls.

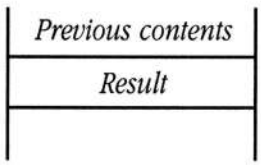
◆ *Note:* If a control can be made the target (`fCtlCanBeTarget` is set to 1 in `ctlMoreFlags` of its control record), then the Control Manager sends events to that control regardless of the setting of the `fCtlWantEvents` bit.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result Boolean
targetOnlyFlag	Word—(Boolean) TRUE = send to target only; FALSE = all controls
– *ctlWindowPtr* –	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	`noExtendedCtlError`	No extended controls in window.
$100C	`noFrontWindowError`	There is no front window.

C

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

Word      targetOnlyFlag;
Pointer   ctlWindowPtr, eTaskRecPtr;
```

SetCtlID $2B10

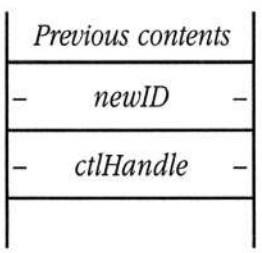
Sets the `ctlID` field in the control record of a specified control. The `ctlID` field is an application-defined tag for a control. Your application can use this field in many ways. For example, since the value of `ctlID` is known at compile time, you can construct efficient routing code for handling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

Use the `GetCtlID` Control Manager call to retrieve the contents of this field.

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

Parameters

Stack before call

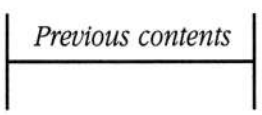


Long—New `ctlID` value for the control

Long—Handle to control

<—SP

Stack after call



<—SP

Errors

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

C

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

Long      newID;
Handle    ctlHandle;
```

SetCtlMoreFlags $2F10

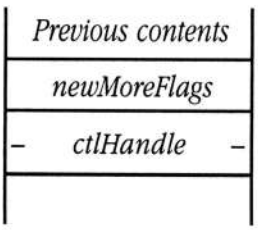
Sets the contents of the `ctlMoreFlags` field of the control record for a specified control. The `ctlMoreFlags` field contains flags governing target status, event processing, and other aspects of the control.

Use the `GetCtlMoreFlags` Control Manager call to retrieve the contents of this field.

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

Parameters

Stack before call

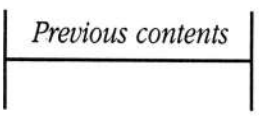


Word—New `ctlMoreFlags` value for the control

Long—Handle to control

<—SP

Stack after call



<—SP

Errors

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

C

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

Word      newMoreFlags;
Handle    ctlHandle;
```

SetCtlParamPtr $3410

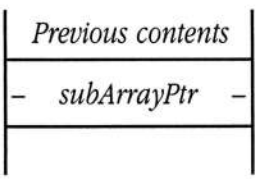
Sets the pointer to the current text substitution array for the Control Manager. This array contains the information used for text substitution in static text controls (see "Static Text Control" earlier in this chapter).

Use the `GetCtlParamPtr` Control Manager tool call to retrieve the contents of this field.

- *Note:* This pointer is global to the Control Manager; it is not associated with a specific control. For this reason, when using this feature with desk accessories be sure to save and restore the previous contents of the field.

Parameters

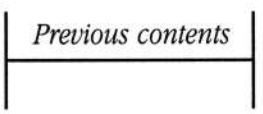
Stack before call



Long—New pointer to text substitution array

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void SetCtlParamPtr(subArrayPtr);

Pointer   subArrayPtr;
```

Control Manager error codes

Table 28-1 lists the error codes that may be returned by Control Manager calls.

■ **Table 28-1** 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	noExtendedCtlError	No extended controls in window
$1006	noCtlTargetError	No target extended control
$1007	notExtendedCtlError	Action valid only for extended controls
$1008	canNotBeTargetError	Specified control cannot be made the target
$1009	noSuchIDError	The specified ID cannot be found
$100A	tooFewParmsError	Too few parameters specified
$100B	noCtlToBeTargetError	No control could be made the target
$100C	noFrontWindowError	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. One of the header fields is a parameter count, allowing extensible support for templates of variable length. The value of the parameter count field tells the Control Manager how many parameters to use, making optional template fields possible.

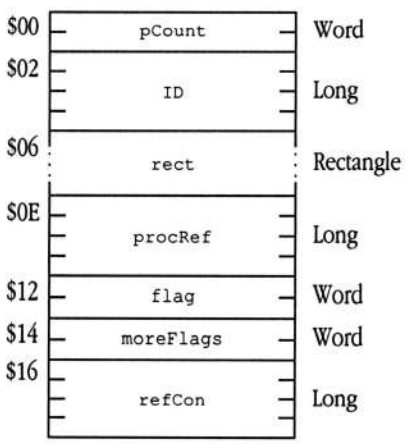
The following sections define the item templates for each control type. Field names marked with an asterisk (*) represent optional fields.

Control template standard header

Each control template contains the standard header, which consists of seven fields. Following that header, some templates have additional fields that further define the control to be created. The format and content of the standard template header are 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 according to the type of control template.

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

`rect` Field that 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 (or a resource ID for) a control definition procedure or the ID of a standard routine. If the `fCtlProcRefNotPtr` flag in the `moreFlags` field is set to 0, then `procRef` must contain a pointer. If the flag is set to 1, then the Control Manager checks the low-order three bytes of `procRef`. If these bytes are all zero, then `procRef` must contain the ID for a standard routine; if these bytes are nonzero, `procRef` contains the resource ID for a control 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 menu
`radioControl`	$84000000	Radio button
`scrollBarControl`	$86000000	Scroll bar
`growControl`	$88000000	Size box
`statTextControl`	$81000000	Static text
`editTextControl`	$85000000	TextEdit

◆ *Note:* The `procRef` value for `iconButtonControl` is not truly a standard value. Rather, it is the resource ID for the standard control definition procedure for icon buttons.

`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`, and the low-order byte contains `ctlFlag`. The bits in `flag` are mapped as follows:

Highlight	bits15–8	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 6–0	Values and meaning depend on 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 this flag is set to 1, this control is currently the target of any typing or editing commands.
`fCtlCanBeTarget`	$4000	If this flag is set to 1, then this control can be made the target control.
`fCtlWantEvents`	$2000	If this flag is 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 receives events sent to it regardless of setting of this flag.
`fCtlProcRefNotPtr`	$1000	If this flag is set to 1, then the Control Manager expects `procRef` to contain the ID or resource ID of a control procedure. If it is set to 0, then `procRef` contains a pointer to a custom control procedure.
`fCtlTellAboutSize`	$0800	If this flag is 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 this flag is set to 1, then this is a multipart control. This flag allows control definition procedures to manage multipart controls (necessary since the Control Manager does not know about all the parts of a multipart 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):

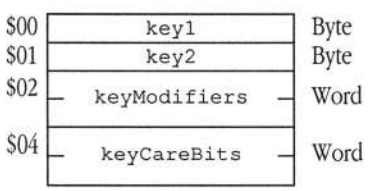
`titleIsPtr`	$00	Title reference is by pointer.
`titleIsHandle`	$01	Title reference is by handle.
`titleIsResource`	$02	Title reference is by resource ID (resource type corresponds to string type).
`colorTableIsPtr`	$00	Color table reference is by pointer.
`colorTableIsHandle`	$04	Color table reference is by handle.
`colorTableIsResource`	$08	Color table reference is by resource ID (resource type is `rCtlColorTbl`, $800D).

`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 uppercase or lowercase key equivalent.

`key2` This is the ASCII code for the lowercase or uppercase 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.

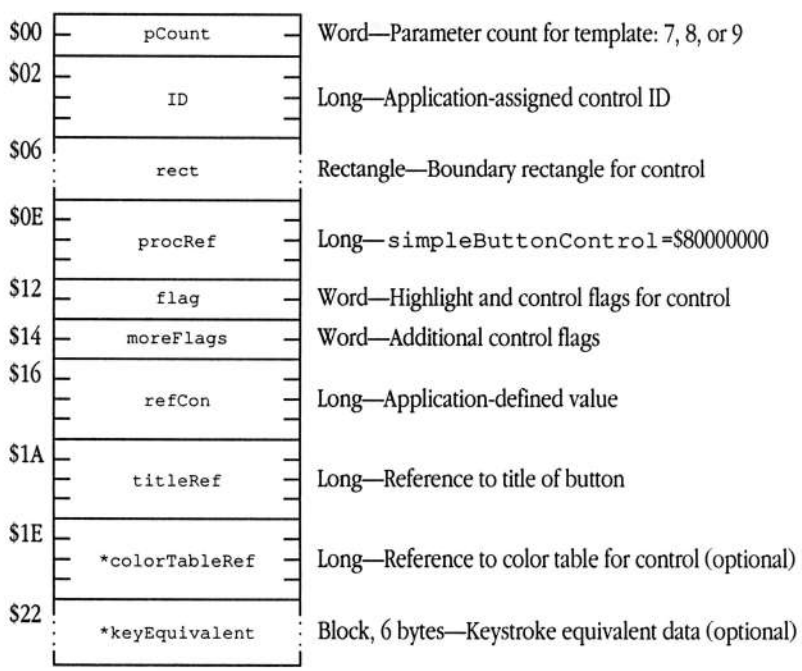
`keyModifiers` These modifiers must be set to 1 if the equivalence test is to pass. The format of this flag word corresponds to that defined for the event record in Chapter 7, "Event Manager," in Volume 1 of the *Toolbox Reference*. Note that only the modifiers in the high-order byte are used here.

`keyCareBits` These modifiers 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, set the following three bits to 1: the `controlKey` bit in `keyModifiers` and both the `optionKey` and the `controlKey` bits in `keyCareBits`. If you want Return and Enter to be treated the same, set the `keyPad` bit 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



Defined bits for `flag` are

Reserved	bits 15–8	Must be set to 0.
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–2	Must be set to 0.
Button type	bits 1–0	Describes 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, drop-shadowed button

Defined bits for `moreFlags` are

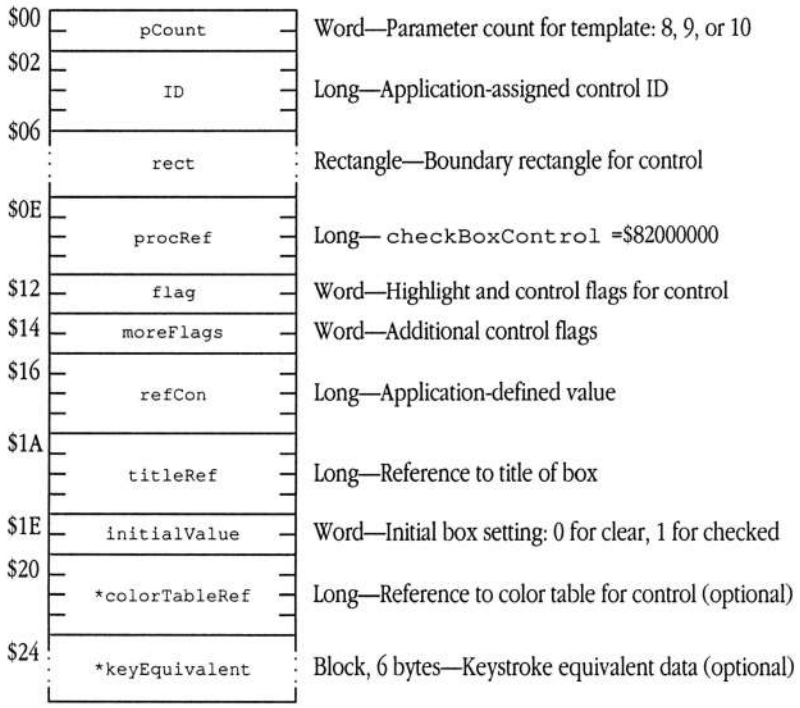
`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Set to 1 if button has keystroke equivalent.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–4	Must be set to 0.
Color table reference	bits 3–2	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 by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Title reference	bits 1–0	Defines type of title reference in `titleRef`. 00 = Title reference is by pointer 01 = Title reference is by handle 10 = Title reference is by resource ID (resource type corresponds to string type) 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



Defined bits for `flag` are

Reserved	bits 15–8	Must be set to 0.
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–0	Must be set to 0.

Defined bits for `moreFlags` are

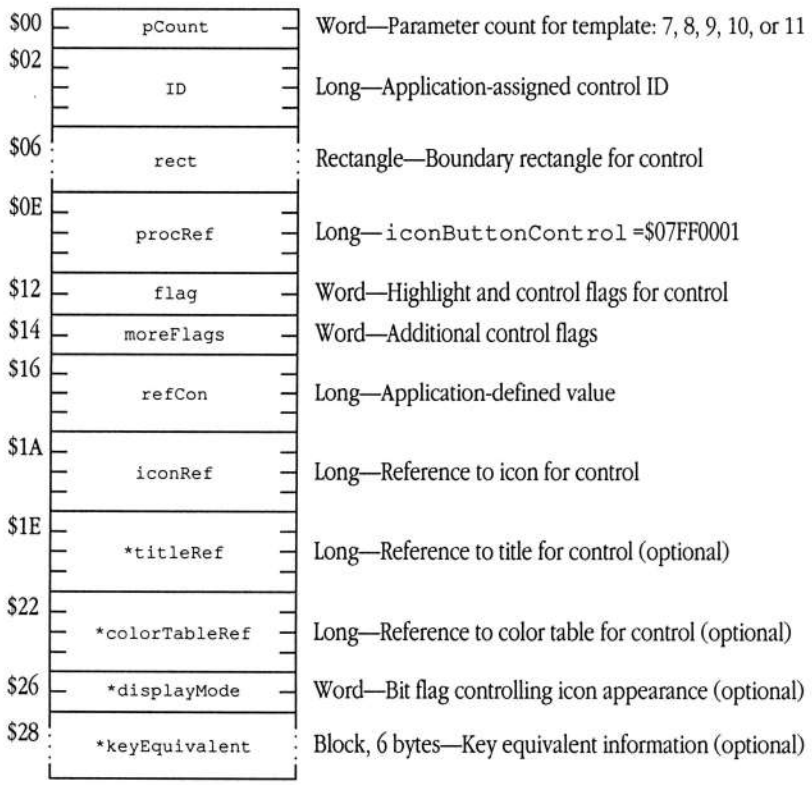
`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Set to 1 if check box has keystroke equivalent.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–4	Must be set to 0.
Color table reference	bits 3–2	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 by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Title reference	bits 1–0	Defines type of title reference in `titleRef`. 00 = Title reference is by pointer 01 = Title reference is by handle 10 = Title reference is by resource ID (resource type corresponds to string type) 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" earlier in this chapter.

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



Defined bits for `flag` are

`ctlHilite`	bits 15–8	Sets the `ctlHilite` field of the control record.
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–3	Must be set to 0.
`showBorder`	bit 2	0 = Show border, 1 = No border.
`buttonType`	bits 1–0	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

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Must be set to 0.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–6	Must be set to 0.
Icon reference	bits 5–4	Defines type of icon reference in `iconRef`. 00 = Icon reference is by pointer 01 = Icon reference is by handle 10 = Icon reference is by resource ID (resource type of `rIcon`, $8001) 11 = Invalid value
Color table reference	bits 3–2	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 by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value

Title reference	bits 1–0	Defines type of title reference in `titleRef`. 00 = Title reference is by pointer 01 = Title reference is by handle 10 = Title reference is by resource ID (resource type of `rPString`, $8006) 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 bits 0 and 1 of `moreFlags` to 0, and set this field to 0.

`displayMode` Passed directly to the `DrawIcon` routine, this field 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 15–12	Defines the background color to apply to the black part of black-and-white icons.
Foreground color	bits 11–8	Defines the foreground color to apply to the white part of black-and-white icons.
Reserved	bits 7–3	Must be set to 0.
`offLine`	bit 2	0 = Don't perform the AND operation on the image. 1 = Perform logical AND operation with light-gray pattern and image being copied.
`openIcon`	bit 1	0 = Don't copy light-gray pattern. 1 = Copy light-gray pattern instead of image.
`selectedIcon`	bit 0	0 = Don't invert image. 1 = Invert image before copying.

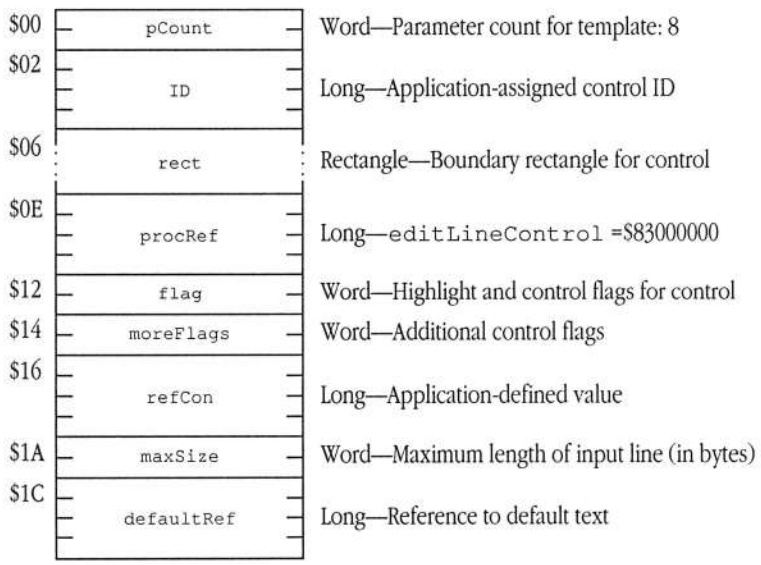
Color values (both foreground and background) are indexes into the current color table. See Chapter 16, "QuickDraw II," in Volume 2 of the *Toolbox Reference* for details about the format and content of these color tables.

`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" earlier in this chapter.

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



Defined bits for `flag` are

Reserved	bits 15–8	Must be set to 0.
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–0	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.
`fCtlWantEvents`	bit 13	Must be set to 1.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–2	Must be set to 0.
Text reference	bits 1–0	Defines type of text reference in `defaultRef`. 00 = Text reference is by pointer 01 = Text reference is by handle 10 = Text reference is by resource ID (resource type of `rPString`, $8006) 11 = Invalid value

`maxSize` Specifies the maximum number of characters allowed in the LineEdit field. Valid values are in the range 1 to 255, inclusive.

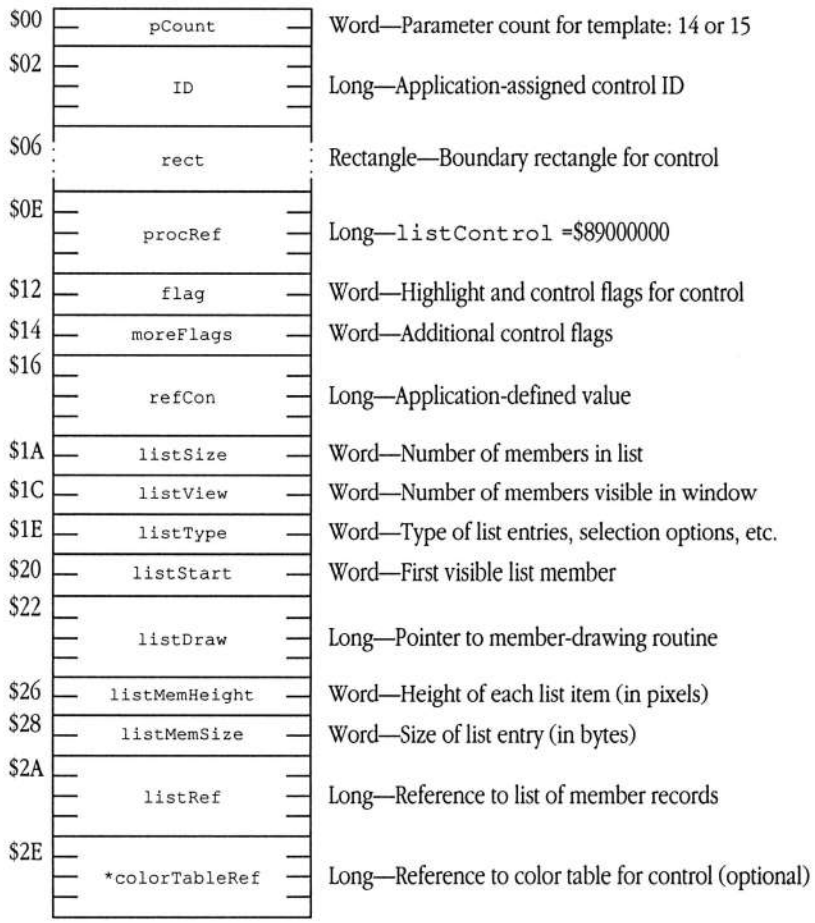
The high-order bit indicates whether the LineEdit field is a password field. Password fields protect user input by echoing asterisks or any application-defined character, rather than the actual user input. If this bit is set to 1, then the LineEdit field is a password field.

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" earlier in this chapter.

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



Defined bits for `flag` are

Reserved	bits 15–8	Must be set to 0.
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–0	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.
`fCtlWantEvents`	bit 13	Must be set to 0.
`fCtlProcRefNotPtr`	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 9–4	Must be set to 0.
Color table reference	bits 3–2	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 by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
List reference	bits 1–0	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 by pointer 01 = List reference is by handle 10 = List reference is by resource ID (resource type of `rListRef`, $801C) 11 = Invalid value

`listType` Valid values for `listType` are

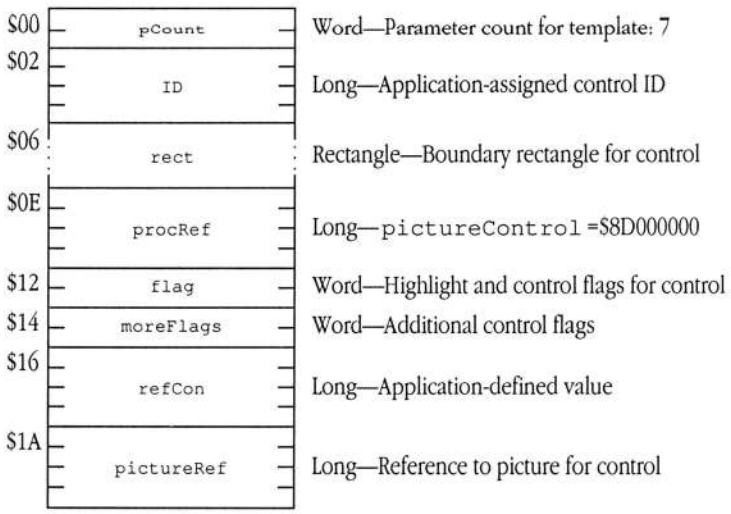
Reserved	bits 15–3	Must be set to 0.
`fListScrollBar`	bit 2	Allows you to control where the scroll bar for the list is drawn. 0 = Scroll bar drawn on outside of boundary rectangle 1 = Scroll bar drawn on inside of boundary rectangle (The List Manager calculates space needed, adjusts dimensions of boundary rectangle, and resets this flag.)
`fListSelect`	bit 1	Controls type of selection options available to the user. 0 = Arbitrary and range selection allowed 1 = Only single selection allowed
`fListString`	bit 0	Defines the type of strings used to define list items. 0 = Pascal strings 1 = Cstrings ($00-terminated)

For details on the remaining custom fields in this template, see the discussion under "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" earlier in this chapter.

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



Defined bits for `flag` are

`ctlHilite`	bits 15–8	Specifies whether the control wants to receive mouse selection events. The values for `ctlHilite` are 0 = Control is active 255 = Control is inactive
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–0	Must be set to 0.

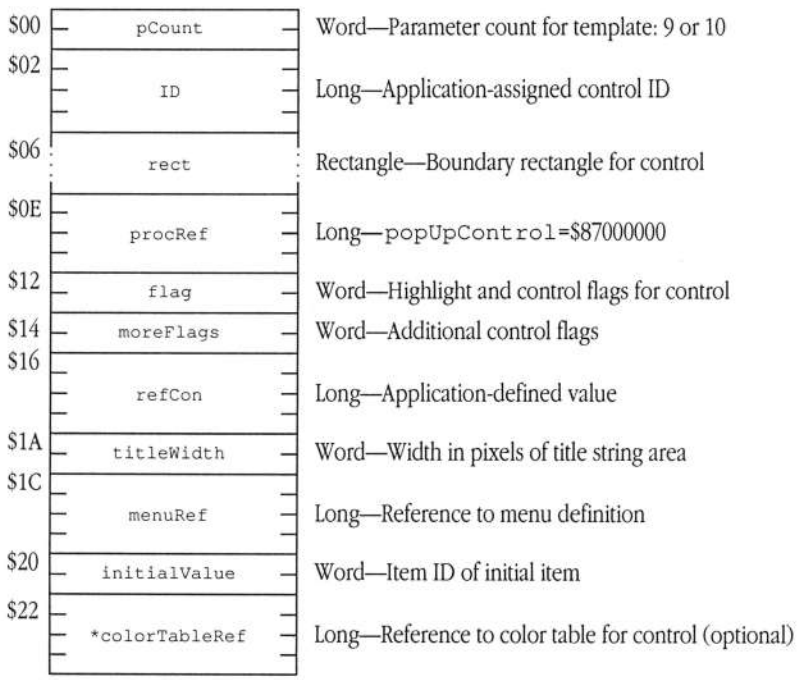
Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Must be set to 0.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–2	Must be set to 0.
Picture reference	bits 1–0	Defines type of picture reference in `pictureRef`. 00 = Invalid value 01 = Reference is by handle 10 = Reference is by resource ID (resource type of `rPicture`, $8002) 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" earlier in this chapter.

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



Defined bits for `flag` are

`ctlHilite`	bits 15–8	Specifies whether the control wants to receive mouse selection events. The values for `ctlHilite` are 0 = Control is active 255 = Control is inactive
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
`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). 0 = Draw normal pop-up menu 1 = Draw pop-up menu with white space (type 2)
`fDontHiliteTitle`	bit 5	Controls highlighting of the menu title. 0 = Highlight title 1 = Do not 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. 0 = Draw the title 1 = Do not draw the title
`fDontDrawResult`	bit 3	Allows you to control whether the selection is drawn in the pop-up rectangle. 0 = Draw the result 1 = Do not draw the result in the result area after a selection
`fInWindowOnly`	bit 2	Controls how much the pop-up menu can expand; this is particularly relevant to type 2 pop-up menus (see Chapter 37, "Menu Manager Update," for details on type 2 pop-up menus). 0 = Allow the pop-up menu to expand to the size of the screen 1 = Keep the pop-up menu in the current window

`fRightJustifyTitle`	bit 1	Controls title justification. 0 = Left-justify the title 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) and the value for `titleWidth` is also adjusted
`fRightJustifyResult`	bit 0	Controls result justification. 0 = Left-justify the selection `titleWidth` pixels from the left of the pop-up rectangle 1 = Right-justify the selection

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Must be set to 1 if the pop-up menu has any keystroke equivalents defined.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–5	Must be set to 0.
Color table reference	bits 4–3	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 by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
`fMenuDefIsText`	bit 2	Defines type of data referred to by `menuRef`. 0 = `menuRef` is a reference to a menu template (See Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference* for details on format and content of a menu template.) 1 = `menuRef` is a pointer to a text stream in `NewMenu` format (Again, see Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference* for details.)

Menu reference	bits 1–0	Defines type of menu reference in `menuRef` (if `fMenuDefIsText` is set to 1, then these bits are ignored). 00 = Menu reference is by pointer 01 = Menu reference is by handle 10 = Menu reference is by resource ID (resource type of `rMenu`, $8009) 11 = Invalid value

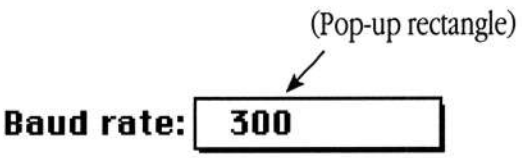
`rect` Defines the boundary rectangle for the pop-up menu and its title, before the menu has been selected by the user. The Menu Manager calculates the lower-right coordinates of the rectangle for you if you specify those coordinates as (0,0).

`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, you can align them vertically by giving all menus the same x_1 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 according to 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.

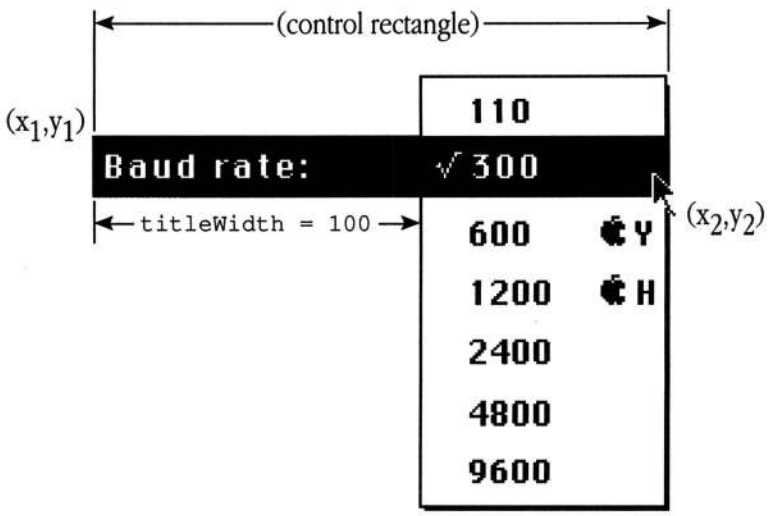
`menuRef` Reference to menu definition (see Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference* and Chapter 37, "Menu Manager Update," in this book for details on menu templates). The type of reference contained in `menuRef` is defined by the menu reference bits in `moreFlags`.

`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 unselected 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, no item is displayed in the pop-up rectangle.

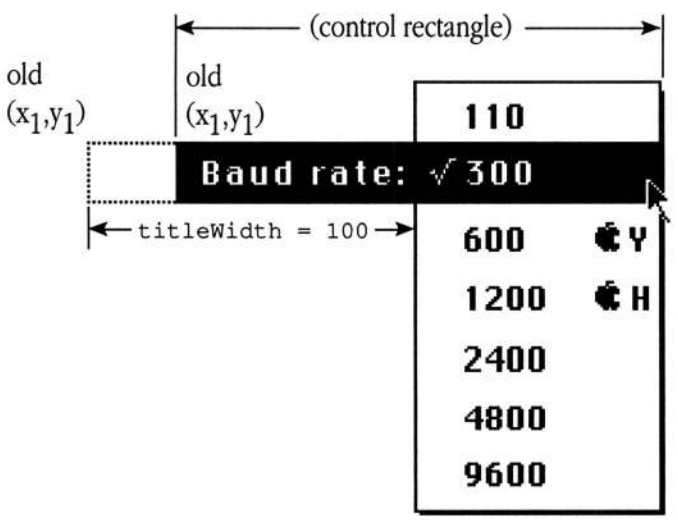
■ **Figure 28-10** Unselected pop-up menu



■ **Figure 28-11** Selected pop-up menu with left-justified title



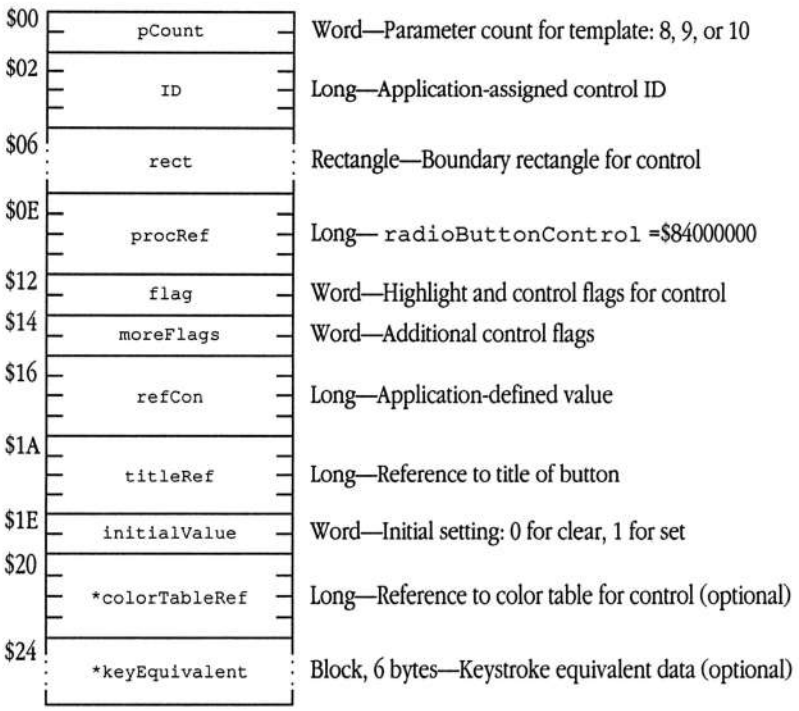
■ **Figure 28-12** Selected pop-up menu with right-justified title



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



Defined bits for `flag` are

Reserved	bits 15–8	Must be set to 0.
`ctlInvis`	bit 7	0=Visible, 1=Invisible.
Family number	bits 6–0	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

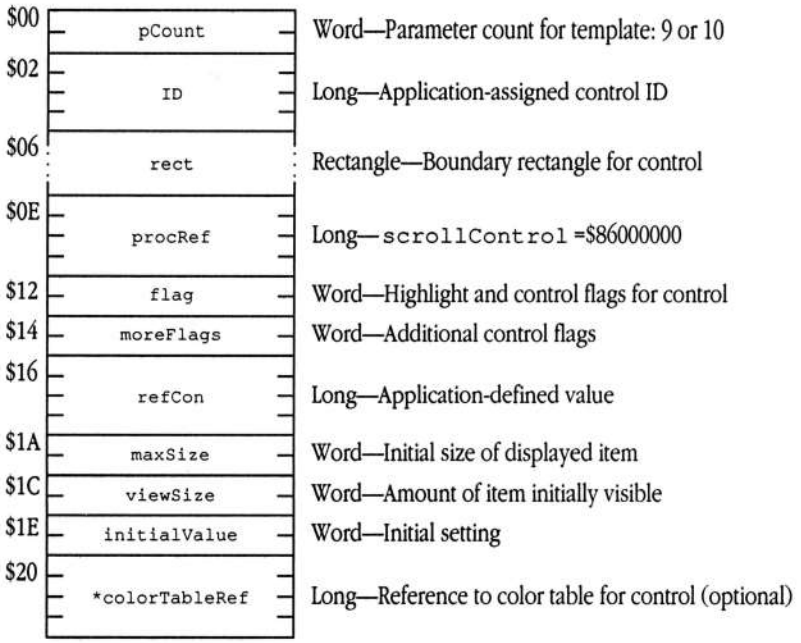
`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Set to 1 if button has keystroke equivalent.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–4	Must be set to 0.
Color table reference	bits 3–2	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 by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Title reference	bits 1–0	Defines type of title reference in `titleRef`. 00 = Title reference is by pointer 01 = Title reference is by handle 10 = Title reference is by resource ID (resource type corresponds to string type) 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



Defined bits for `flag` are

Reserved	bits 15–8	Must be set to 0.
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–5	Must be set to 0.
`horScroll`	bit 4	0 = Vertical scroll bar, 1 = Horizontal scroll bar.
`rightFlag`	bit 3	0 = Bar has no right arrow, 1 = Bar has right arrow.
`leftFlag`	bit 2	0 = Bar has no left arrow, 1 = Bar has left arrow.
`downFlag`	bit 1	0 = Bar has no down arrow, 1 = Bar has down arrow.
`upFlag`	bit 0	0 = Bar has no up arrow, 1 = Bar has up arrow.

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

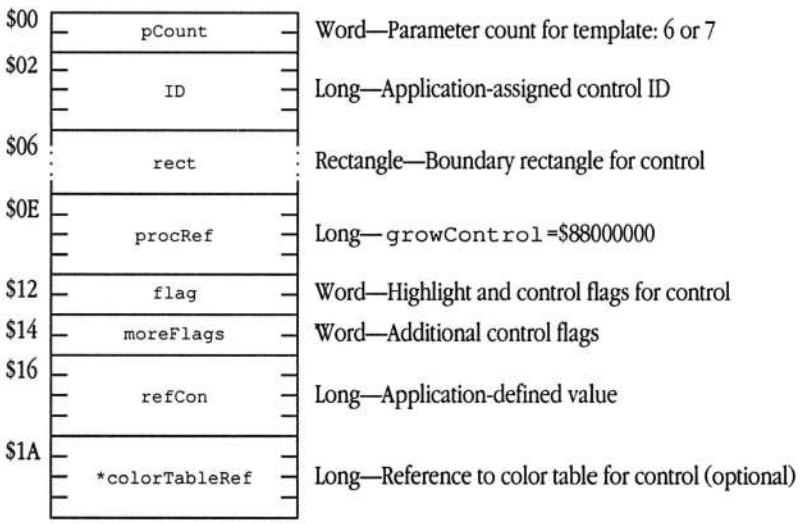
Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Must be set to 0.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–4	Must be set to 0.
Color table reference	bits 3–2	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 by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Reserved	bits 1–0	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



Defined bits for `flag` are

Reserved	bits 15–8	Must be set to 0.
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–1	Must be set to 0.
`fCallWindowMgr`	bit 0	0 = Just highlight control, 1 = Call `GrowWindow` and `SizeWindow` to track this control.

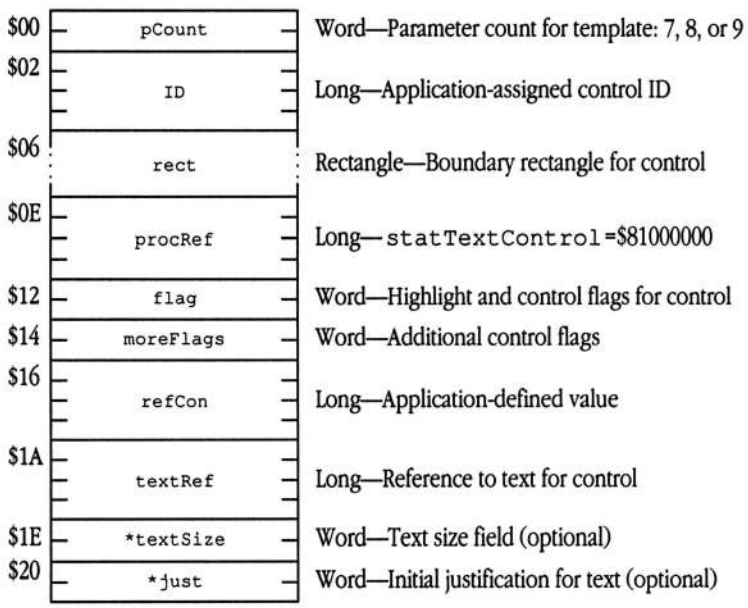
Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Must be set to 0.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–4	Must be set to 0.
Color table reference	bits 3–2	Defines type of reference in `colorTableRef` (see "Error Corrections" at the beginning of this chapter for the definition of the size box color table). 00 = Color table reference is by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Reserved	bits 1–0	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" earlier in this chapter.

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



Defined bits for `flag` are

Reserved	bits 15–8	Must be set to 0.
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–2	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.
`fCtlWantEvents`	bit 13	Must be set to 0.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–2	Must be set to 0.
Text reference	bits 1–0	Defines type of text reference in `textRef`. 00 = Text reference is by pointer 01 = Text reference is by handle 10 = Text reference is by resource ID (resource type of `rTextForLETextBox2`, $800B) 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 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

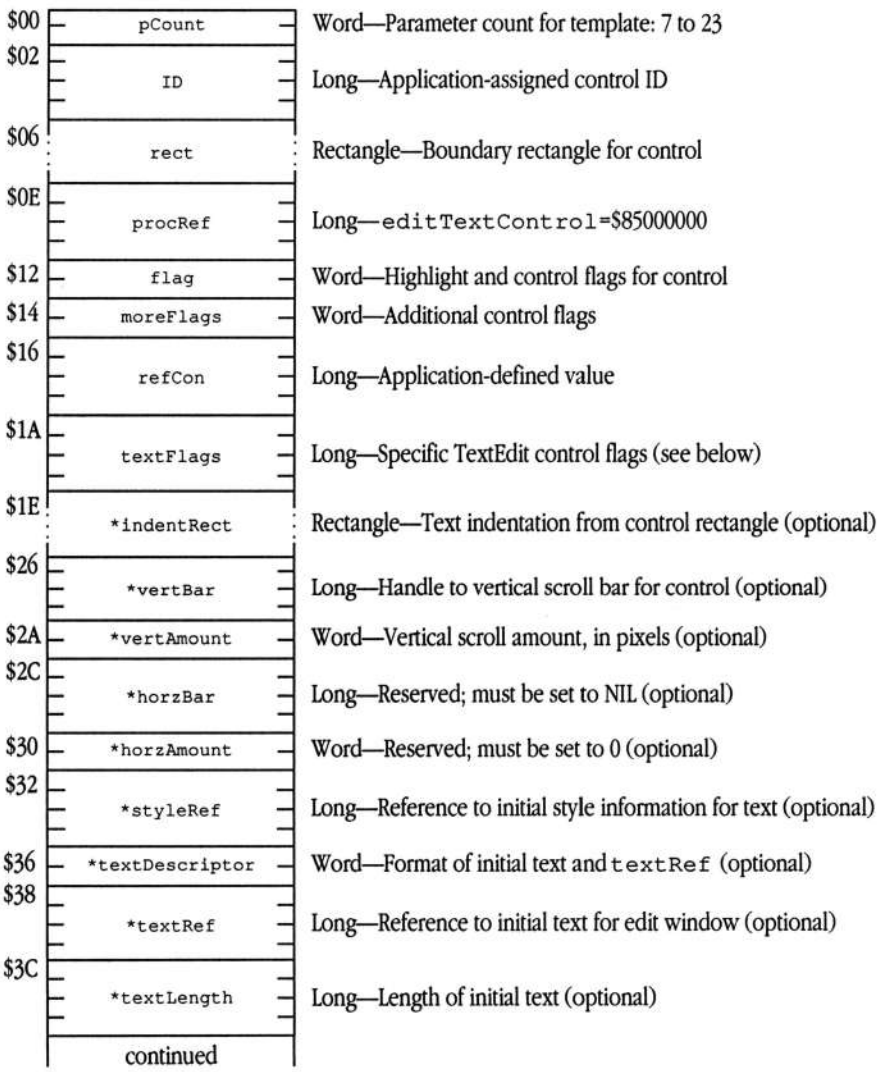
`leftJustify`	0	Text is left justified in the display window.
`centerJustify`	1	Text is centered in the display window.
`rightJustify`	–1	Text is right justified in the display window.
`fullJustify`	2	Text is fully justified (both left and right) in the display window.

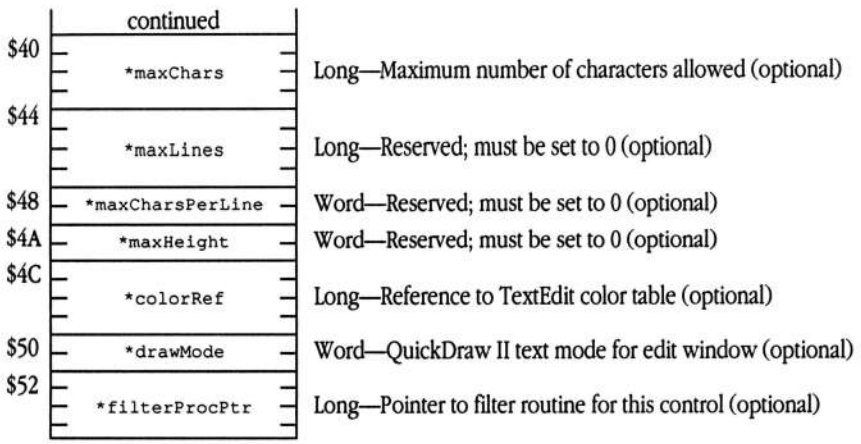
Static text controls do not support color tables. To display text of different color, you must embed the appropriate commands into the text string you are displaying. See the discussion of `LETextBox2` in Chapter 10, "LineEdit Tool Set," in Volume 1 of the *Toolbox Reference* for details on command format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

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





Defined bits for `flag` are

Reserved	bits 15–8	Must be set to 0.
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–0	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.
`fCtlWantEvents`	bit 13	Must be set to 1.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	If this bit is set to 1, a size box is created in the lower-right corner of the window. Whenever the control window is resized, the edit text is resized and redrawn.
`fCtlIsMultiPart`	bit 10	Must be set to 1.
Reserved	bits 9–4	Must be set to 0.
Color table reference	bits 3–2	Defines type of reference in `colorRef`; the color table for a TextEdit control (`TEColorTable`) is described in Chapter 49, "TextEdit Tool Set," in this book. 00 = Color table reference is by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value

Style reference	bits 1–0	Defines type of style reference in `styleRef`; the format for a TextEdit style descriptor is described in Chapter 49, "TextEdit Tool Set," in this book. 00 = Style reference is by pointer 01 = Style reference is by handle 10 = Style reference is by resource ID (resource type of `rStyleBlock`, $8012) 11 = Invalid value

△ **Important** Do not set `fCtlTellAboutSize` to 1 unless the text edit record also has a vertical scroll bar. This flag works only for TextEdit records that are controls. △

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. 0 = Do not restrict the number of styles in the text 1 = Allow only one style in the text
`fNoWordWrap`	bit 28	Allows you to control TextEdit word wrap behavior. 0 = Perform word wrap to fit the ruler 1 = Do not word wrap the text; break lines only on CR ($0D) characters
`fNoScroll`	bit 27	Controls user access to scrolling. 0 = Scrolling permitted 1 = Do not allow either manual or auto-scrolling
`fReadOnly`	bit 26	Restricts the text in the window to read-only operations (copying from the window will still be allowed). 0 = Editing permitted 1 = No editing allowed
`fSmartCutPaste`	bit 25	Controls TextEdit support for smart cut and paste (see Chapter 49, "TextEdit Tool Set," for details on smart cut and paste support). 0 = Do not use smart cut and paste 1 = Use smart cut and paste

`fTabSwitch`	bit 24	Defines behavior of the Tab key (see Chapter 49, "TextEdit Tool Set," for details). 0 = Tab inserted in TextEdit document 1 = Tab to next control in the window
`fDrawBounds`	bit 23	Tells TextEdit whether to draw a box around the edit window, just inside `rect`; the pen for this box is 2 pixels wide and 1 pixel high. 0 = Do not draw rectangle 1 = 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's resizing of the edit window; if this bit is set to 1, TextEdit automatically adjusts the right margin value for the ruler. 0 = Do not resize the ruler 1 = Resize the ruler
`fDisableSelection`	bit 20	Controls whether user can select text. 0 = User can select text 1 = User cannot select text
`fDrawInactiveSelection`	bit 19	Controls how inactive selected text is displayed. 0 = TextEdit does not display inactive selections 1 = TextEdit draws a box around inactive selections
Reserved	bits 18–0	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. 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. 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 Tool Set," 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 Tool Set," 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.

`maxCharsPerLine`
Must be set to NULL.

`maxHeight` Must be set to 0.

`colorRef` Reference to the color table for the text. This is a TextEdit color table (see Chapter 49, "TextEdit Tool Set," 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 Tool Set," 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
fMenuDefIsText          equ $0004
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 solid
                                    ;  black/white
            dc.W $00F0              ; inactive bar color
;
; Definition of a simple vertical scroll bar
;
theScroll   dc.W 10                 ; number of params
            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
            dc.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

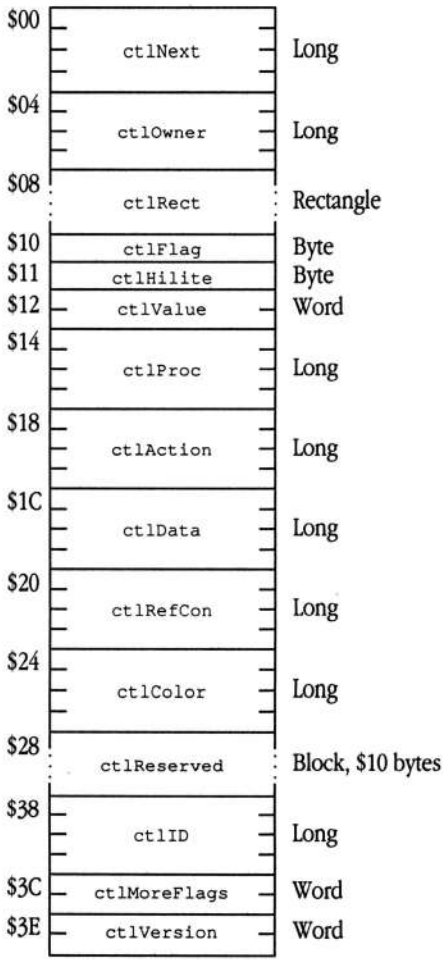
The `NewControl2` tool call creates extended control records (as discussed earlier in this chapter in "New and Changed Controls"). This section describes the format and content of the control records created by `NewControl2`.

▲ **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. Figure 28-18 shows the layout of the new generic control record.

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



`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 0 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 position and size of the control 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

0	Control active; no highlighted parts—this value causes 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 current setting of the control. For check boxes and radio buttons, a zero value indicates that the control is off, and a nonzero value indicates that 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 use by custom controls as appropriate.

`ctlProc` For standard controls, this field indicates the control type, identified by its ID or resource 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 or resource 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 menu
`radioControl`	$84000000	Radio button
`scrollBarControl`	$86000000	Scroll bar
`growControl`	$88000000	Size box
`statTextControl`	$81000000	Static text
`editTextControl`	$85000000	TextEdit

◆ *Note:* The `ctlProc` value for `iconButtonControl` is not truly a standard value, but rather the resource ID for the standard control definition procedure for icon buttons.

`ctlAction` Pointer to the custom action procedure for the control, if there is one. The `TrackControl` routine may call the custom action procedure in response to the user's dragging an icon 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 title.

`ctlRefCon` This field is reserved for application use.

`ctlColor` This field contains a reference to the color table to use when the control is drawn. If the field is set to NIL, the Control Manager uses a default color table defined by the control's definition procedure. Otherwise, `ctlColor` references which color table to use by a pointer, handle, or resource ID. 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 Control Manager uses the high-order byte to store its own control information. The control definition procedure uses the low-order byte to define reference types.

The defined Control Manager flags are

`fCtlTarget`	$8000	If this flag is set to 1, this control is currently the target of any typing or editing commands.
`fCtlCanBeTarget`	$4000	If this flag is set to 1, this control can be made the target control.
`fCtlWantEvents`	$2000	If this flag is 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 receives events sent to it regardless of the setting of this flag.

`fCtlProcRefNotPtr`	$1000	If this flag is set to 1, the Control Manager expects `ctlProc` to contain the ID or resource ID of a control procedure. If this flag is set to 0, `ctlProc` contains a pointer to a custom control procedure.
`fCtlTellAboutSize`	$0800	If this flag is set to 1, 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 this flag is set to 1, this is a multipart control. This flag allows control definition procedures to manage multipart controls (necessary since the Control Manager does not know about all the parts of a multipart control).

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

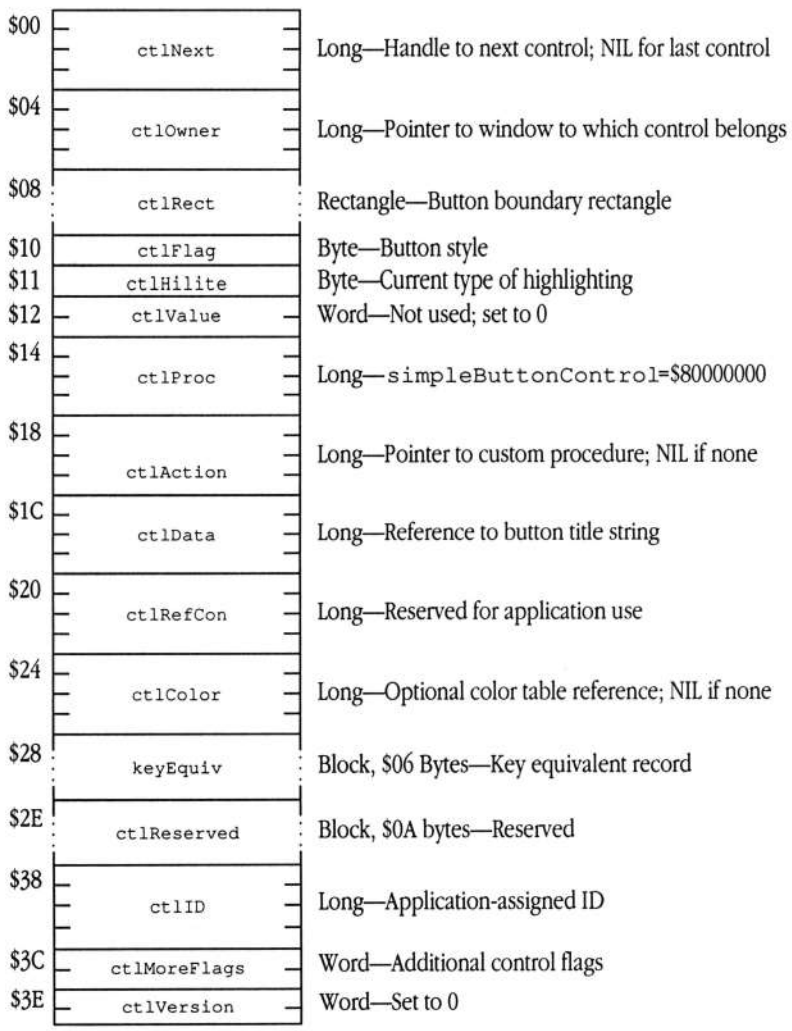
`titleIsPtr`	$00	Title reference is by pointer.
`titleIsHandle`	$01	Title reference is by handle.
`titleIsResource`	$02	Title reference is by resource ID (resource type corresponds to string type).
`colorTableIsPtr`	$00	Color table reference is by pointer.
`colorTableIsHandle`	$04	Color table reference is by handle.
`colorTableIsResource`	$08	Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D).

`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



Valid values for `ctlFlag` are

`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–2	Must be set to 0.
Button type	bits 1–0	Describes 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, drop-shadowed button

Valid values for `ctlMoreFlags` are

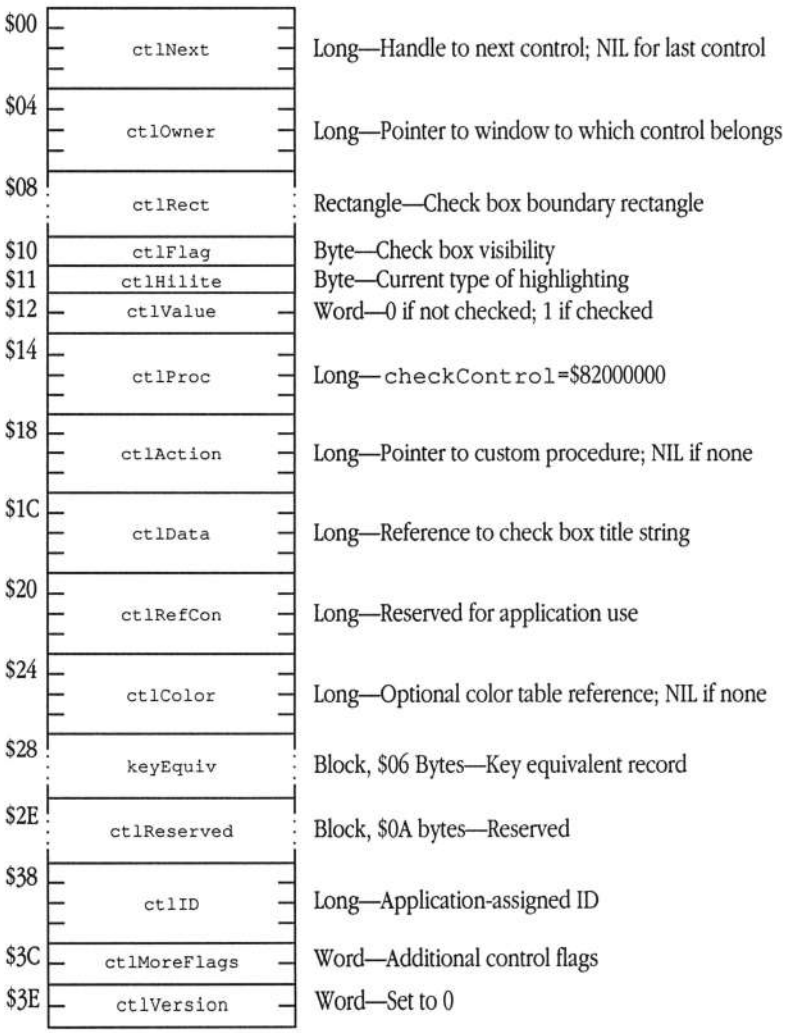
`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Set to 1 if button has keystroke equivalent.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–4	Must be set to 0.
Color table reference	bits 3–2	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 by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Title reference	bits 1–0	Defines type of title reference in `ctlData`. 00 = Title reference is by pointer 01 = Title reference is by handle 10 = Title reference is by resource ID (resource type corresponds to string type) 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



Valid values for `ctlFlag` are

`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–0	Must be set to 0.

Valid values for `ctlMoreFlags` are

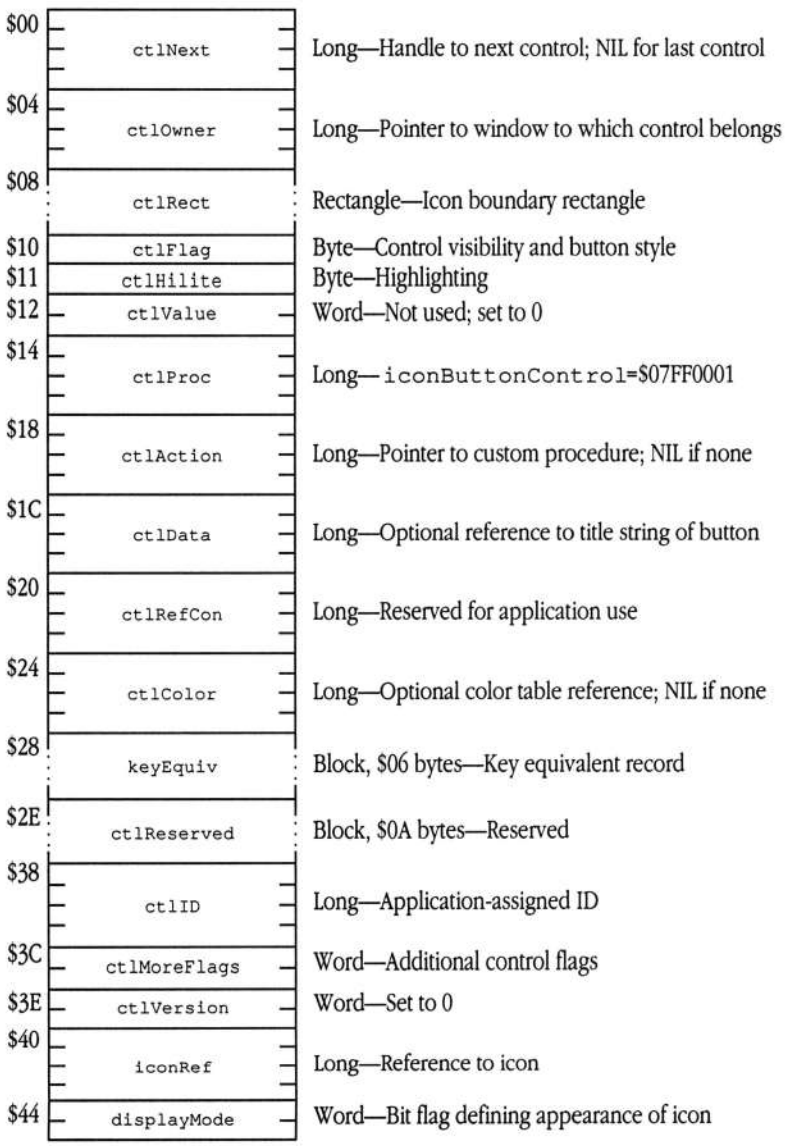
`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Set to 1 if check box has keystroke equivalent.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–4	Must be set to 0.
Color table reference	bits 3–2	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 by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Title reference	bits 1–0	Defines type of title reference in `ctlData`. 00 = Title reference is by pointer 01 = Title reference is by handle 10 = Title reference is by resource ID (resource type corresponds to string type) 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



Valid values for `ctlFlag` are

`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–3	Must be set to 0.
`showBorder`	bit 2	1 = No border, 0 = Show border.
`buttonType`	bits 1–0	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

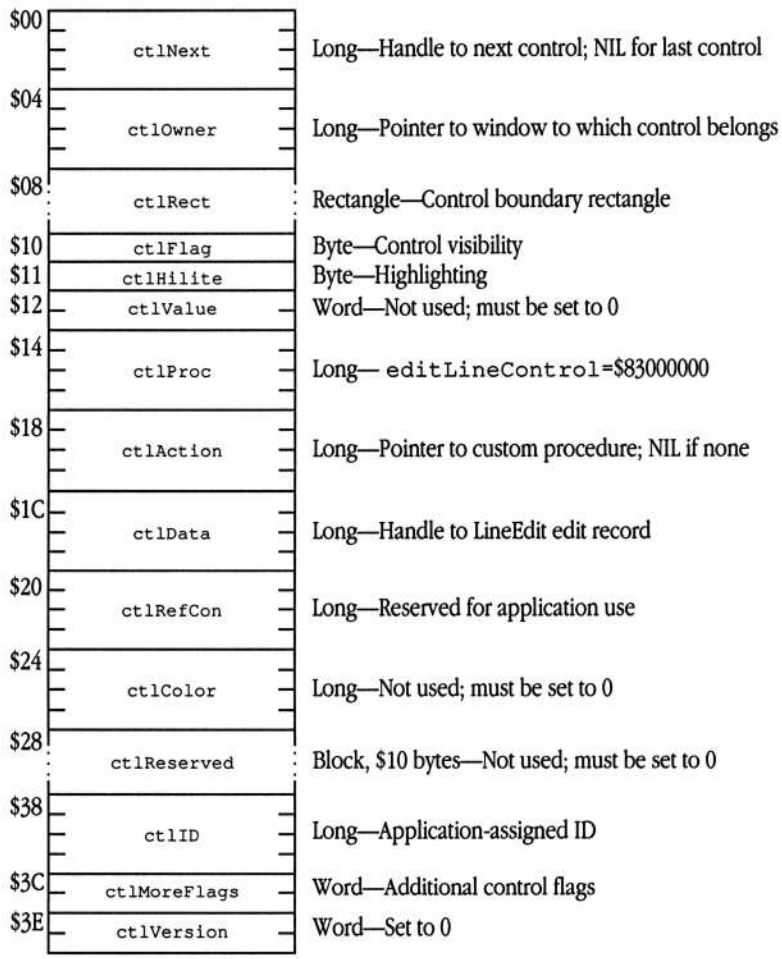
`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Must be set to 0.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–6	Must be set to 0.
Icon reference	bits 5–4	Defines type of icon reference in `iconRef`. 00 = Icon reference is by pointer 01 = Icon reference is by handle 10 = Icon reference is by resource ID (resource type of `rIcon`, $8001) 11 = Invalid value
Color table reference	bits 3–2	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 by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Title reference	bits 1–0	Defines type of title reference in `ctlData`. 00 = Title reference is by pointer 01 = Title reference is by handle 10 = Title reference is by resource ID (resource type of `rPString`, $8006) 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



Valid values for `ctlFlag` are

`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–0	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.
`fCtlWantEvents`	bit 13	Must be set to 1.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–2	Must be set to 0.
Text reference	bits 1–0	Defines type of text reference in `ctlData`. 00 = Text reference is by pointer 01 = Text reference is by handle 10 = Text reference is by resource ID (resource type of `rPString`, $8006) 11 = Invalid value

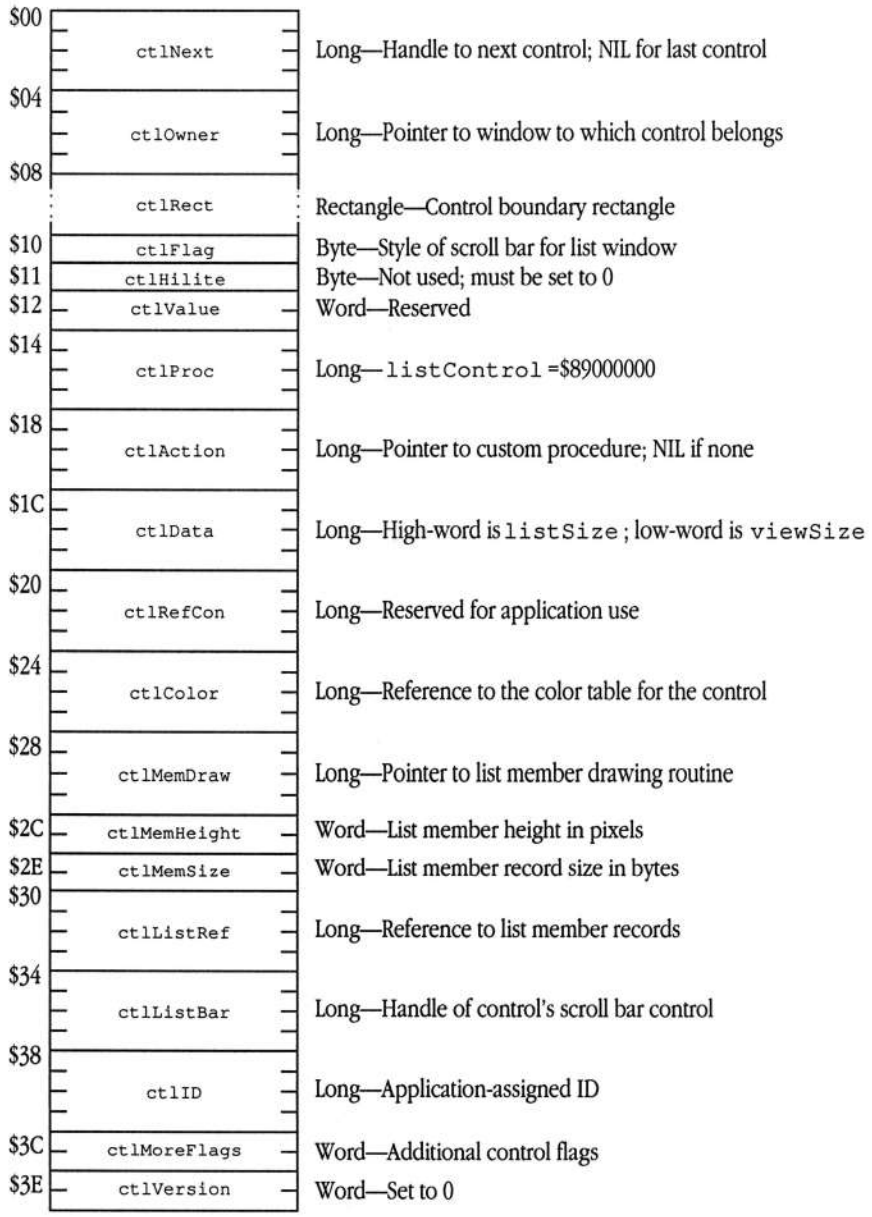
`ctlData` The Control Manager stores the handle to the LineEdit edit record in the `ctlData` field. If you want to issue LineEdit tool calls directly, you can retrieve the handle from that field.

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



Valid values for `ctlFlag` are

`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–0	Must be set to 0.

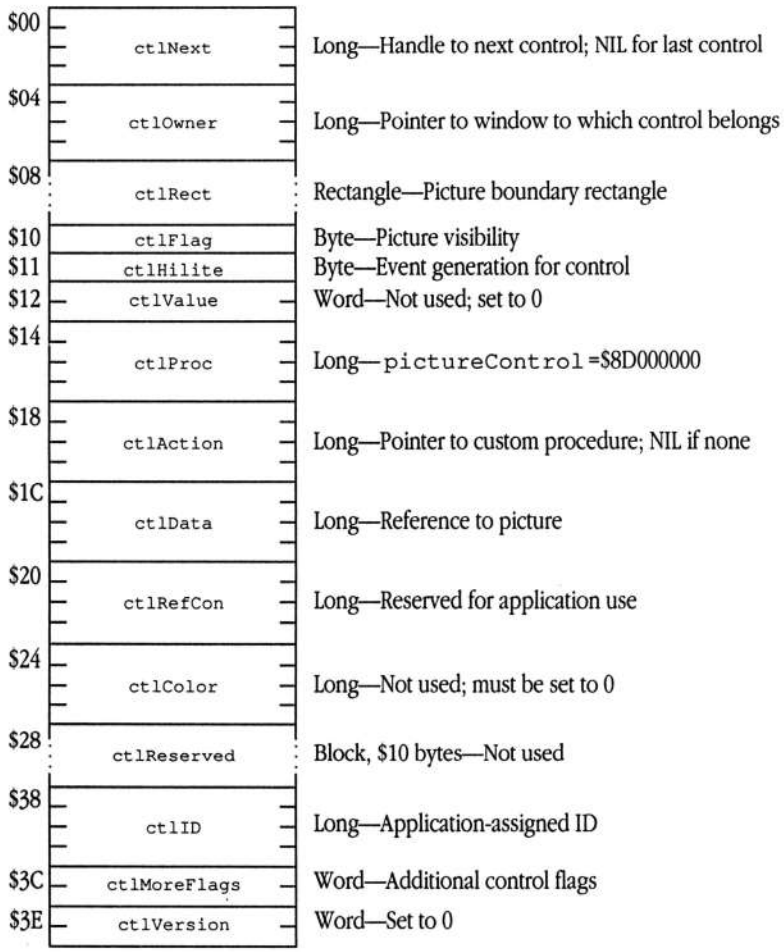
Valid values for `ctlMoreFlags` are

`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Must be set to 0.
`fCtlProcRefNotPtr`	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 9–4	Must be set to 0.
Color table reference	bits 3–2	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 by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
List reference	bits 1–0	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 by pointer 01 = List reference is by handle 10 = List reference is by resource ID (resource type of `rListRef`, $801C) 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



Valid values for `ctlFlag` are

`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–0	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.
`fCtlWantEvents`	bit 13	Must be set to 0.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–2	Must be set to 0.
Picture reference	bits 1–0	Defines type of picture reference in `ctlData`. 00 = Invalid value 01 = Reference is by handle 10 = Reference is by resource ID (resource of type `rPicture`, $8002) 11 = Invalid value

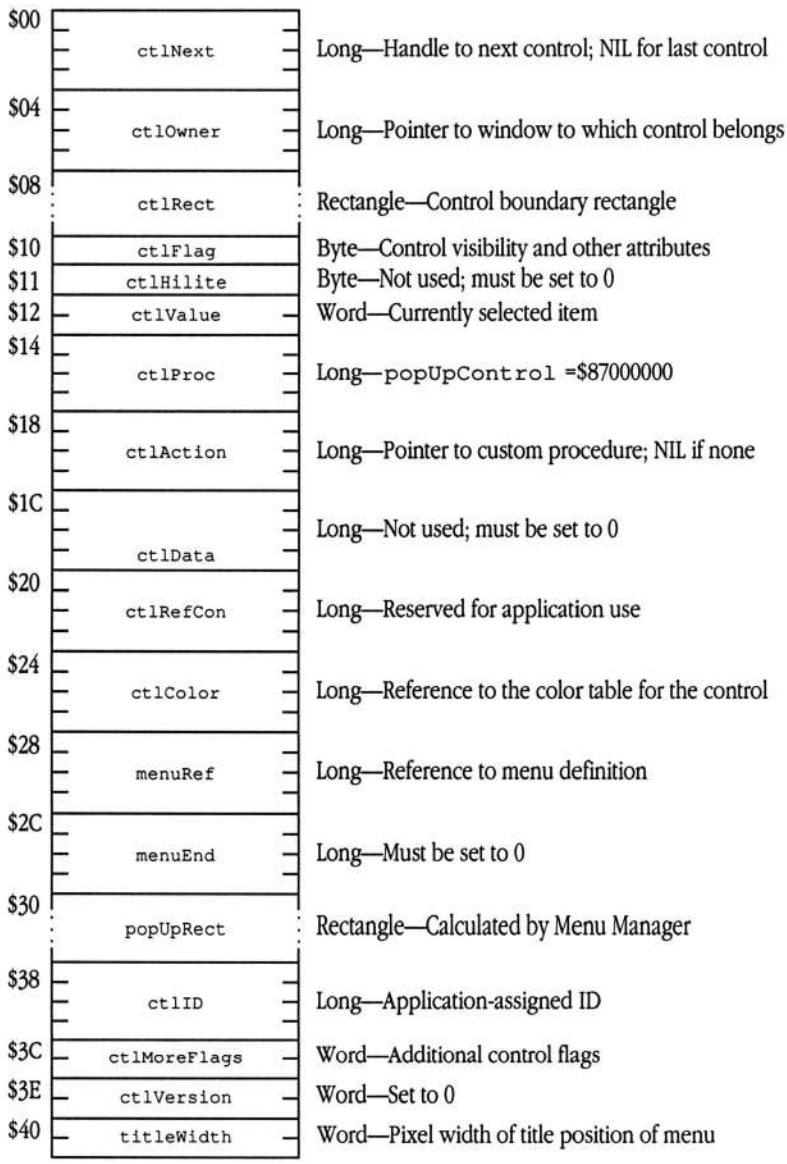
`ctlHilite` Specifies whether the control wants to receive mouse events. The values for `ctlHilite` are

- 0 Events are generated when the mouse button is pressed in the control
- 255 No events are generated when the mouse button 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



Valid values for `ctlFlag` are

`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
`fType2PopUp`	bit 6	Indicates type of pop-up menu. 0 = Draw normal pop-up menu 1 = Draw pop-up menu with white space (type 2)
`fDontHiliteTitle`	bit 5	Controls highlighting of the control title. 0 = Highlight title 1 = Do not highlight title
`fDontDrawTitle`	bit 4	Indicates whether the Control Manager is to draw the menu title. 0 = Draw the title 1 = Do not draw the title
`fDontDrawResult`	bit 3	Indicates whether result is shown. 0 = Draw the result 1 = Do not draw the result in the result area after a selection
`fInWindowOnly`	bit 2	Controls how much the pop-up menu can expand; this is particularly relevant to type 2 pop-up menus (see Chapter 37, "Menu Manager Update," for details on type 2 pop-up menus). 0 = Allow the pop-up menu to expand to the size of the screen 1 = Keep the pop-up menu in the current window
`fRightJustifyTitle`	bit 1	Controls title justification. 0 = Left-justify the title 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) and the value for `titleWidth` is also adjusted
`fRightJustifyResult`	bit 0	Controls result justification. 0 = Left-justify the selection `titleWidth` pixels from the left of the pop-up rectangle 1 = Right-justify the selection

Valid values for `ctlMoreFlags` are

`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Must be set to 1 if the pop-up menu has any keystroke equivalents defined.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–5	Must be set to 0.
Color table reference	bits 4–3	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 by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
`fMenuDefIsText`	bit 2	Defines type of data referred to by `menuRef`. 0 = `menuRef` is a reference to a menu template (See Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference* for details on format and content of a menu template.) 1 = `menuRef` is a pointer to a text stream in `NewMenu` format (Again, see Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference* for details.)
Menu reference	bits 1–0	Defines type of menu reference in `menuRef` (if `fMenuDefIsText` is set to 1, then these bits are ignored). 00 = Menu reference is by pointer 01 = Menu reference is by handle 10 = Menu reference is by resource ID (resource type of `rMenu`, $8009) 11 = Invalid value

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

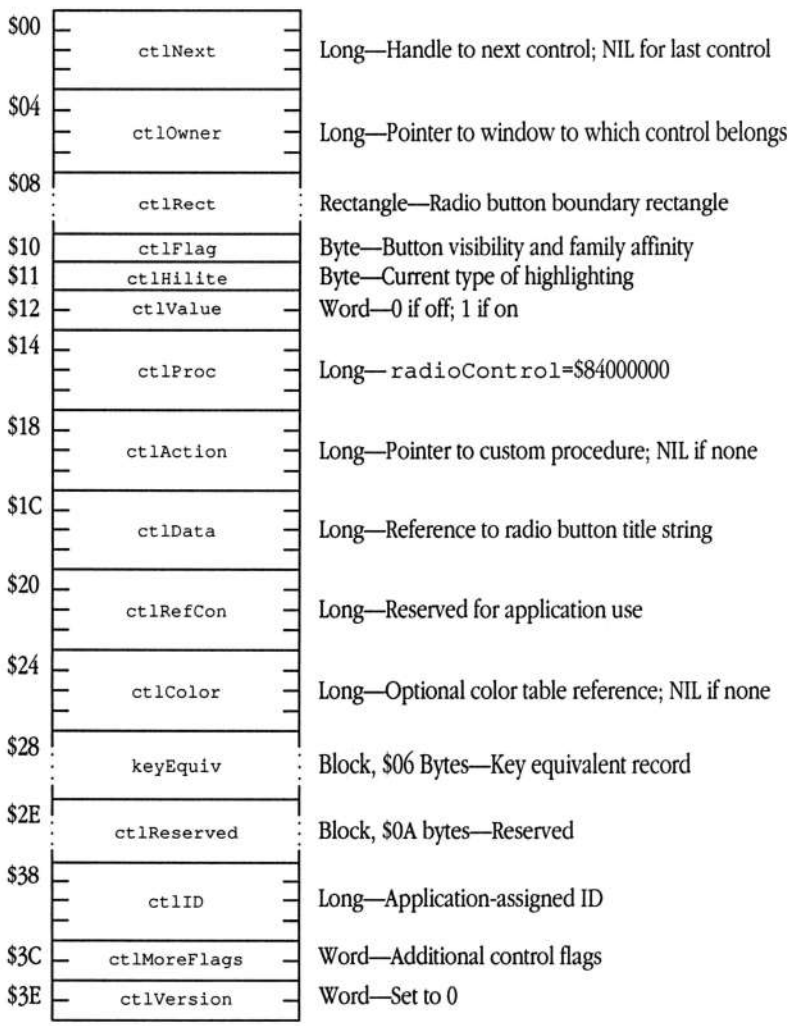
`ctlValue` Contains the item number of the currently selected item.

`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.
`titleWidth`	Contains the value set in the `titleWidth` 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



Valid values for `ctlFlag` are

`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Family number	bits 6–0	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

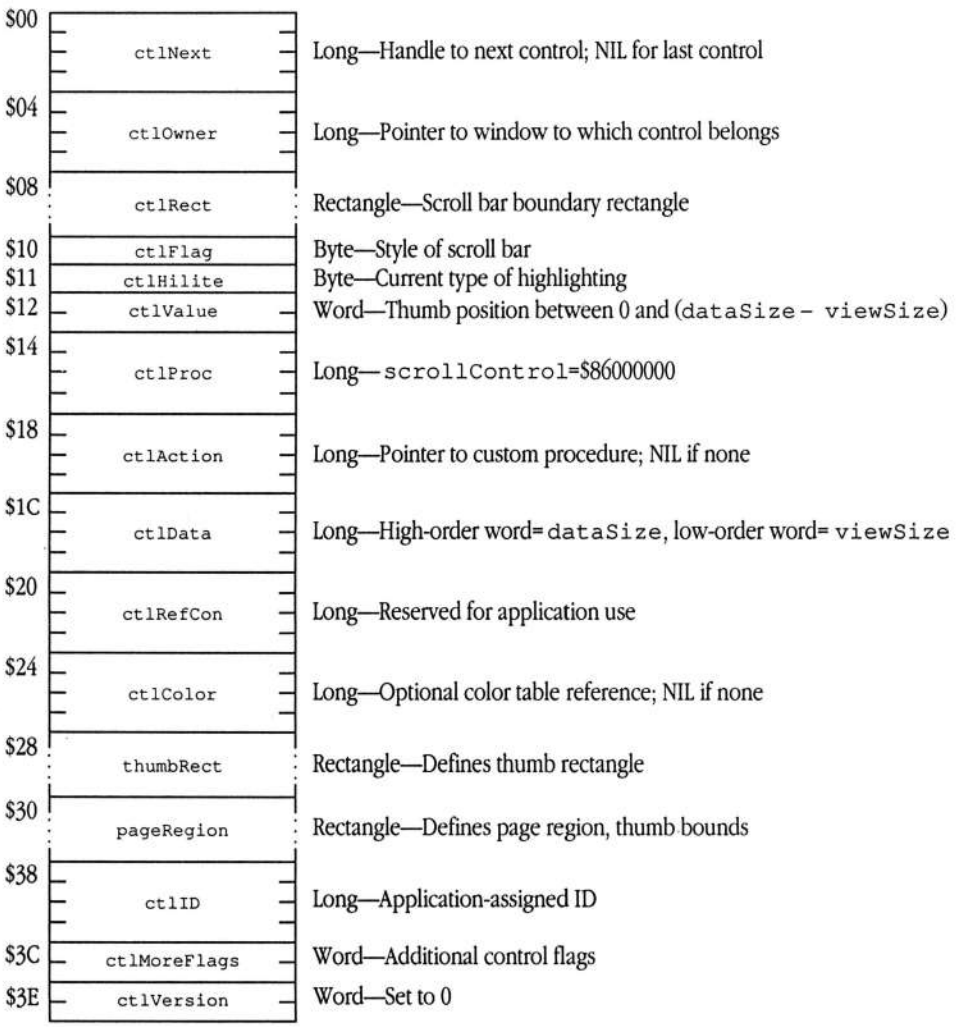
`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Set to 1 if button has keystroke equivalent.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–4	Must be set to 0.
Color table reference	bits 3–2	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 by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Title reference	bits 1–0	Defines type of title reference in `ctlData`. 00 = Title reference is by pointer 01 = Title reference is by handle 10 = Title reference is by resource ID (resource type corresponds to string type) 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



Valid values for `ctlFlag` are

`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–5	Must be set to 0.
`horScroll`	bit 4	0 = Vertical scroll bar, 1 = Horizontal scroll bar.
`rightFlag`	bit 3	0 = Bar has no right arrow, 1 = Bar has right arrow.
`leftFlag`	bit 2	0 = Bar has no left arrow, 1 = Bar has left arrow.
`downFlag`	bit 1	0 = Bar has no down arrow, 1 = Bar has down arrow.
`upFlag`	bit 0	0 = Bar has no up arrow, 1 = Bar has up arrow.

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

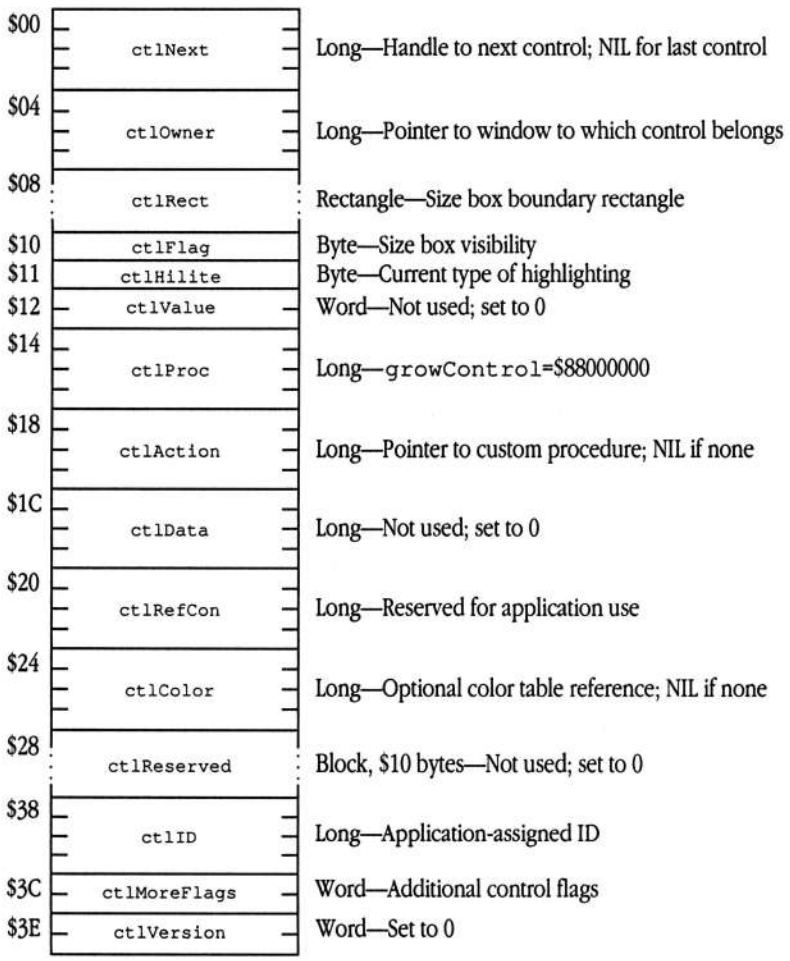
Valid values for `ctlMoreFlags` are

`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Must be set to 0.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–4	Must be set to 0.
Color table reference	bits 3–2	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" earlier in this chapter for the definition of the scroll bar color table. 00 = Color table reference is by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Reserved	bits 1–0	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



Valid values for `ctlFlag` are

`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–1	Must be set to 0.
`fCallWindowMgr`	bit 0	0 = Just highlight control, 1 = Call `GrowWindow` and `SizeWindow` to track this control.

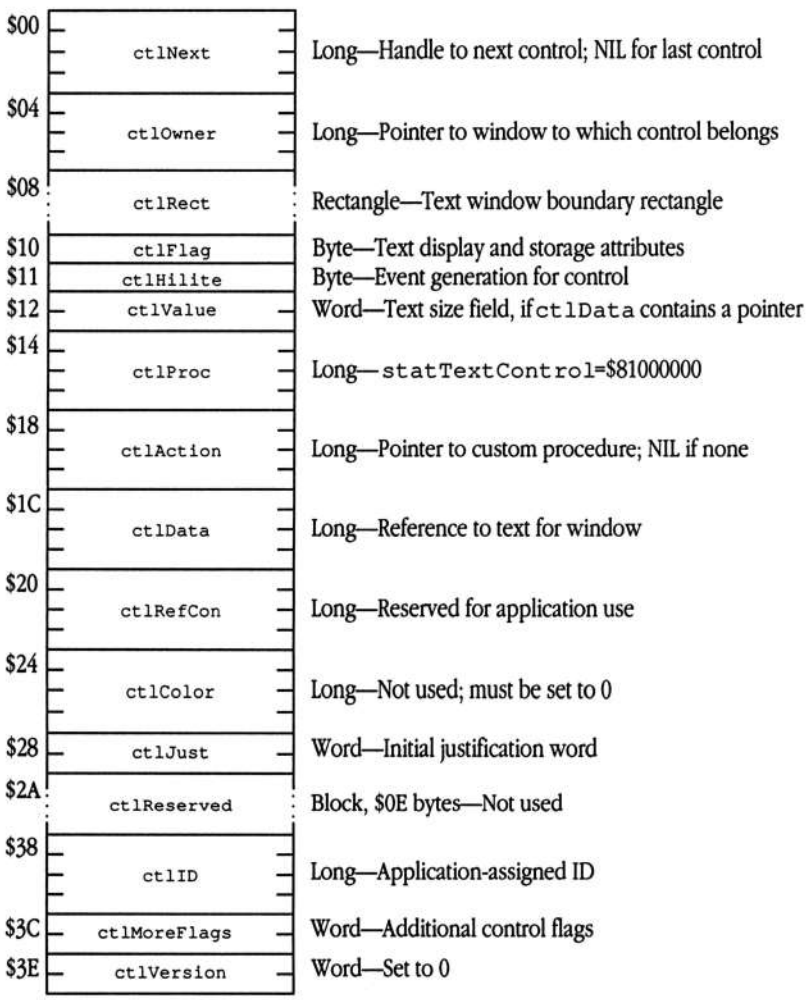
Valid values for `ctlMoreFlags` are

`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Must be set to 0.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–4	Must be set to 0.
Color table reference	bits 3–2	Defines type of reference in `ctlColor` (if it is not NIL). See "Error Corrections" at the beginning of this chapter for the definition of the size box color table. 00 = Color table reference is by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Reserved	bits 1–0	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



Valid values for `ctlFlag` are

`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–2	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.
`fCtlWantEvents`	bit 13	Must be set to 0.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–2	Must be set to 0.
Text reference	bits 1–0	Defines type of text reference in `ctlData`. 00 = Text reference is by pointer 01 = Text reference is by handle 10 = Text reference is by resource ID (resource type of `rTextForLETextBox2`, $800B) 11 = Invalid value

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

- 0 Events are generated when the mouse button is pressed in the control
- 255 No events are 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 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

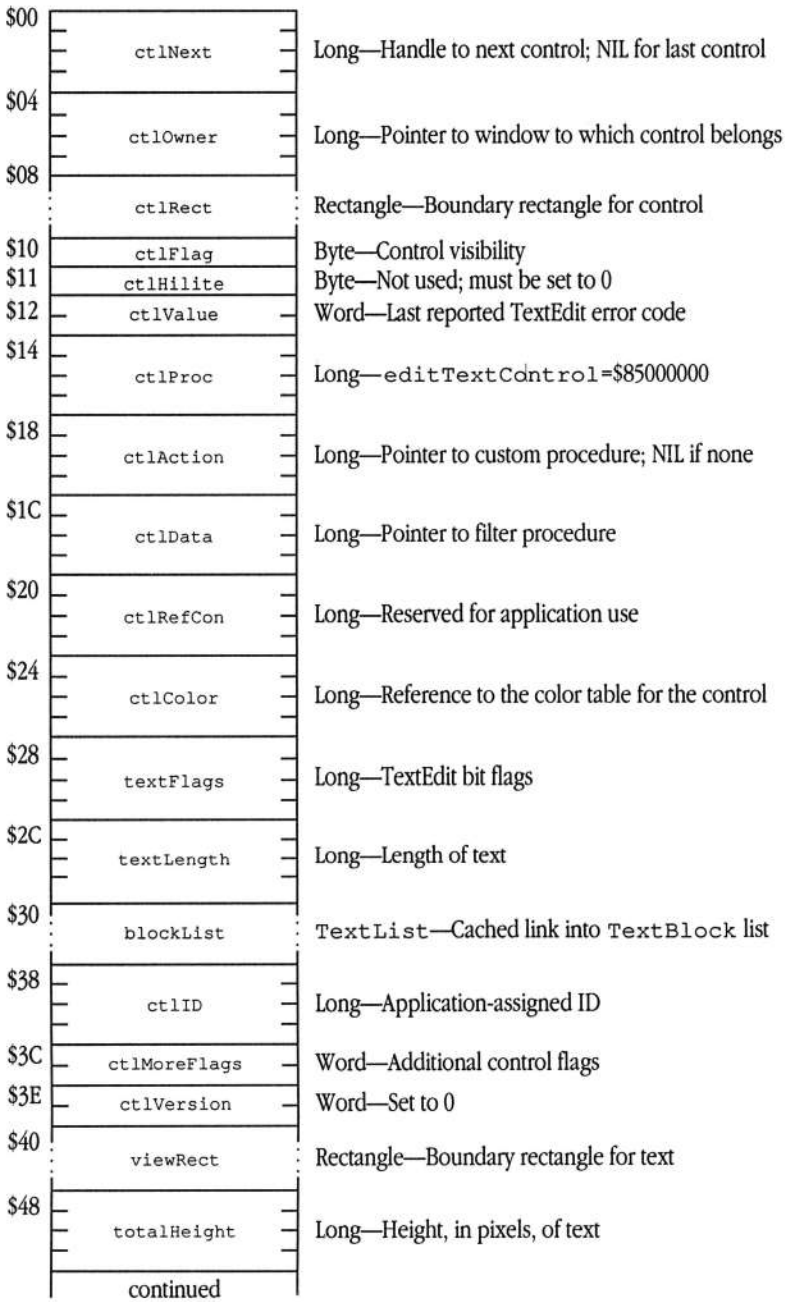
`leftJustify`	0	Text is left justified in the display window.
`centerJustify`	1	Text is centered in the display window.
`rightJustify`	–1	Text is right justified in the display window.
`fullJustify`	2	Text is fully justified (both left and right) in the display window.

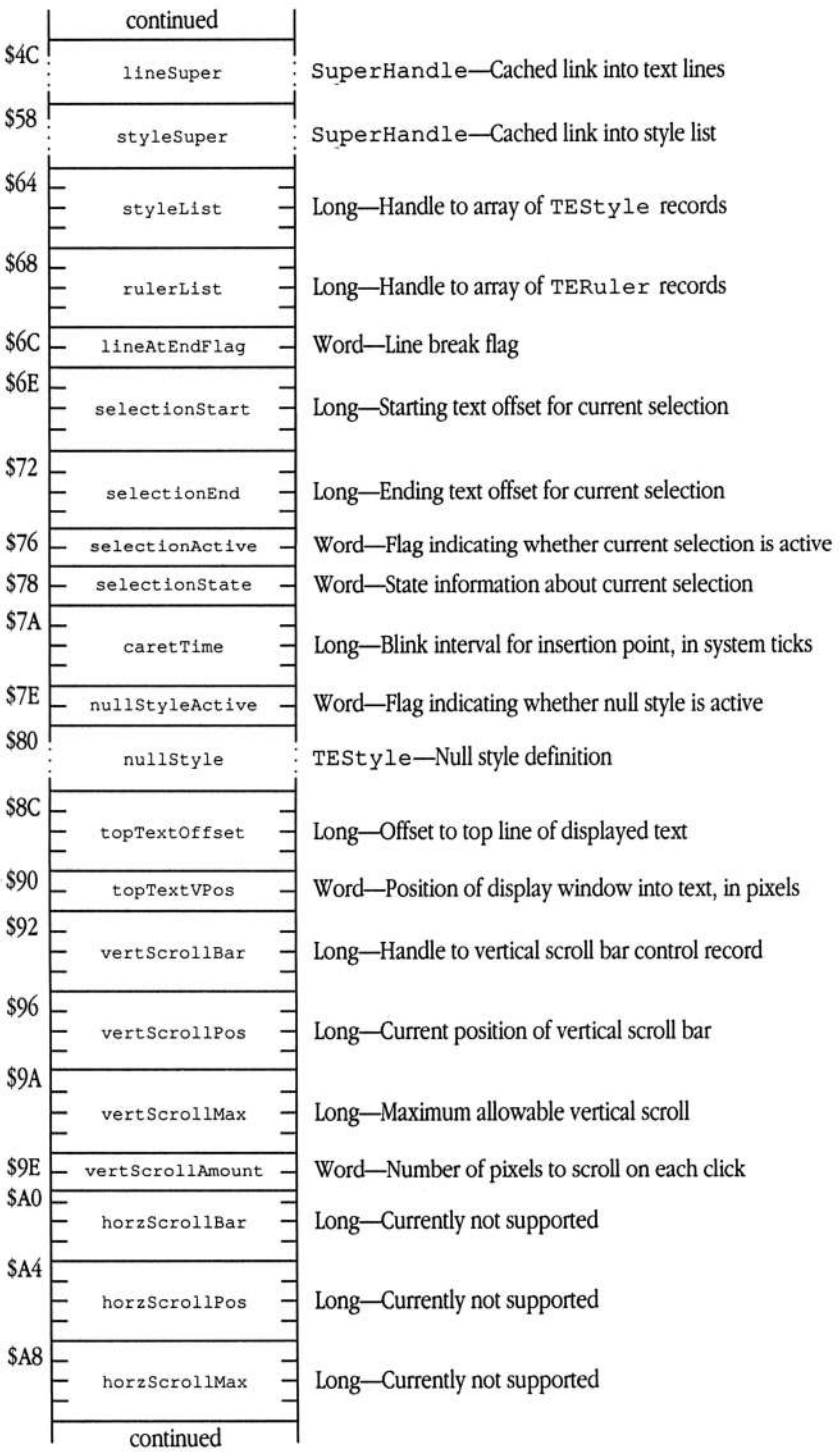
Static text controls do not support color tables. To display text of different color, you must embed the appropriate commands into the text string you are displaying. See the discussion of `LETextBox2` in Chapter 10, "LineEdit Tool Set," in Volume 1 of the *Toolbox Reference* for details on command format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

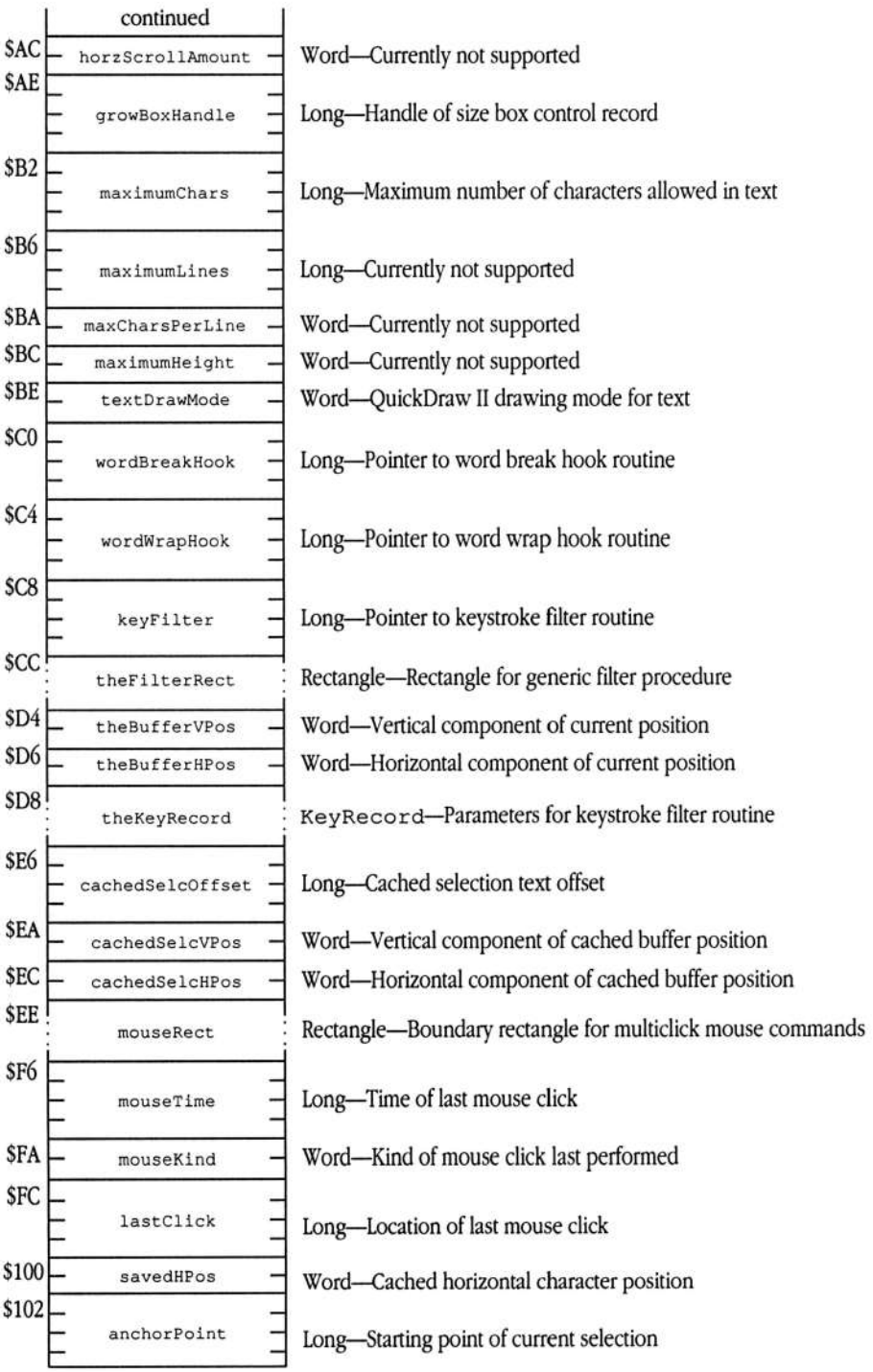
TextEdit control record

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

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




continued
$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
$76
selectionActive
Word—Flag indicating whether current selection is active
$78
selectionState
Word—State information about current selection
$7A
caretTime
Long—Blink interval for insertion point, 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
$A0
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
$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	0 = Visible, 1 = Invisible.
`fRecordDirty`	bit 6	Indicates whether text or style information for the record has changed (TextEdit sets this bit but never clears it—your application must set the bit to 0 whenever it saves the record). 0 = No text or style information has changed 1 = Text or style information has changed
Reserved	bits 5–0	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. 0 = Do not restrict the number of styles in the text 1 = Allow only one style in the text
`fNoWordWrap`	bit 28	Indicates TextEdit word wrap behavior. 0 = Perform word wrap to fit the ruler 1 = Do not word wrap the text; break lines only on CR ($0D) characters
`fNoScroll`	bit 27	Controls user access to scrolling. 0 = Scrolling permitted 1 = Do not allow either manual or auto-scrolling
`fReadOnly`	bit 26	Restricts the text in the window to read-only operations (copying from the window will still be allowed). 0 = Editing permitted 1 = No editing allowed
`fSmartCutPaste`	bit 25	Controls TextEdit support for smart cut and paste (see Chapter 49, "TextEdit Tool Set," for details on smart cut and paste support). 0 = Do not use smart cut and paste 1 = Use smart cut and paste
`fTabSwitch`	bit 24	Defines behavior of the Tab key (see Chapter 49, "TextEdit Tool Set," for details). 0 = Tab inserted in TextEdit document 1 = Tab to next control in the window

`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). 0 = Do not draw rectangle 1 = 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's resizing of the edit window. If this bit is set to 1, TextEdit automatically adjusts the right margin value for the ruler. 0 = Do not resize the ruler 1 = Resize the ruler
`fDisableSelection`	bit 20	Controls whether user can select text. 0 = User can select text 1 = User cannot select text
`fDrawInactiveSelection`	bit 19	Controls how inactive selected text is displayed. 0 = TextEdit does nothing special when displaying inactive selections 1 = TextEdit draws a box around inactive selections
Reserved	bits 18–0	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.
`fCtlWantEvents`	bit 13	Must be set to 1.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	If this bit is set to 1, a size box is created in the lower-right corner of the window. Whenever the control window is resized, the edit text is resized and redrawn.
`fCtlIsMultiPart`	bit 10	Must be set to 1.

Reserved	bits 9–4	Must be set to 0.
Color table reference	bits 3–2	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 Tool Set," in this book. 00 = Color table reference is by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Style reference	bits 1–0	Defines type of style reference in `styleRef`. The format for a TextEdit style descriptor is described in Chapter 49, "TextEdit Tool Set," later in this book. 00 = Style reference is by pointer 01 = Style reference is by handle 10 = Style reference is by resource ID (resource type of `rStyleBlock`, $8012) 11 = Invalid value

△ **Important** Do not set `fCtlTellAboutSize` to 1 unless the text edit record also has a vertical scroll bar. This flag works only for TextEdit records that are controls. △

`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 the unique styles 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 or equal to `selectionEnd`.

`selectionEnd` Ending text offset for the current selection. Must always be greater than or equal to `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 cursor, 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 on 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.

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

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

◆ *Note:* Although TextEdit defines the `vertScrollPos` field as a long word, standard Apple IIGS scroll bars support only the low-order word. This leads to unpredictable scroll bar behavior during the editing of large documents.

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

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

`horzScrollBar` Currently not supported.

`horzScrollPos` Currently not supported.

`horzScrollMax` Currently not supported.

`horzScrollAmount`
Currently not supported.

`growBoxHandle` Handle of size box control record.

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

`maximumLines` Currently not supported.

`maxCharsPerLine`
Currently not supported.

`maximumHeight` Currently not supported.

`textDrawMode` QuickDraw II drawing mode for the text. See Chapter 16, "QuickDraw II," in Volume 2 of the *Toolbox Reference* for more information on QuickDraw II drawing modes.

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

`keyFilter` Pointer to the keystroke filter routine. See Chapter 49, "TextEdit Tool Set," 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 Tool Set," 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 Tool Set," 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 Tool Set," 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 more than once in the region defined by this rectangle during 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 determine where on a line the cursor should appear when the user presses the up or down scroll arrow.

`anchorPoint` Defines the character at which the user began to select the text in 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 of the Desk Manager

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

Scrollable CDA menu

The classic desk accessory (CDA) menu is now scrollable. Previously, the menu held a maximum of 13 commands 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 (◆◆◆ `more` ◆◆◆) at each scrollable end of the menu. That is, if there are additional commands above those currently visible, the more message appears at the top of the menu. Similarly, if there are more commands 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 the selection box up one entry in the menu; no effect if the selection box is at the top of the menu
Command–Up Arrow	Moves the selection box up one page in the menu; no effect if the selection box is at the top of the menu
Down Arrow	Moves the selection box down one entry in the menu; no effect if the selection box is at the bottom of the menu
Command–Down Arrow	Moves the selection box down one page in the menu; no effect if the selection box is at the bottom of the menu
Enter or Return	Selects the highlighted item
Esc	Selects Quit

Run queue

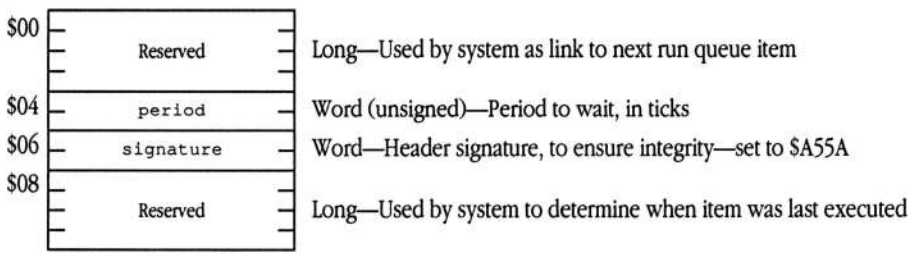
The **run queue** allows you to install tasks **(run items)** that need to be called periodically. You establish the periodicity of the call by managing a field in the run item header. The Desk Manager has two new system calls, `AddToRunQ` and `RemoveFromRunQ`, that allow you to install and remove run 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 in Figure 29-1.

■ **Figure 29-1** Run item header



`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. Although the run queue supports call frequencies up to approximately 60 calls 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, you must ensure that run items 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.

Although 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 quite 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

The following sample run item causes the speaker to beep every 15 minutes:

```
;
; RunQ example task that beeps every 15 minutes.
; It is provided in MPW IIgs 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 preceding item into the run queue:

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

New Desk Manager calls

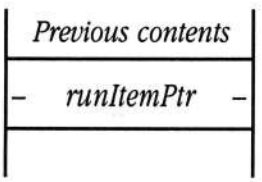
The following new Desk Manager calls support the run queue and desk accessory removal.

AddToRunQ $1F05

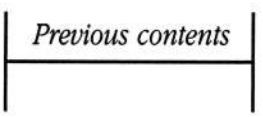
Adds the specified routine to the head of the run queue.

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal void AddToRunQ(runItemPtr);

Pointer   runItemPtr;
```

RemoveCDA $2105

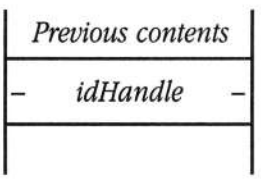
Removes the specified CDA from the Desk Manager CDA list. This routine does *not* dispose of the memory used by the desk accessory.

This routine is the complement of `InstallCDA` (which is described in Chapter 5, "Desk Manager," in Volume 1 of the *Toolbox Reference*).

Issue this call with caution. 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

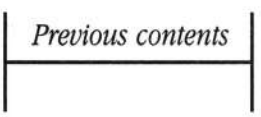
Stack before call



Long—Handle to CDA header

<—SP

Stack after call



<—SP

Errors

$0510	`daNotFound`	Specified desk accessory not found.

C

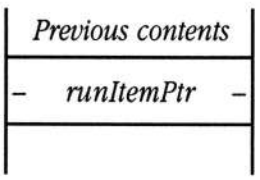
```
extern pascal void RemoveCDA(idHandle);

Handle    idHandle;
```

RemoveFromRunQ $2005

Removes the specified run item from the run queue.

Parameters

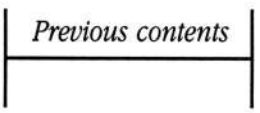
Stack before call



Long—Pointer to run item to remove

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void RemoveFromRunQ(runItemPtr);

Pointer   runItemPtr;
```

RemoveNDA $2205

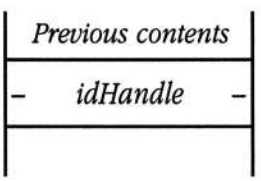
Removes the specified NDA from the Desk Manager NDA list. This routine does *not* dispose of the memory used by the desk accessory.

This routine is the complement of `InstallNDA` (which is described in Chapter 5, "Desk Manager," in Volume 1 of the *Toolbox Reference*).

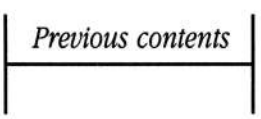
This call does not rebuild the Apple menu. Your application must rebuild the menu by issuing the `FixAppleMenu` tool call.

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 error corrections to the documentation 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 documents errors in Chapter 6, "Dialog Manager," in Volume 1 of the *Toolbox Reference.*

- A statement about `SetDItemType` on page 6-82 of Volume 1 of the *Toolbox Reference* is in error. This call is *not* 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.
- An entry in Table 6-3 on page 6-12 of Volume 1 of the *Toolbox Reference* is incorrect. The Dialog Manager does *not* support dialog item type values of `picItem` or `iconItem`.

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.*

Error correction

This section documents an error in Chapter 7, "Event Manager," in Volume 1 of the *Toolbox Reference*.

- The description of the `EMShutDown` tool call incorrectly states that the call returns no errors. This call can return any valid Event Manager error code.

New features of the Event Manager

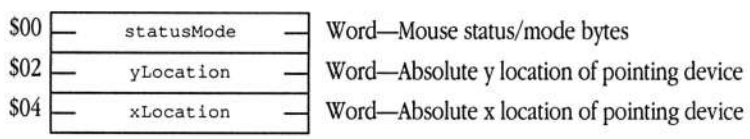
The following sections discuss new features of the Event Manager.

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` while journaling is enabled, 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 (called `EventJournalRec`) has the format shown in Figure 31-1.

- **Figure 31-1** Journal record for mouse event



`statusMode` Mouse status and mode bytes, as described on pages 14-35 and 14-36 of the *Toolbox Reference*, Volume 1.

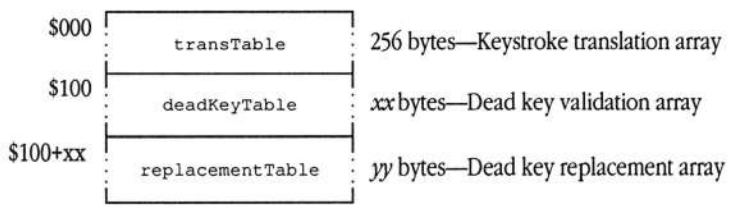
Keyboard input changes

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. The Event Manager provides new 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 than the entries in `deadKeyTable` and `replacementTable`, since the first table is more straightforward than the last two.

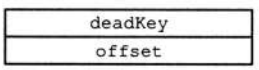
The keystroke translation table must be formatted as shown in Figure 31-2.

■ **Figure 31-2** Keystroke translation table



`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 carriage 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—keystrokes used to introduce multikey sequences that result in single characters. For example, pressing Option-U followed by e yields the character ë. 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:

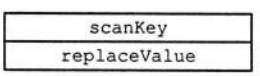


Byte—Character code for dead key
Byte—Offset from `deadKeyTable` into `replacementTable`

`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 subarray must be set to $0000. Each entry in the `replacementTable` must be formatted as follows:



Byte—Character code for dead key combination
Byte—Result character code for dead key combination

`scanKey` Contains a valid character code for a 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

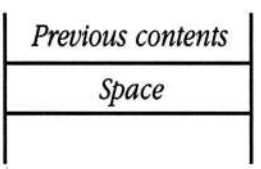
This section describes several new Event Manager calls, many concerning the new keyboard translation feature.

`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

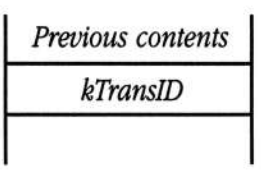
Stack before call



Word—Space for result

<—SP

Stack after call



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

<—SP

Errors None

C

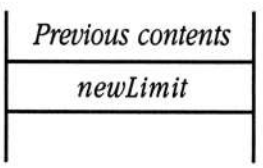
```
extern pascal Word GetKeyTranslation();
```

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

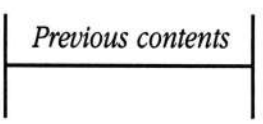
Stack before call



Word—Limit for inserted auto-key events

<—SP

Stack after call



<—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.

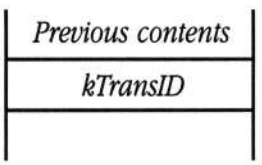
The system reads keystroke translation tables from resources of type `rkTransTable` ($8021) and ID $0FFF06xx, where xx derives from the low-order byte of the *kTransID* parameter.

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

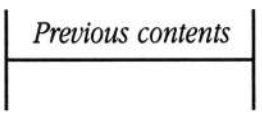
Stack before call



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 sequences)

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.*

Error corrections

- On page 8-4 of Volume 1 of the *Toolbox Reference,* the font family number for the Shaston font is given as 65,524. This is incorrect. The correct decimal value is 65,534 ($FFFE).
- Page 8-24, Volume 1 of the *Toolbox Reference* incorrectly describes the *newSpecs* parameter, indicating that it contains a word of `FontSpecBits`. Actually, this parameter contains `FontStatBits` for the new font.
- Contrary to the call description in the *Toolbox Reference,* the `FMSetSysFont` tool call does *not* load or install the indicated font.

New features of 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 Choose Font dialog box now uses `NewWindow2`, with a control template that can be kept in a resource file. As a result, this dialog box can be translated to languages other than English more easily.
- Scaled fonts may now contain more than 65,535 bytes of data. See Chapter 43, "QuickDraw II Update," in this book for the layout of the new font record.

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.*

Error corrections

- On page 8-4 of Volume 1 of the *Toolbox Reference,* the font family number for the Shaston font is given as 65,524. This is incorrect. The correct decimal value is 65,534 ($FFFE).
- Page 8-24, Volume 1 of the *Toolbox Reference* incorrectly describes the *newSpecs* parameter, indicating that it contains a word of `FontSpecBits`. Actually, this parameter contains `FontStatBits` for the new font.
- Contrary to the call description in the *Toolbox Reference,* the `FMSetSysFont` tool call does *not* load or install the indicated font.

New features of 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 Choose Font dialog box now uses `NewWindow2`, with a control template that can be kept in a resource file. As a result, this dialog box can be translated to languages other than English more easily.
- Scaled fonts may now contain more than 65,535 bytes of data. See Chapter 43, "QuickDraw II Update," 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 box and in the Font menu, and Font Manager crashes, among others.

New Font Manager 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.

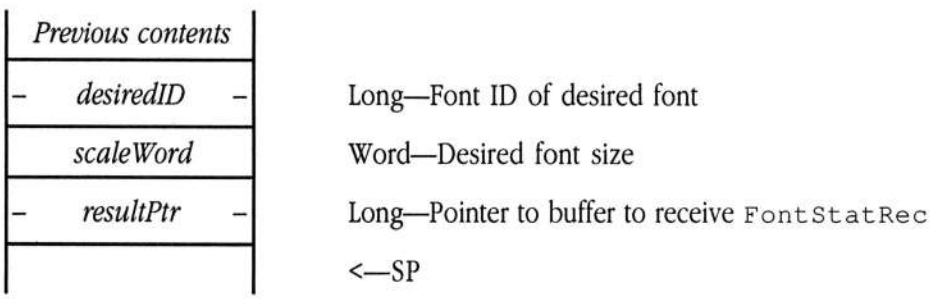
`InstallWithStats` $1C1B

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 installs the font that matches the requested font most closely.

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

Parameters

Stack before call



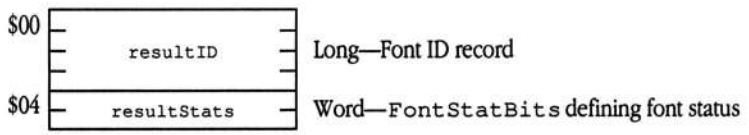
Stack after call

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* contains a `FontStatRec` record 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.*

Clarification

This section presents new information about the `Long2Dec` Integer Math tool call.

- The `Long2Dec` Integer Math tool call now correctly handles input long values whose low-order three bytes are set to 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 of the LineEdit Tool Set

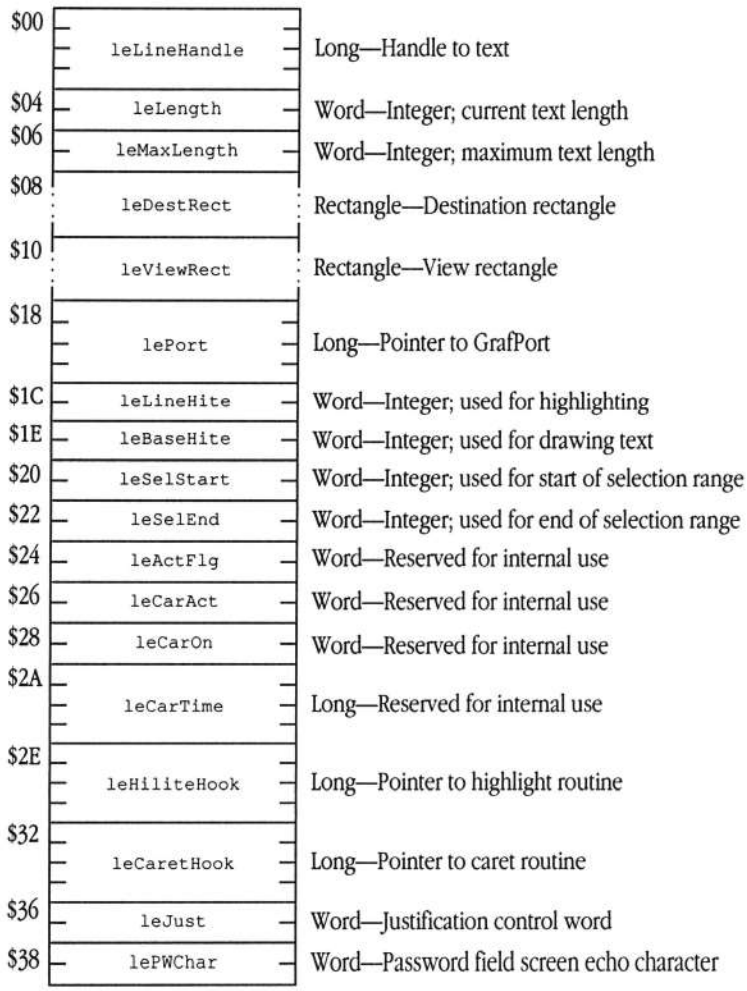
The LineEdit Tool Set supports a number of new features. The following section discusses these new features in detail.

- The LineEdit Tool Set now works within controls. See Chapter 28, "Control Manager Update," in this book for details.
- 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 the 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. Figure 34-1 shows the new format of the LineEdit record.

  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," in this book for more information).

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



`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. The default system value is the asterisk (`*`).

New LineEdit call

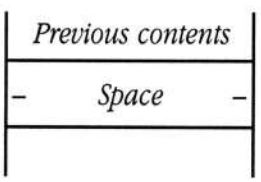
This new LineEdit tool call returns the address of the current LineEdit control definition procedure.

GetLEDefProc $2414

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

Parameters

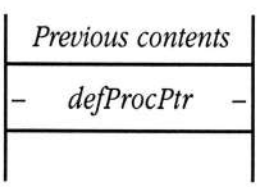
Stack before call



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 following items provide additional information about features previously described in Volume 1 of the *Toolbox Reference.*

- The *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 a user can, for instance, select a disabled command as part of a help dialog box. To make an item unselectable, make it inactive (see "List Manager Definitions" later in this chapter).
- Any List Manager tool call that draws will change fields in the GrafPort record. 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.
- Item 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 understand this relationship, 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 automatically adjusts the `listRect.v2` field and sets 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 of a list item:

inactive	Inactive items appear dimmed and cannot be highlighted or selected. Bit 5 of the list item's `memFlag` field is set to 1.
disabled	Disabled items appear dimmed and cannot be highlighted. Bit 6 of the list item's `memFlag` field is set to 1.
enabled	Enabled items are not dimmed and can be highlighted. Bit 6 of the list item's `memFlag` field is set to 0.
selected	This bit is set when a user clicks the list item or when the item is within a range of selected items. A selected item appears highlighted only if it is also enabled. Bit 7 of the list item's `memFlag` field is set to 1.
highlighted	An item in a list appears highlighted only when it is both selected and enabled. A highlighted item is drawn in the highlight colors. Bit 7 of the `memFlag` field is set to 1 and bit 6 is set to 0.

New features of the List Manager

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

- 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. The List Manager returns this handle when your program issues the `CreateList` List Manager tool call, or preferably, the `NewControl2` Control Manager tool call.
- 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 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` $161C" 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

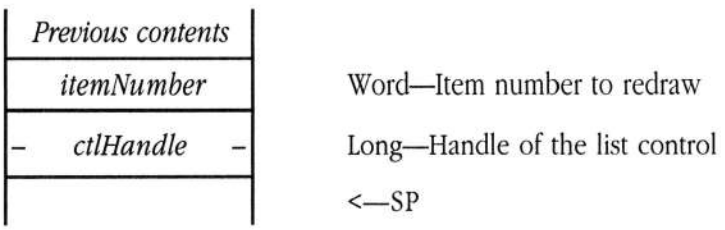
The following new List Manager calls support a new, more flexible programming interface. In general, these calls provide the same functionality as the old versions.

DrawMember2 $111C

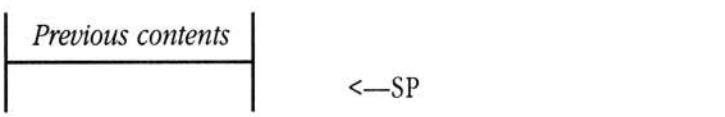
Draws one or all members of a specified list. If your application goes directly to the member record to change the state of a member, the application should then call `DrawMember` or `DrawMember2`. Unlike `DrawMember`, this call accepts an item number specification for the member to draw. Passing an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

Parameters

Stack before call



Stack after call



Errors None

C

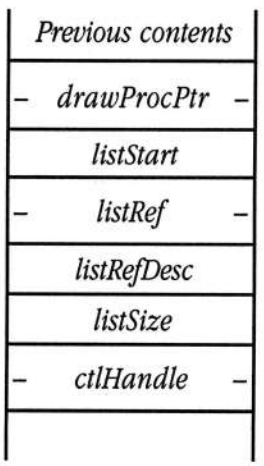
```
extern pascal void DrawMember2(itemNumber,
          ctlHandle);

Word      itemNumber;
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`). The routine uses the *listStart*, *listSize*, and *listRef* parameters to reset the list control.

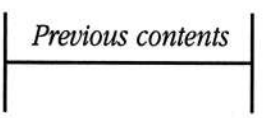
Parameters

Stack before call



Previous contents	
– *drawProcPtr* –	Long—Pointer to member-drawing 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(drawProcPtr, listStart,
          listRef, listRefDesc, listSize,
          ctlHandle);

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

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

listStart Item number of the 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.

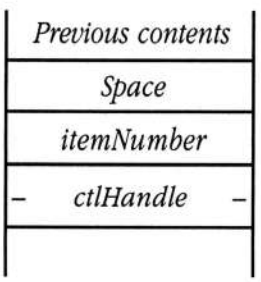
listSize Number of entries in the list. A value of $FFFF tells the List Manager to use the value currently stored in the list control record.

NextMember2 $121C

Searches a specified list record, starting with a specified 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



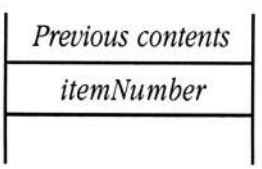
Word—Space for result

Word—Number of item at which search begins

Long—Handle of the list control

<—SP

Stack after call



Word—Item number of selected member; 0 if no more

<—SP

Errors None

C

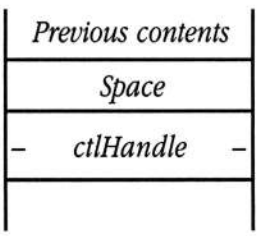
```
extern pascal Word NextMember2(itemNumber,
          ctlHandle);

Word      itemNumber;
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. A list member is considered selected if bit 7 of the member's *memFlag* field is set to 1. 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

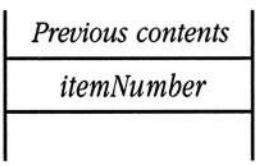


Word—Space for result

Long—Handle of the list control

<—SP

Stack after call



Word—Item number of selected member; 0 if no more

<—SP

Errors None

C

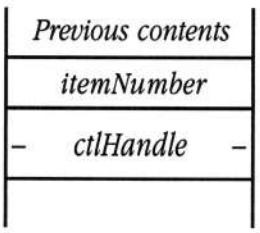
```
extern pascal Word ResetMember2(ctlHandle);

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

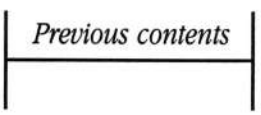


Word—Item number of member to select

Long—Handle of the list control

<—SP

Stack after call



<—SP

Errors None

C

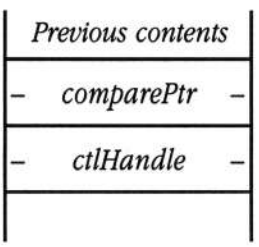
```
extern pascal void SelectMember2(itemNumber,
          ctlHandle);

Word      itemNumber;
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

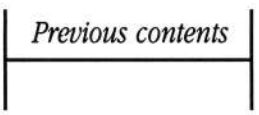


Long—Pointer to comparison routine; NIL for standard compare

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

Figure 12-7 on page 12-10 of Volume 1 of the *Toolbox Reference* shows the low-order bit of the user ID as 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) includes the statement, "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 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 of 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, shortening allocation time.

Out-of-memory queue

The **out-of-memory queue** allows application code to recover gracefully 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 memory that is not crucial to the function of an application or notify the application 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. Purges level 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.

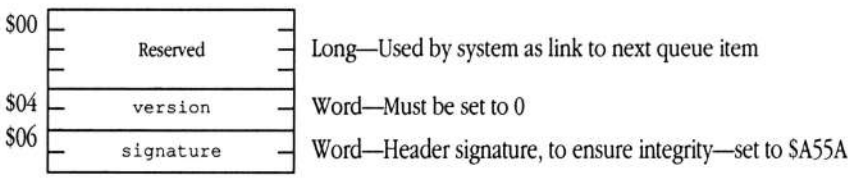
Note that the Memory Manager may invoke an out-of-memory routine twice during the same low-memory condition. In the invocation parameter block for an out-of-memory routine, the Memory Manager passes a flag indicating whether this is the first or second time through the out-of-memory queue. By examining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

Any application, desk accessory, or initialization resource 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 issuing 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 (shown in "Out-of-Memory Routine Example" later in this chapter) for sample application and out-of-memory routine code.

An out-of-memory routine must be preceded by a header formatted as shown in Figure 36-1.

■ **Figure 36-1** Out-of-memory routine header

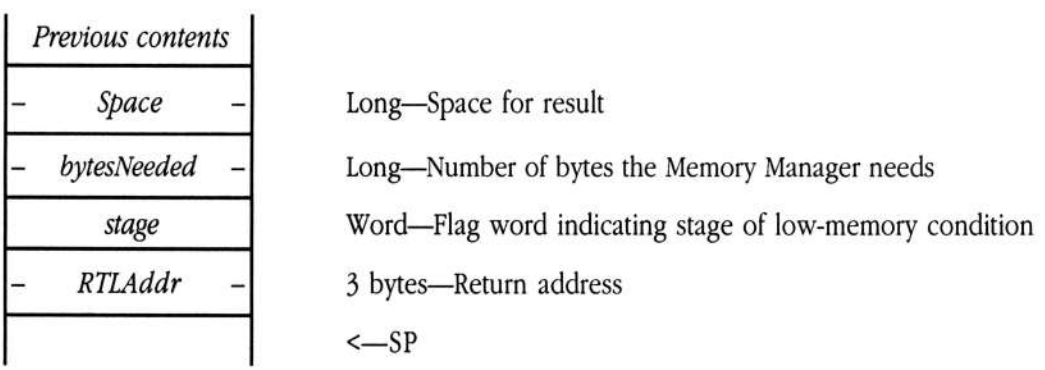


`version` Allows the system to discriminate between current and future types of out-of-memory routines. Must be set to 0.

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

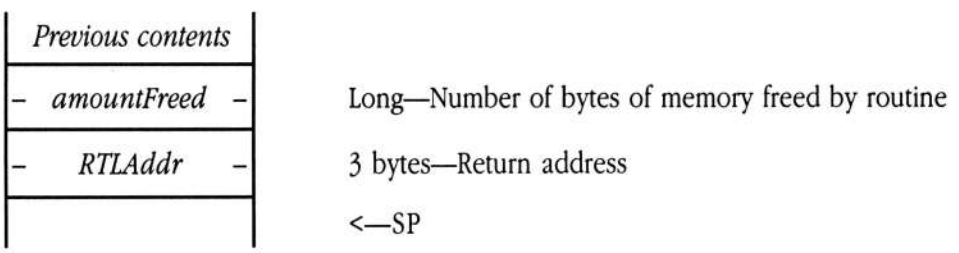
The out-of-memory routine code must immediately follow the `signature` word. If the Memory Manager finds an invalid header for any out-of-memory routine, it terminates with a system death error code of $0209.

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



stage Indicates the stage of the low-memory condition. This flag allows the routine to determine whether this is the first or second invocation for this condition. If the field is set to 0, then this is the first invocation, and the Memory Manager has not done anything else. If the field is set to 1, then this is the second invocation for this low-memory condition, and the Memory Manager reports 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:



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 movable
            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. 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 nonzero 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

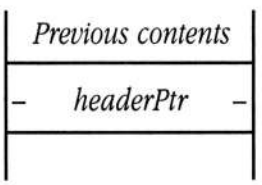
The new Memory Manager call `RealFreeMem` is designed to provide accurate information about available memory. Other new Memory Manager calls support the out-of-memory queue.

`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

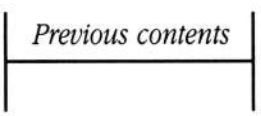
Stack before call



Long—Pointer to out-of-memory routine

<—SP

Stack after call



<—SP

Errors

$0381	`invalidTag`	Correct signature value not found in header.

C

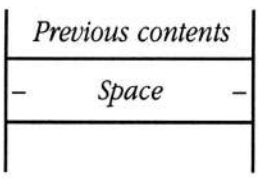
```
extern pascal void AddToOOMQueue(headerPtr);

Pointer   headerPtr;
```

RealFreeMem $2F02

Returns the number of bytes in memory that are free, plus the number that could be made free by purging. The `FreeMem` routine returns only 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

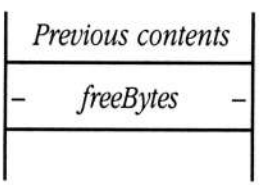
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Number of available bytes in memory

<—SP

Errors None

C

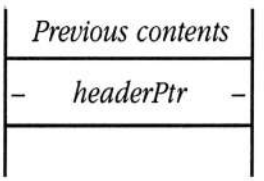
```
extern pascal Long RealFreeMem();
```

RemoveFromOOMQueue $0D02

Removes the specified out-of-memory routine from the queue as described earlier (see "Out-of-Memory Queue" earlier in this chapter). The *headerPtr* parameter should contain the address of the routine header block.

Parameters

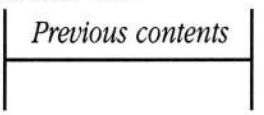
Stack before call



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

This section documents errors in Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference.*

- Part of the description of the `SetSysBar` tool call (pages 13-3 and 13-86) in Volume 1 of the *Toolbox Reference* is incorrect. It includes the mistaken statement that, after an application issues this call, the new system menu bar becomes the current menu bar. In reality, 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 and 13-18), Volume 1 of the *Toolbox Reference* shows that bits 0–5 of the `ctlFlag` field are used to indicate the starting position of the first title in the menu bar. This is incorrect. The `ctlHilite` field defines the starting position of 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.
- The descriptions for the `MenuKey` and `MenuSelect` tool calls are incorrect. The calls do not return selection status information in the `when` field of the event record. Rather, these calls both return selection status information in the `TaskData` field of the task record.

Clarifications

The following items provide additional information about features previously described in Volume 1 of the *Toolbox Reference.*

- 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` 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 the size of a menu. 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 calculate the parameter only if the current setting is 0.

  These are the effects of all three uses:

  - **Pass the new value.** The value passed determines the size of the resulting menu. Use this method when you need a menu of a specific size.
  - **Pass $0000.** The size value is automatically computed. This option is useful if commands are added or deleted, resulting in an incorrect size. The height and width of the menu can be automatically adjusted by calling `CalcMenuSize` with *newWidth* and *newHeight* equal to $0000.
  - **Pass $FFFF.** The width and height of a menu are 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.

- To provide the user with a consistent visual interface, you should always pad your menu titles with leading and trailing space characters. The Apple IIGS Finder™ uses two spaces.

New features of the Menu Manager

This section lists several new features of the Menu Manager and clarifies some information given previously.

- Menus in windows can now display the Apple character (ASCII $14), although not as a multicolored image.
- The color of the menu outline is now also used for lines separating commands.
- 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 of the first menu title in a menu bar is 10 pixels from the left edge of the screen in 640 mode; in 320 mode the title is indented 5 pixels.
- The Menu Manager's justification procedures adjust menu bars in windows. Menu titles are moved to the left if they would otherwise appear to the right of the right edge of the menu bar.
- 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. Here's an 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 are now correct.

  In addition, the Menu Manager now supports two new special characters for menu definition:

  O Outline the text
  S Shadow the text

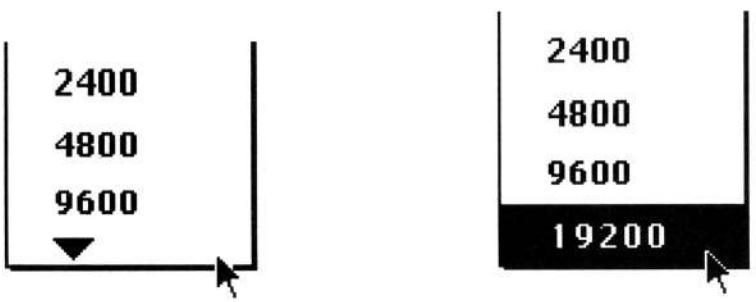
  Other special characters are listed on page 13-14 of Volume 1 of the *Toolbox Reference.* Note that this feature requires the QuickDraw II Auxiliary Tool Set.

- Menus now scroll up or down if their contents do not fit on the screen. Scrollable menus have an arrow at the top and/or bottom, indicating in which directions the menu is scrollable. See Figure 37-1.

  The arrow indicator is not highlighted, but the menu contents scroll when the user drags onto the arrow indicator. When the previously hidden contents are displayed, the indicator disappears.

  Menus scroll at two speeds, depending on what part of the indicator is dragged. If the user drags within the first five pixels of an indicator, scrolling occurs at 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 movable window, dragging that window close to the bottom of the screen may force some of the menus to be scrollable. If there is not enough room for three visible items (up and down indicators and one menu item), then the menu drops 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 pointer is not in 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.
- The Menu Manager `GetMenuTitle` and `GetMItem` tool calls can now return pointers, handles, or resource IDs, depending on how the menu data was originally specified to the `NewMenu` tool call. The type of reference you use when you specify data for the Menu Manager governs how that data is later accessed.

Menu caching

The current version of the Menu Manager introduces new menu caching features. Menu caching provides 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 disappears from the screen, the contents of the saved buffer are copied back to the screen.

With the menu caching feature, when a saved screen image is copied back to the screen, the menu that disappears from the screen 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, if a command is disabled or the items on the menu change—then the cached image is inaccurate, and the Menu Manager must redraw the menu. When a menu image does not change, however, 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.

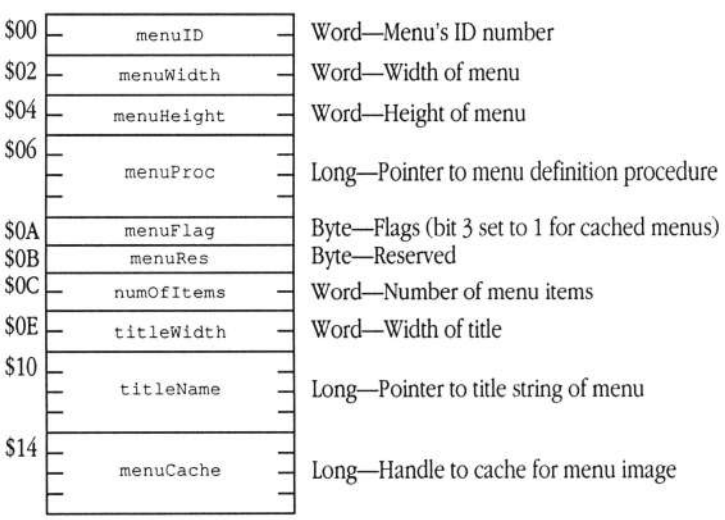
This menu caching scheme should work properly with all existing standard menus. You will have to alter custom menus, however, so that they can take advantage of menu caching. Custom menus will still function normally as long as they do not change the menu record directly, but they will not be able to take advantage of the menu caching scheme to speed display.

Because caching does not work with menus in windows, 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 the definition procedure of a menu knows about caching. A value of 1 indicates that the menu in question is cacheable. A custom menu that uses caching must define a 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 Figure 37-2.

■ **Figure 37-2** Menu record layout for cached menu



Pop-up menus

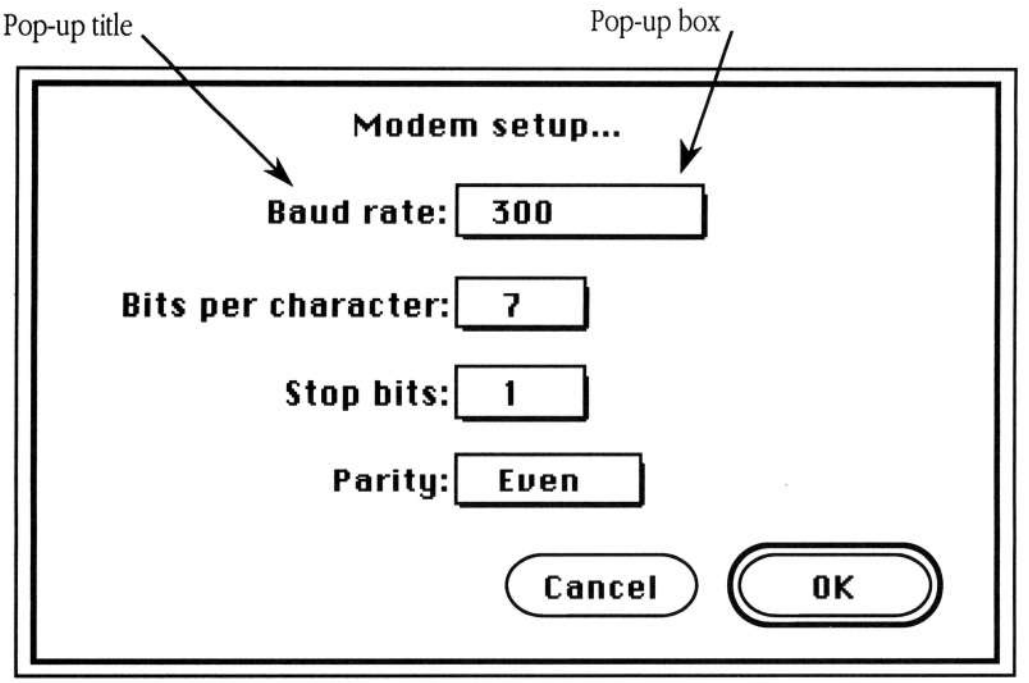
The Menu Manager now supports **pop-up menus.** Pop-up menus exist in a window, not in the menu bar. Figure 37-3 shows a window with pop-up menus. The screen representation of a pop-up menu is a box with a drop shadow that is one pixel thick. When the user clicks inside the pop-up box, the menu appears, with the current value highlighted under the arrow, as shown in Figure 37-4. 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 the pointer in the menu, select an item by positioning the pointer over it and clicking, or not select any item by dragging the pointer 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.

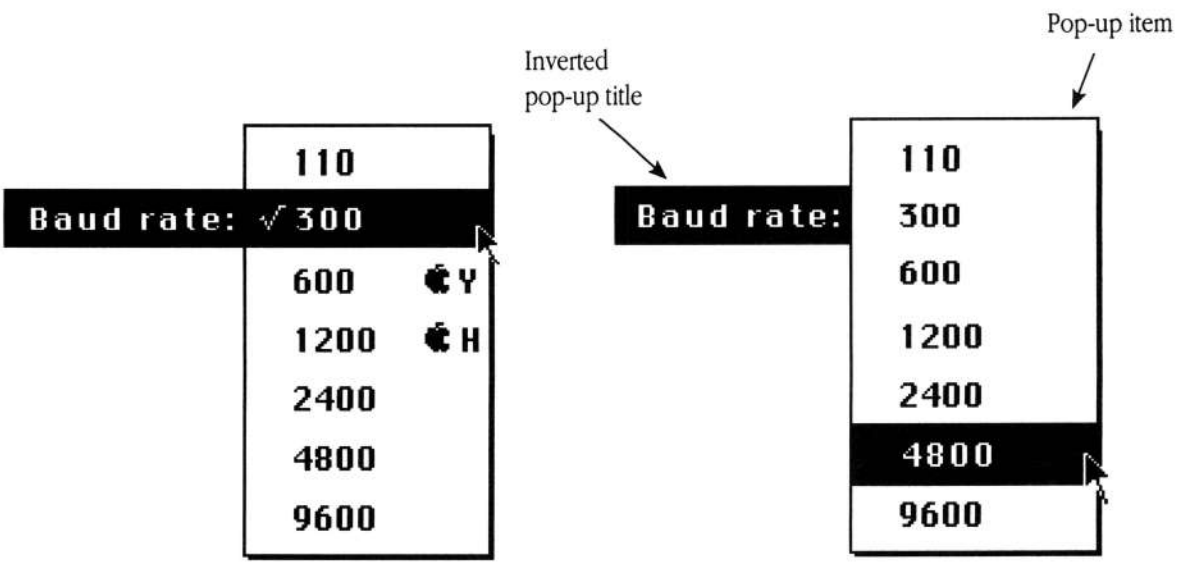
Pop-up menus support most of the standard features and calls available with standard menus:

- Pop-up menu items support keystroke equivalents, which are displayed in the menu (Apple logo with character). Note that if a pop-up keystroke equivalent conflicts with a standard menu 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 style.

■ **Figure 37-3** Window with pop-up menus



■ **Figure 37-4** Dragging through a pop-up menu

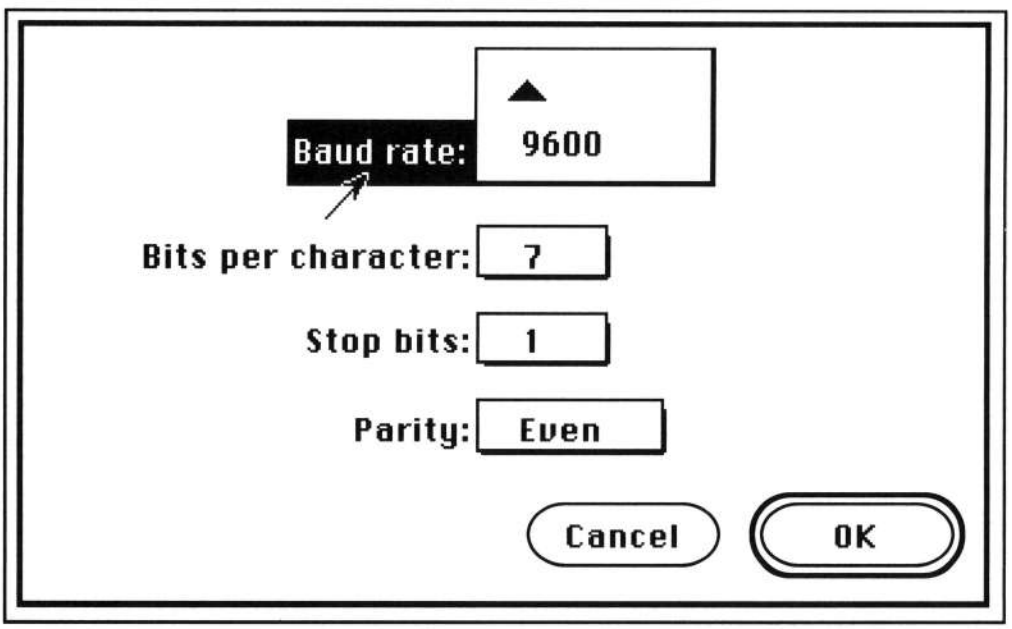


Pop-up menu scrolling options

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.**

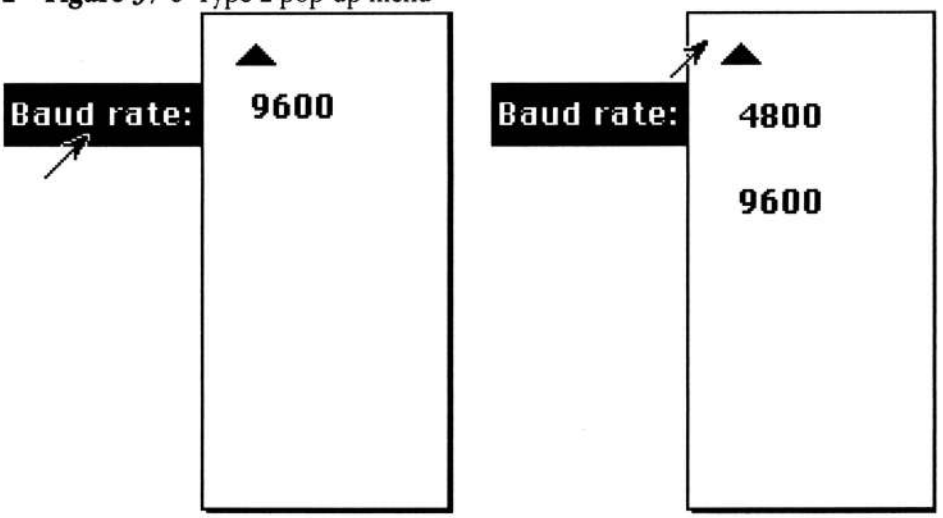
The Menu Manager determines the size of the rectangle in which to draw a type 1 pop-up menu according to the relative position of the current item in the menu and the window constraints of the pop-up menu (see Figure 37-5). The Menu Manager draws the pop-up menu with the current item highlighted and 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 37-5, in which the user can display only a single item.

■ **Figure 37-5** Type 1 pop-up menu



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 37-6 shows how the Baud Rate pop-up menu from Figure 37-5 would appear if it had been defined as a type 2 pop-up menu.

■ **Figure 37-6** Type 2 pop-up menu



By dragging over the scroll indicator, the user can eventually scroll into view all menu items that will fit on the screen, regardless of the menu's proximity to the top or bottom of screen.

How to use pop-up menus

Your application can define pop-up menus in two ways, as controls or menus.

If your application defines its pop-up menus as controls, using the `NewControl2` Control Manager tool call, then drawing, updating, resizing, and tracking are all handled by TaskMaster and `TrackControl` automatically. TaskMaster also deals 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`.

By contrast, if your application defines its pop-up menus 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 is 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 menu.

- *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 menu until it 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`	`SetMenuBar`
`CheckMItem`	`SetMenuFlag`
`CountMItems`	`SetMenuID`
`DeleteMItem`	`SetMenuTitle`
`DisableMItem`	`SetMenuTitle2`
`EnableMItem`	`SetMItem`
`GetMenuFlag`	`SetMItem2`
`GetMenuTitle`	`SetMItemBlink`
`GetMHandle`	`SetMItemFlag`
`GetMItem`	`SetMItemMark`
`GetMItemFlag`	`SetMItemName`
`GetMItemMark`	`SetMItemName2`
`GetMItemStyle`	`SetMItemStyle`
`GetMTitleWidth`	`SetMTitleWidth`
`InsertMItem`	

Each of the routines listed in Table 37-1 operates on the current menu bar. If your application defines its pop-up menus using `NewControl2`, then it must make the pop-up menu the current menu by issuing the `SetMenuBar` call and specifying the control handle for the pop-up menu 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

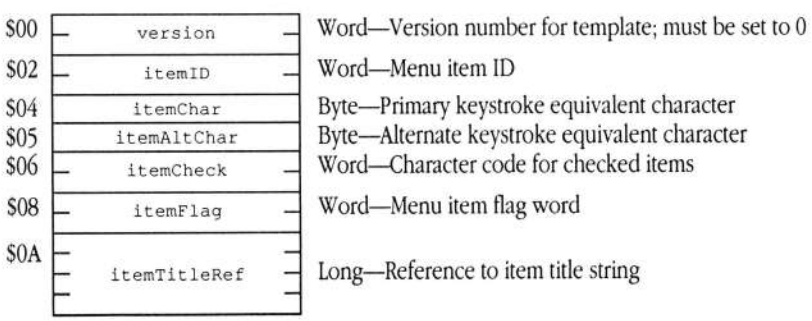
The new Menu Manager calls allow you to define menus using **templates,** analogous to the templates used by the `NewControl2` Control Manager tool call. These templates can be stored in fixed memory, in allocated memory referenced by handle, or in resources. When using any of these new calls, your program must specify the input data with the appropriate templates. The type of reference you use when you specify data for the Menu Manager governs how that data is later accessed. For example, if you originally specify the title for a menu with a handle, then anytime the system returns a reference to that menu title, the reference is a handle; similarly, your application must always refer to that title with a handle.

- *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, 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 37-7 shows the template that defines the characteristics of a menu item. Use it with new Menu Manager calls that require menu item templates.

Figure 37-7 `MenuItemTemplate` layout



`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 Volume 1 of 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 uppercase and lowercase ASCII codes for a particular character. If you have only 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 15–14	Defines the type of reference in `itemTitleRef`. 00 = Reference is by pointer 01 = Reference is by handle 10 = Reference is by 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 10–8	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 item 1 = Use XOR to highlight item
Reserved	bits 4–3	Must be set to 0.
`underline`	bit 2	Controls item underlining. 0 = Do not underline item 1 = Underline item

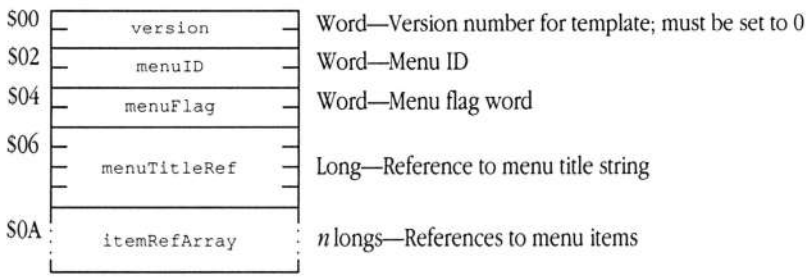
`italic`	bit 1	Indicates whether item is italicized. 0 = Not italicized 1 = Italicized
`bold`	bit 0	Indicates whether item is in boldface. 0 = Not bold 1 = Bold

`itemTitleRef` Reference to the title string of the 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 37-8 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 37-8** `MenuTemplate` layout



`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 Volume 1 of 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 15–14 Defines the type of reference in `menuTitleRef`.
00 = Reference is by pointer
01 = Reference is by handle
10 = Reference is by resource ID
11 = Invalid value

`itemRefType` bits 13–12 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 11–9 Must be set to 0.

`alwaysCallmChoose`	bit 8	Causes the Menu Manager to call a custom menu defProc `mChoose` routine even when the pointer 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 item 1 = Use XOR to highlight item
`custom`	bit 4	Indicates whether the menu is custom or standard. 0 = Standard menu 1 = Custom menu
`allowCache`	bit 3	Controls menu caching. 0 = Do not cache menu 1 = Menu caching allowed
Reserved	bits 2–0	Must be set to 0.

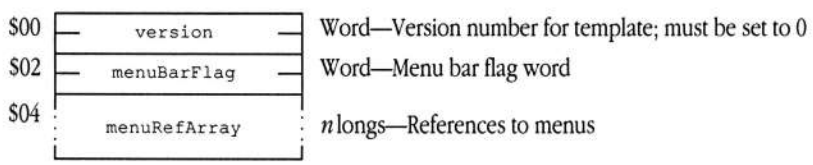
`menuTitleRef` Reference to the title string of the menu. The `titleRefType` bits in `menuFlag` indicate whether `menuTitleRef` contains a pointer, a handle, or a resource ID. If `menuTitleRef` is a pointer, then the title string must be a Pascal string. Otherwise, the Menu Manager can retrieve the string length from control information in the handle.

`itemRefArray` Array of references to the items in 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 37-9 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 37-9** `MenuBarTemplate` layout



`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 15–14 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 13–0 Must be set to 0.

`menuRefArray` Array of references to the menus in 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.

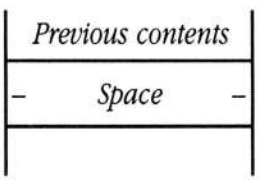
GetPopUpDefProc $3B0F

Returns a pointer to the control definition procedure for pop-up menus. Your application should not issue this call.

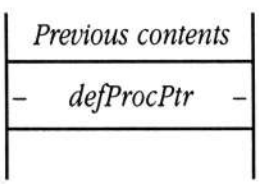
The system issues this call during Control Manager startup processing to obtain the address of the pop-up menu definition procedure.

Parameters

Stack before call



Stack after call



Errors None

C

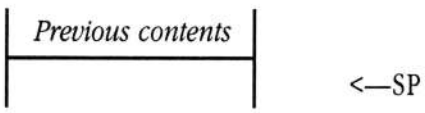
```
extern pascal Pointer GetPopUpDefProc();
```

HideMenuBar $450F

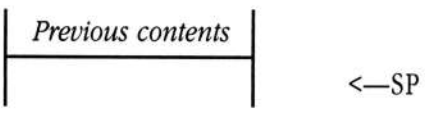
Hides the system menu bar by adding the menu bar to the desktop region. This call sets the invisible flag for the menu bar, resets scan lines 2 through 9 (which had been changed to display the colors of the Apple logo), and refreshes the desktop. The system ignores all subsequent calls to `DrawMenuBar` 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

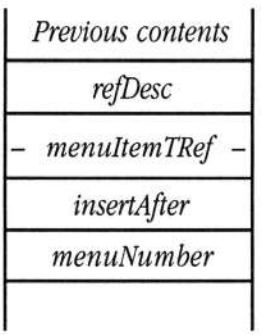
```
extern pascal void HideMenuBar();
```

InsertMItem2 $3F0F

Inserts an item into a menu after a specified menu item or at the top of the menu. This call accepts a menu item template as its input specification.

Parameters

Stack before call



Word—Defines type of reference in *menuItemTRef*

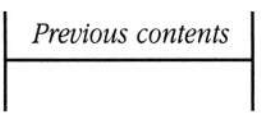
Long—Reference to menu item template

Word—ID of item after which to insert this item

Word—ID of menu into which to insert this item

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void InsertMItem2(refDesc,
          menuItemTRef, insertAfter, menuNumber);

Word      refDesc, insertAfter, menuNumber;
Long      menuItemTRef;
```

refDesc Indicates the type of reference stored in *menuTRef*. Valid values are

0 Reference is by pointer
1 Reference is by handle
2 Reference is by resource ID

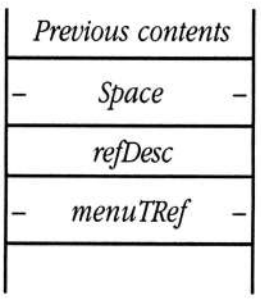
insertAfter Specifies ID of item after which the new item is to be inserted. To insert the new item at the top of the menu, set this field to 0. To insert the new item at the end, set this field to $FFFF.

NewMenu2 $3E0F

Allocates space for a menu list and its items. This call accepts a menu template as its input specification.

Parameters

Stack before call



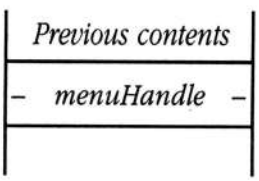
Long—Space for result

Word—Defines type of reference in *menuTRef*

Long—Reference to menu template

<—SP

Stack after call



Long—Handle for new menu

<—SP

Errors None

C

```
extern pascal Long NewMenu2(refDesc, menuTRef);

Word      refDesc;
Long      menuTRef;
```

refDesc Indicates the type of reference stored in *menuTRef*. Valid values are

0 Reference is by pointer
1 Reference is by handle
2 Reference is by resource ID

NewMenuBar2 $430F

Creates a menu bar using a menu bar template as its input specification.

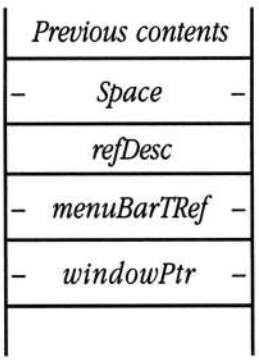
The upper-left corner of the default menu bar matches the port and is as wide as the screen. The bar is 13 pixels high.

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. 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



Long—Space for result

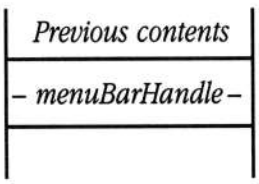
Word—Defines type of reference in *menuBarTRef*

Long—Reference to menu bar template

Long—Pointer to port for window; NIL for system menu bar

<—SP

Stack after call



Long—Handle for new menu bar

<—SP

Errors None

C

```
extern pascal Long NewMenuBar2(refDesc, menuBarTRef,
          windowPtr);

Word      refDesc;
Long      menuBarTRef;
Pointer   windowPtr;
```

refDesc Indicates the type of reference stored in *menuBarTRef*. Valid values are

0 Reference is by pointer
1 Reference is by handle
2 Reference is by 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 menu 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 it is displayed in the pop-up rectangle of unselected menus. You specify an item by its ID, that is, its relative position within the array of items for the menu. 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 menu and the top of the current selection, in global coordinates.

flag Flag word for the tool call. Bits are defined as follows:

Reserved	bits 15–7	Must be set to 0.
`type2`	bit 6	Indicates whether pop-up menu is type 1 or type 2. 0 = Type 1 menu (no white space added) 1 = Type 2 menu (white space added)
Reserved	bits 5–0	Must be set to 0.

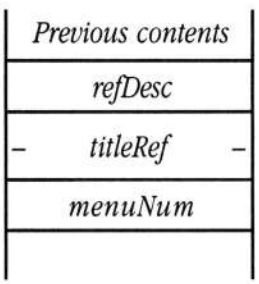
menuHandle The handle of the pop-up menu. The Menu Manager returned this value to your application from `NewMenu` or `NewMenu2`.

SetMenuTitle2 $400F

Specifies the title of a menu. The reference to the title string can be by pointer, handle, or resource ID.

Parameters

Stack before call



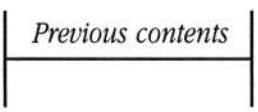
Word—Defines type of reference in *titleRef*

Long—Reference to title string of menu

Word—ID of menu to receive title

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void SetMenuTitle2(refDesc, titleRef,
          menuNum);

Word      refDesc, menuNum;
Long      menuItemTRef;
```

refDesc Indicates the type of reference stored in *titleRef*. Valid values are

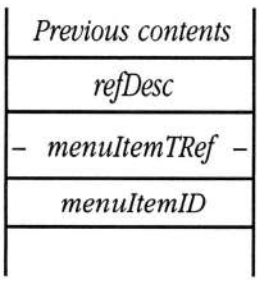
0 Reference is by pointer
1 Reference is by handle
2 Reference is by resource ID

SetMItem2 $410F

Specifies the name of a menu item. This call accepts a menu item template as its input specification.

Parameters

Stack before call



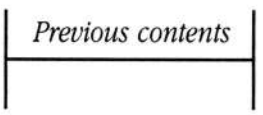
Word—Defines type of reference in *menuItemTRef*

Long—Reference to menu item template

Word—ID of item to be changed

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void SetMItem2(refDesc, menuItemTRef,
                menuItemID);

Word        refDesc, menuItemID;
Long        menuItemTRef;
```

refDesc Indicates the type of reference stored in *menuItemTRef*. Valid values are

0 Reference is by pointer
1 Reference is by handle
2 Reference is by resource ID

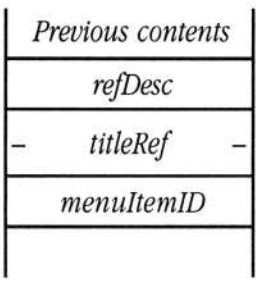
menuItemID Specifies the menu item to be changed. Note that you can change the item ID by specifying a different item number in the menu item template. The Menu Manager applies the item ID from the template to the item to be changed.

SetMItemName2 $420F

Specifies the name of a menu item. The reference to the title string can be by pointer, handle, or resource ID.

Parameters

Stack before call



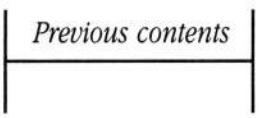
Word—Defines type of reference in *titleRef*

Long—Reference to menu item title

Word—ID of item to be changed

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void SetMItemName2(refDesc, titleRef,
          menuItemID);

Word      refDesc, menuNum;
Long      titleRef;
```

refDesc Indicates the type of reference stored in *titleRef*. Valid values are

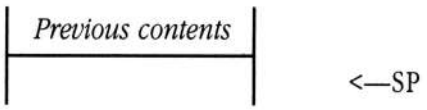
0 Reference is by pointer
1 Reference is by handle
2 Reference is by resource ID

ShowMenuBar $460F

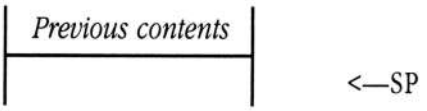
Reveals the system menu bar by subtracting the menu bar from the desktop region. This call also resets the `invisible` flag for the menu bar, resets scan lines 2 through 9 (to display the colors of the Apple logo), and draws the menu. Use the `HideMenuBar` call to make the menu bar invisible.

Parameters

Stack before call



Stack after call



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 for 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 performs 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 time-stamps, 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 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.

The following list summarizes the capabilities of the MIDI Tool Set. The tool calls are grouped according to function. Later sections of this chapter discuss the tool set in greater detail and define the precise syntax of the MIDI tool calls.

Routine	**Description**
Housekeeping routines	
`MidiBootInit`	Called only by the Tool Locator—must not be called by an application
`MidiStartUp`	Initializes the MIDI Tool Set for use by an application
`MidiShutDown`	Informs the MIDI Tool Set that an application is finished using its tool calls
`MidiVersion`	Returns the MIDI Tool Set version number

`MidiReset`	Called only when the system is reset—must not be called by an application
`MidiStatus`	Returns the operational status of the MIDI Tool Set

MIDI tool calls

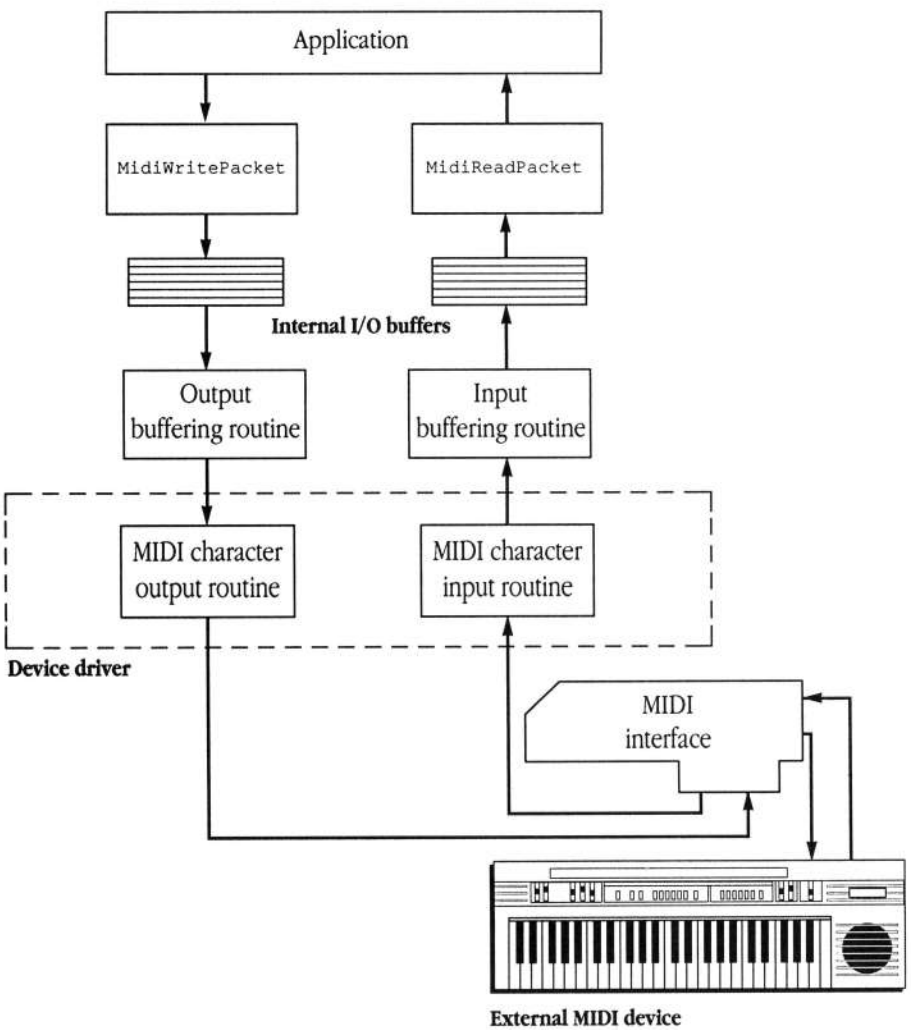
`MidiClock`	Controls operation of the optional time-stamp clock
`MidiControl`	Performs 18 MIDI control functions
`MidiDevice`	Selects, loads, and unloads MIDI device drivers
`MidiInfo`	Returns current operational information about the MIDI Tool Set
`MidiReadPacket`	Reads MIDI data from the tool set's internal buffers into a specified memory location
`MidiWritePacket`	Queues MIDI data for output

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 Tool Set functions.

Figure 38-1 illustrates some of the relationships between a typical MIDI application, the MIDI Tool Set, MIDI device drivers, and the Apple 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 their names 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 interruption 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 the 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. The tool set marks MIDI data packets with 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 you can budget processor resources intelligently. The input, output, and clock processes consume a great deal of processor time and 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.

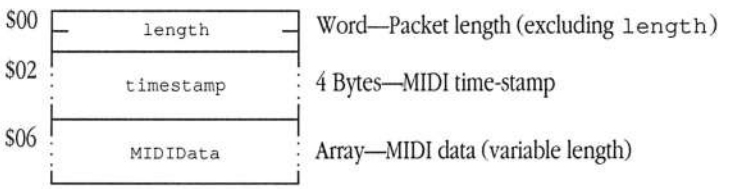
The Sound Tool Set must be started before your application can use the MIDI Tool Set.

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 they are equal to the value of the MIDI clock at the time the packet was received. The remaining bytes are the actual MIDI data.



A NoteOn command might look like this (in hexadecimal 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 follows.

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. The `MidiReadPacket` call 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 is returned to the application just as it is 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.

The `MidiReadPacket` call also returns real-time commands in raw mode unless a real-time vector is installed. See "`MidiControl` $0920" later in this chapter for more information.

MIDI Tool Set service routines

Your program can contain service routines that the MIDI Tool Set invokes under certain 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. Use the `miSetRTVec` function of the `MidiControl` tool call to set the vector to this routine.
Real-time error routine	Called when the MIDI Tool Set encounters an error during real-time processing. Use the `miSetErrVec` function of the `MidiControl` tool call to set the vector to this routine.
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

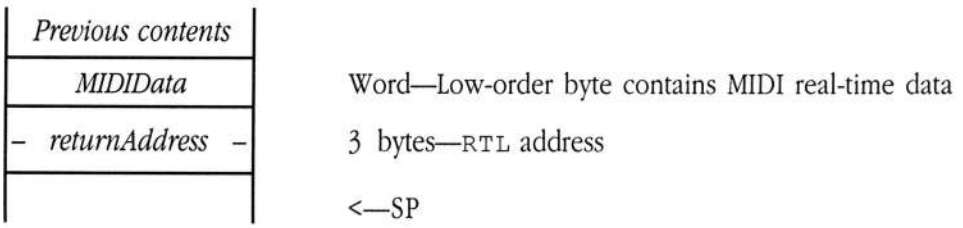
When the MIDI Tool Set receives MIDI real-time commands, it calls this service routine. The service routine must not enable interrupts, and if it runs longer than 300 microseconds, it must call the MIDI polling vector at least every 270 microseconds. The only MIDI calls that the service routine should make are `MidiReadPacket` and `MidiWritePacket`.

Real-time MIDI data is 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.

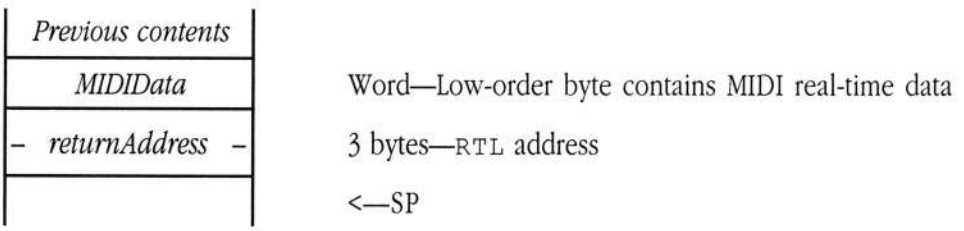
You set the vector to this routine with the `miSetRTVec` function of the `MidiControl` tool call.

Parameters

Stack before call



Stack after call



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 runs 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.

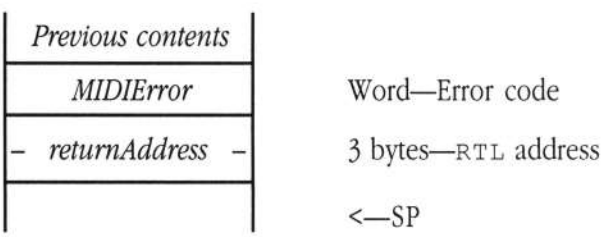
Use the `miSetErrVec` function of the `MidiControl` tool call to set the vector to this routine.

The service routine may receive the following error codes:

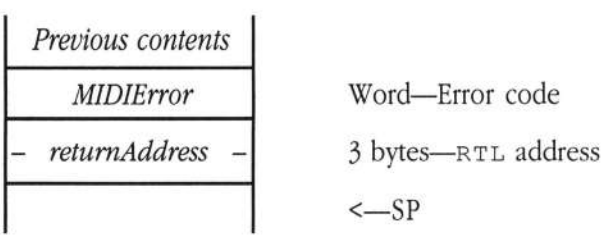
$200A	`miClockErr`	MIDI clock wrapped to 0.
$2084	`miDevNoConnect`	No connection to MIDI interface.

Parameters

Stack before call



Stack after call



Input data routine

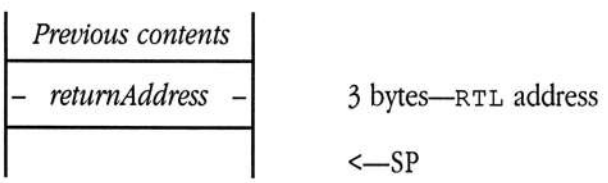
The MIDI Tool Set calls this routine during processing of the `miStartInput` function of the `MidiControl` tool call when the first packet is available in a previously empty input buffer. The service routine must not enable interrupts, and if it runs longer than 300 microseconds, it must call `MidiInputPoll` 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 calls 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.

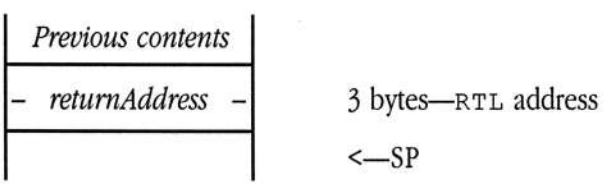
You set the vector to this routine when you issue the `miStartInput` function of the `MidiControl` tool call.

Parameters

Stack before call



Stack after call



Output data routine

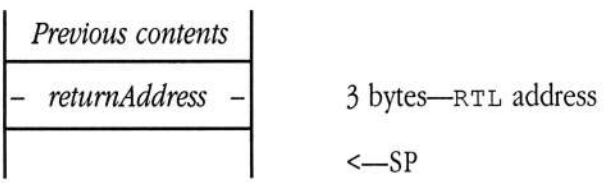
The MIDI Tool Set calls this routine during processing of the `miStartOutput` function of the `MidiControl` tool call when the output buffer becomes completely empty. The service routine must not enable interrupts, and if it runs longer than 300 microseconds, it must call the MIDI polling vector at least 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.

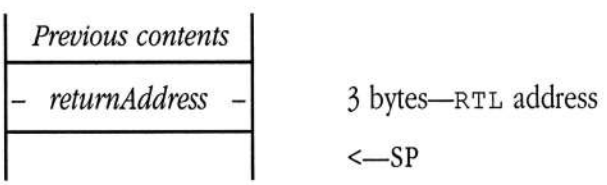
You set the vector to this routine when you issue the `miStartOutput` function of the `MidiControl` tool call.

Parameters

Stack before call



Stack after call



Starting up the MIDI Tool Set

The `MidiStartUp` call takes two 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 pressed 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 cause problems. 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 set 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 was 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 is present returns in from 8 to 30 microseconds, and when data is 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 complex 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 process MIDI data correctly.

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 interfere 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 32 KB, so your application may have to run the MIDI interface in raw mode (rather than packet mode) to support system-exclusive messages longer than 16 KB.

You might want to keep statistics on the maximum number of data bytes in the input buffer so that your application can adjust the input buffer size intelligently. Several MIDI Tool Set calls return information you can use for this purpose; see "MIDI Tool Calls" later 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 nonbuffered 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

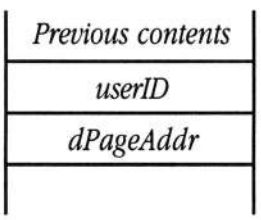
```
extern pascal void MidiBootInit();
```

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

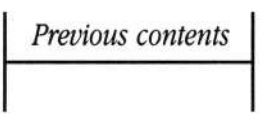


Word—Application user ID (for the Memory Manager)

Word—Beginning of MIDI direct-page space

<—SP

Stack after call



<—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 three 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. These actions take place immediately, so the application should take any necessary steps to see that all 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

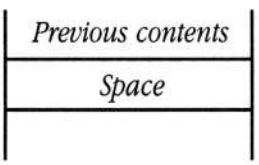
```
extern pascal void MidiShutDown();
```

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

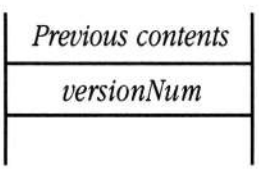
Stack before call



Word—Space for result

<—SP

Stack after call



Word—MIDI tools version number

<—SP

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, allowing 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();
```

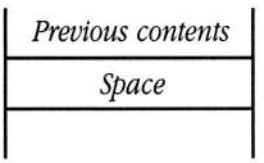
MidiStatus $0620

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

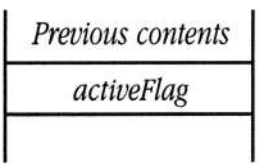
Stack before call



Word—Space for result

<—SP

Stack after call



Word—Boolean; TRUE if the tool set is active

<—SP

Errors None

C

```
extern pascal Boolean 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.*

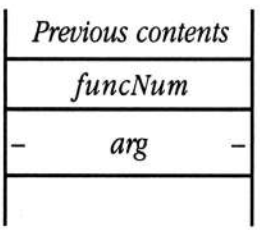
The routines used to work with the MIDI Tool Set are `MidiClock`, `MidiControl`, `MidiDevice`, `MidiInfo`, `MidiReadPacket`, and `MidiWritePacket`. Four of these calls are multifunction calls, which perform different actions depending on a control parameter passed to them. The workhorse of the group 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, allowing MIDI data to be sent and received 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

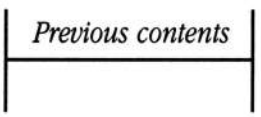


Word—Specifies `MidiClock` function number

Long—Argument passed to `MidiClock` function

<—SP

Stack after call



<—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.

`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 when the clock frequency is set to 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, and MIDI data could be lost if the clock is started 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).

The clock frequency affects the rate of playback. Unless you intend to vary the tempo during playback, be careful to set the clock frequency to the same value that was used when the sequence was recorded.

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).

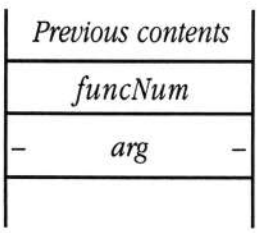
MidiControl $0920

Performs 18 different control functions required by the MIDI Tool Set.

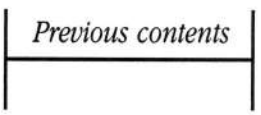
The *funcNum* parameter selects which function is to be performed, and the *arg* parameter passes any argument required by that function.

Parameters

Stack before call



Stack after call



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

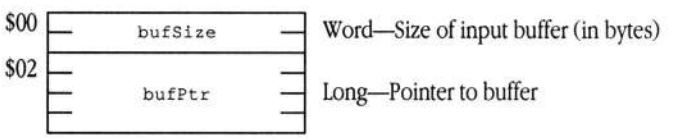
1 `miSetErrVec`

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



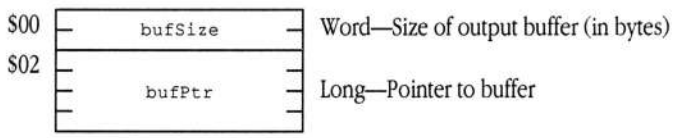
If the `bufPtr` parameter is set to 0, the MIDI Tool Set will allocate the input buffer. If the `bufSize` parameter is set to 0, the MIDI tools will allocate a buffer 8 KB in size. Note that these parameters are independent; your program may set either one of them to 0. If the application allocates the buffer, it must be nonpurgeable, must exist 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



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, must be 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.

4 `miStartInput`

Starts an interrupt-driven process that reads MIDI data into the MIDI Tool Set's input buffer. Data being received when this call is made is discarded until the first MIDI status byte is received. An application can retrieve this 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 no complete packet is 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" earlier 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. This process is inherently less accurate and uses more processor time. The default delay value is 0, or no delay.

Many synthesizers may need a delay value to process the many high-speed NoteOff commands generated by the `miFlushOutput` function correctly.

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. For this reason, 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

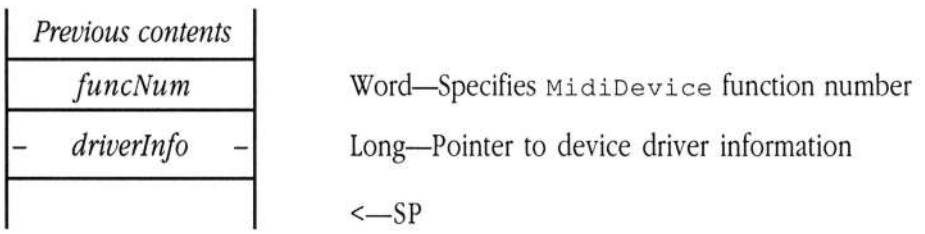
MidiDevice $0A20

Allows an application to select, load, and unload device drivers for use with the tools. The `MidiDevice` tool call 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.

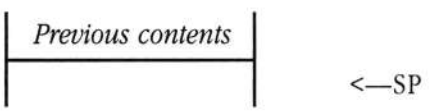
The call interprets the *driverInfo* parameter as the address of the driver to be loaded. The *funcNum* parameter specifies whether the driver is to be loaded or unloaded.

Parameters

Stack before call



Stack after call



Errors See the `MidiDevice` function descriptions.

C

```
extern pascal void MidiDevice(funcNum, driverInfo);

Word       funcNum;
Pointer    driverInfo;
```

funcNum Specifies the `MidiDevice` function to be performed.

`0` **Not yet implemented**

1 `miLoadDrvr`

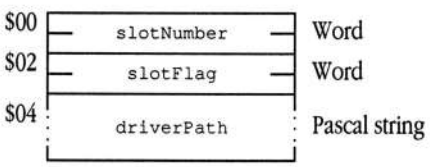
Loads the specified device driver into memory, after shutting down and unloading any previously loaded device drivers. It then initializes the newly loaded driver. The *driverInfo* parameter points to a device driver record, which specifies a device driver to be loaded.

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.

driverInfo

The record pointed to by the *driverInfo* parameter contains device driver information.



`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 the 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.

Errors None

2 `miUnloadDrvr`

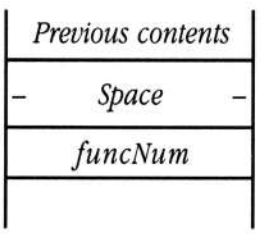
Shuts down and unloads the currently loaded device driver. Terminates MIDI transmission and reception if they are currently active. Releases memory occupied by the device driver.

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

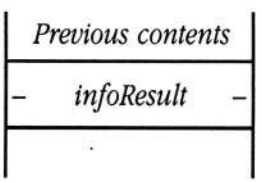


Long—Space for result

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 4-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, 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.

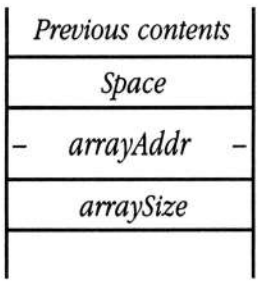
Errors None

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



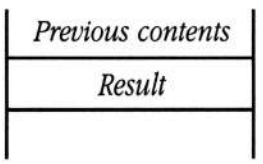
Word—Space for result

Long—Pointer to buffer for received data

Word—Length, in bytes, of the receive buffer

<—SP

Stack after call



Word—Number of bytes actually returned

<—SP

Errors

$2001	`miPacketErr`	Incorrect packet length received.
$2002	`miArrayErr`	Array size invalid.
$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(arrayAddr,
          arraySize);

Pointer   arrayAddr;
Word      arraySize;
```

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" earlier in this chapter.

The `MidiWritePacket` call returns within one-fiftieth 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 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. For this reason, 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.

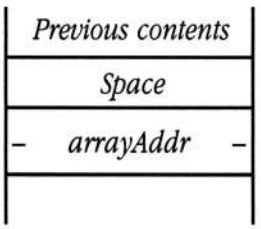
If the MIDI clock is stopped, then all packets with a time-stamp less than or equal to the value of the clock are immediately transmitted, and all packets with a value greater than the clock remain in the buffer unless the clock is restarted and its value becomes greater than that of the time-stamps.

Two special time-stamp values override normal output buffer processing, irrespective of MIDI clock state. Any packet with a time-stamp of 0 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

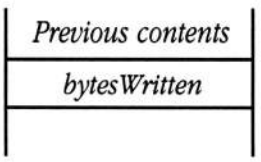


Word—Space for result

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(arrayAddr);

Pointer   arrayAddr;
```

MIDI Tool Set error codes

Table 38-1 lists the error codes that may be returned by MIDI Tool Set calls.

■ **Table 38-1** MIDI Tool Set error codes

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 Tool Set is in Volume 1, Chapter 14 of the *Apple IIGS Toolbox Reference.*

Error corrections

This section documents errors in Chapter 14, "Miscellaneous Tool Set," in Volume 1 of the *Toolbox Reference.*

- On page 14-58 of Volume 1 of the *Toolbox Reference,* Figure 14-3 shows the low-order bit of the user ID as 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 two errors. In the code to clear the 1-second IRQ source, the second instruction reads

  ```
  TSB   $C032
  ```

  This instruction should read

  ```
  TRB   $C032
  ```

  In addition, preceding this instruction the following code should be inserted

  ```
  PEA   $0000
  PLB
  PLB
  ```

  These three instructions allow the code to reliably access the appropriate location in bank zero memory. These same three instructions should also be inserted in the code shown on page 14-29, immediately preceding the `STA` instruction.
- 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 a pointer to a pointer. Creating *startHandle* as a handle will cause unpredictable system behavior.
- Throughout Chapter 14 of the *Toolbox Reference* the value of the signature word for Miscellaneous Tool Set data structures is given as $5AA5 and $A55A. Signature words are *always* $A55A, *never* $5AA5.

Clarification

Note that the `ClrHeartBeat` tool call removes all tasks from the Heartbeat Interrupt Task queue, including those installed by system software. Consequently, only system software should issue the `ClrHeartBeat` tool call.

New features of the Miscellaneous Tool Set

The Miscellaneous Tool Set now supports a number of new features. This section discusses these new features in detail.

- The `ClearHeartBeat` and `DeleteHeartBeat` calls turn off the interrupts that occur every one-sixtieth 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.
- The `SetVector` and `GetVector` calls 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:* The `SetVector` call no longer validates the input vector number. Therefore, you must be extremely careful when using this call to avoid corrupting memory.

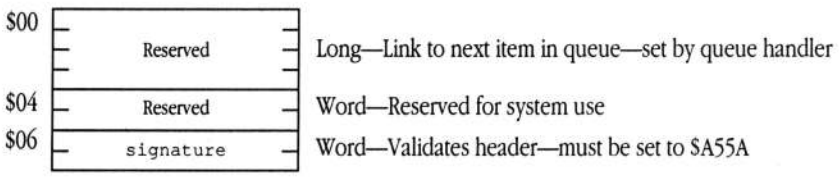
Queue handling

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 elements to 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 use `AddToQueue` to add this first element to the queue.

Figure 39-1 shows the format of the queue header.

■ **Figure 39-1** Queue header layout



Application data immediately follows the header.

See "New Miscellaneous Tool Set Calls" later in this chapter for details on `AddToQueue` and `DeleteFromQueue`.

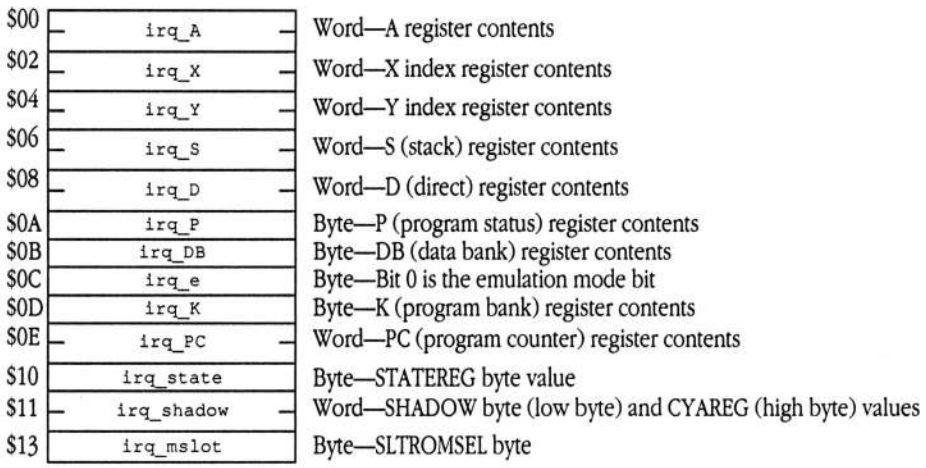
Interrupt state information

The Miscellaneous Tool Set now provides a set of calls (`GetInterruptState` and `SetInterruptState`) 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



New Miscellaneous Tool Set calls

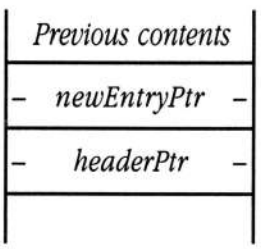
The following sections introduce several new Miscellaneous Tool Set calls.

AddToQueue $2E03

Adds the specified entry to a queue.

Parameters

Stack before call

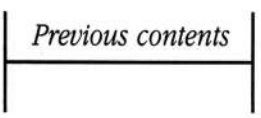


Long—Pointer to element to add to queue

Long—Pointer to first queue element

<—SP

Stack after call



<—SP

Errors

$0381	`invalidTag`	Signature value invalid in element header.
$0382	`alreadyInQueue`	Specified element already in queue.

C

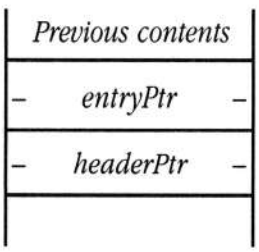
```
extern pascal void AddToQueue(newEntryPtr,
          headerPtr);

Pointer   newEntryPtr, headerPtr;
```

DeleteFromQueue $2F03

Deletes a specified element from a queue.

Parameters

Stack before call

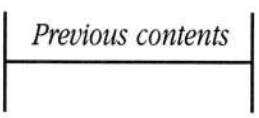


Long—Pointer to element to delete from queue

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 Set call because the loader is not in directly accessible memory (it is in the bank 1 language card, which may or may not be addressable at any given time).

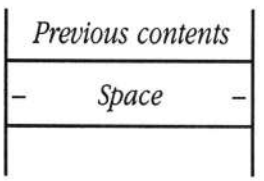
Your program would use this call in conjunction with the `ResourceConverter` tool call (see Chapter 45, "Resource Manager," in this book). For example, the Control Manager issues the following call during its startup processing:

```
ResourceConverter(GetCodeResConverter(),
                  rCtlDefProc,
                  LogConverterIn+SysConverterList);
```

After this call is issued, all future calls to the Resource Manager to load resources of type `rCtlDefProc` use `GetCodeResConverter` to bring the resource into memory. Note that this routine does not preserve the memory attributes of the converted resource (for more information on resource converters, see Chapter 45, "Resource Manager," in this book).

Parameters

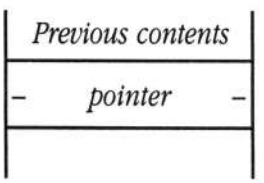
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to code resource converter routine

<—SP

Errors None

C

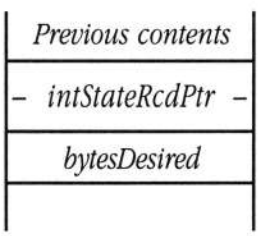
```
extern pascal Pointer GetCodeResConverter();
```

GetInterruptState $3103

Copies the specified number of bytes into a specified input interrupt state record from the system interrupt variables. For information about record layouts, see "Interrupt State Information" earlier in this chapter. 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

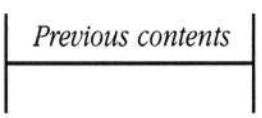


Long—Pointer to interrupt state record

Word—Number of bytes to copy from system to record

<—SP

Stack after call



<—SP

Errors None

C

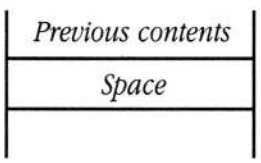
```
extern pascal void GetInterruptState(intStateRcdPtr,
          bytesDesired);

Pointer   intStateRcdPtr;
Word      bytesDesired;
```

GetIntStateRecSize $3203

Returns the size (in bytes) of the interrupt state record. This call allows applications to work with extended interrupt state records.

Parameters

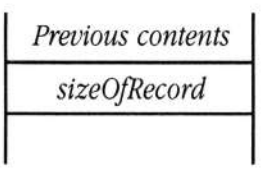
Stack before call



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 for use only 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

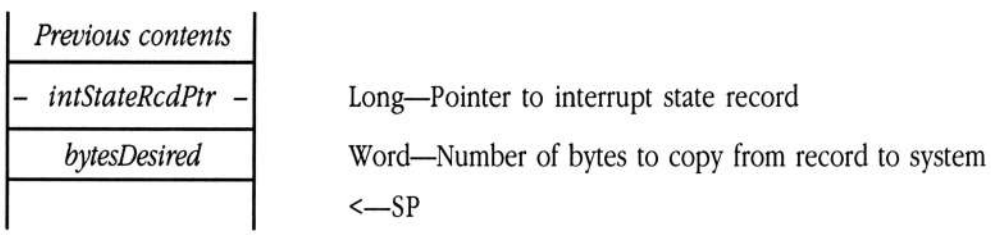
This call is for use only by system firmware.

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



Stack after call

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 Tool Set 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.

The following list summarizes the capabilities of the Note Sequencer. The tool calls are grouped according to function. Later sections of this chapter discuss the tool set in greater detail and define the precise syntax of the Note Sequencer tool calls.

Routine	Description
Housekeeping routines	
`SeqBootInit`	Called only by the Tool Locator—must not be called by an application
`SeqStartUp`	Initializes the Note Sequencer for use by an application and establishes the values of many important operational parameters
`SeqShutDown`	Informs the Note Sequencer that an application is finished using its tool calls
`SeqVersion`	Returns the Note Sequencer version number
`SeqReset`	Called only when the system is reset—must not be called by an application
`SeqStatus`	Returns the operational status of the Note Sequencer

Note Sequencer tool calls

`ClearIncr`	Sets the increment value to 0, halting the current sequence
`GetLoc`	Returns operational information about the current sequence
`GetTimer`	Returns the value of the Note Sequencer tick counter
`SeqAllNotesOff`	Switches off all notes currently playing, but does not halt the sequence
`SetIncr`	Sets the increment value
`SetInstTable`	Sets the current instrument table
`SetTrkInfo`	Assigns instruments to tracks
`StartInts`	Enables Note Synthesizer and Note Sequencer interrupts
`StartSeq`	Instructs the Note Sequencer to start playing a sequence that contains absolute addresses
`StartSeqRel`	Instructs the Note Sequencer to start playing a sequence that contains relative addresses
`StepSeq`	Increments the Note Sequencer tick counter
`StopInts`	Disables Note Synthesizer and Note Sequencer interrupts
`StopSeq`	Stops the current sequence

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 access 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" later 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 control the execution of a sequence explicitly 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 melody played by a violin accompanied by a viola and a flute. The three instruments often play at once, sounding different notes. The Note Sequencer, however, always plays notes in sequence, one after another, no matter how many instruments are used to play the notes.

A chord, which is a group of different notes played at the same time, is executed by the Note Sequencer as a series of discrete notes started 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 were to wait for each note to finish before beginning the next one, the resulting passage would be three distinct notes of equal length—not the intended result. 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.

Setting the `chord` bit to 1 in a note command indicates that the next note should sound a chord with the current one. If, by contrast, the `delay` bit is set to 1, the current note is 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," earlier 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 to 1 when the `SeqStartUp` call is made, then MIDI is enabled. If a particular track is to use MIDI, you must use the `SetTrkInfo` call to enable it for that track. 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.

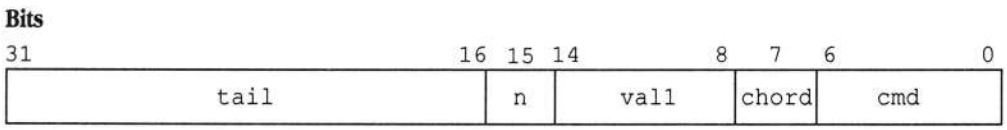
If all the appropriate flags—the *mode* flag of `SeqStartUp`, the *trackName* flag of `SetTrkInfo`, and the command or tool call flag—are enabled, then MIDI commands are sent to external MIDI devices. This arrangement is designed to provide flexibility in execution. You could, for example, play only 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 *sequence items,* or *seqItems.* These 32-bit items contain information that the Note Sequencer needs to classify commands 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 `setVibratoDepth` 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.

`val1` The meaning of the `val1` field depends on the command being issued.

`n` The `n` bit identifies note commands. If bit `n` is set to 1, the seqItem is a note command.

`tail` The format of the `tail` field depends on the command type. It contains two or more subfields 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 applications to gain control at the end of a sequence and whenever 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 no custom routine of the appropriate type is 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 the 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 passes to the routine specified by *compRoutine*.

Whenever it encounters an error during sequence processing, the Note Sequencer tries to call the error handler for the sequence. A useful function for 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.

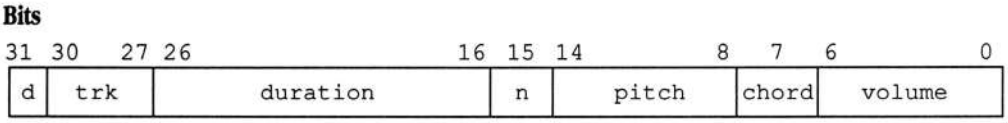
The Note Sequencer passes error codes to the error handler in the A register. In step mode, the Note Sequencer both reports the error condition to the error handler and posts it in the A register at the completion of the call to `StepSeq`. In interrupt mode, the Note Sequencer only reports the error 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. These commands are not the same as the Note Synthesizer `NoteOn` and `NoteOff` tool calls. 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 and 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). The format of note commands is shown in Figure 40-2.

■ **Figure 40-2** Note command format



`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 `d` 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" later in this chapter for details).

`n` 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 duration of the note to be played by the Note Sequencer. Values may range from 0 to 2047 and indicate the duration of the note in number of ticks.

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.
`d`	If the `d` (delay) 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.

`noteOff` command

Stops a note that was previously started with a `noteOn` command.

`volume`	Note volume = 0
`chord`	Set if the note is part of a chord
`pitch`	127–0; must be the same as matching `noteOn`
`n`	1
`duration`	0
`trk`	15–0; must be the same as matching `noteOn`
`d`	0

`noteOn` command

Starts a note playing.

`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
`n`	1
`duration`	0
`trk`	15–0
`d`	0

Filler notes

Filler notes are used to create silences 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. It not only fails to play any notes until the delay period has elapsed but also fails to perform other services, such as turning notes off. Using delays to create rests could thus lead to unpredictable behavior in the creation of sequences.

An alternative approach is to use filler notes. A filler note is simply a note command with a pitch value of 0. The Note Sequencer plays such a note as though it were an ordinary note but does not produce a tone. You can therefore use filler notes to fill out rests at points where you might have supposed a delay would be needed. For example, a passage may contain a chord consisting of notes of different duration, followed by a run of other notes. In this case, you need 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.

`fillerNote` command

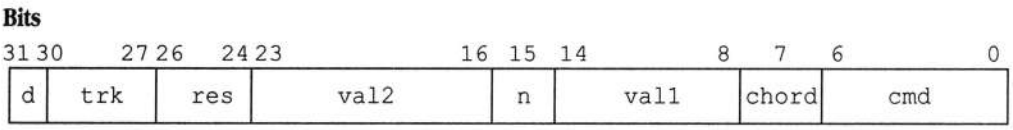
Creates silences in musical sequences.

`volume`	0
`chord`	1
`pitch`	Pitch value = 0
`n`	1
`duration`	Desired delay time
`trk`	0
`d`	Set to 1 if a delay is desired

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. The format of control commands is shown in Figure 40-3.

- **Figure 40-3** Control command format



`cmd`	Command number.
`chord`	Should be set to 1 in a control command. Setting the `chord` bit to 0 can sometimes cause unwanted delays in the playing of a sequence.
`val1`	Contains data specific to each command.
`n`	Always set this bit to 0 for control commands. Setting the `n` bit to 1 causes the seqItem to be processed as a note command instead of a control command.
`val2`	Contains data specific to each command.
`res`	Reserved for control. 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.
`d`	Should always be set to 0 in control commands, since they have no duration.

`callRoutine` 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.

`cmd`	30
`chord`	1
`val1`	0
`n`	0
bits 16–23	Low-order byte of routine address
bits 24–31	High-order byte of routine address

On entry, interrupts are disabled, and very little stack space remains. The Note Sequencer saves its registers before issuing the call. However, because the direct-page and data bank registers are set for the Note Sequencer, your routine code must change these 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

The Note Sequencer's equivalent of a jump or goto command in a conventional programming language. Execution of seqItems continues 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. Such a jump will produce unpredictable results.

`cmd`	3
`chord`	1
`val1`	`val1` is the high 7 bits of the destination
`n`	0
`val2`	`val2` is the low 8 bits of the destination
`res`	0
`trk`	not used
`d`	0

Note that this command causes a jump in the sequence being processed. To jump to executable code from a sequence, use the `callRoutine` command.

pitchBend 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 `pitchBend` commands to achieve the smooth portamento usually associated with a pitch bend.

`cmd`	0
`chord`	1
`val1`	Pitch wheel position. Values greater than 64 specify sharp pitch bend; values less than 64 specify flat; intervals are expressed in fractions of the current pitch bend range
`n`	0
`val2`	No significance in the `pitchBend` command; the `val2` field should always be set to 0 for `pitchBend`
`res`	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
`d`	0

The `res` 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.

programChange 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.

`cmd`	5
`chord`	1
`val1`	Instrument index from instrument table
`n`	0
`val2`	New MIDI program number, if the sequence is using MIDI
`res`	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
`d`	0

If MIDI is enabled and the `res` 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

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.

`cmd`	1
`chord`	1
`val1`	New increment; the value may vary from 0 to 127
`n`	0
`val2`	0
`res`	0
`trk`	0
`d`	0

turnNotesOff 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.

cmd	2
chord	1
val1	0
n	0
val2	0
res	0
trk	0
d	0

setVibratoDepth 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.

cmd	4
chord	1
val1	The new value for vibrato depth; the value may vary from 0 to 127
n	0
val2	Control number if a MIDI command is generated
res	Specifies MIDI usage; legal values are 0 Internal and MIDI vibrato 1 Internal only
trk	Track number
d	0

If MIDI support has been enabled and the `res` 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.

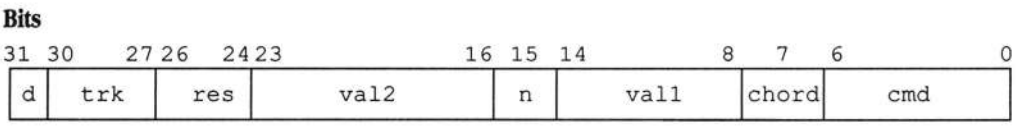
Register commands

Register commands provide the Note Sequencer with program control capabilities. The Note Sequencer maintains eight 8-bit registers 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` and `setRegister` 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 registers; these registers 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 shows the format of register commands.

- **Figure 40-4** Register command format



`cmd`	Command number.
`chord`	Should be set to 1 in a register command. Setting the `chord` bit to 0 can sometimes cause unwanted delays in the playing of a sequence.
`val1`	Contains data specific to each command. Generally specifies the register number for the command.
`n`	Always set this bit to 0 for register commands. Setting the `n` bit to 1 causes the seqItem to be processed as a note command instead of a register command.
`val2`	Contains data specific to each command.
`res`	Reserved for control. These bits should always be set to 0 unless otherwise specified.
`trk`	Always set to 0 for register commands.
`d`	Should always be set to 0 in register commands, since they have no duration.

`decRegister` command

Decrements the value of the specified register. If the value is 0 when the command is executed, the register's value will wrap to $FF.

`chord`	1
`val1`	Low 3 bits contain the register number
`n`	0
`val2`	0
`res`	0
`trk`	0
`d`	0

`ifGo` command

Tests the specified register for the specified value. If the register 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 the sequence. The `ifGo` command does not check the bounds of the offset provided. For this reason, the value must be valid, or the effects will be unpredictable.

`cmd`	7
`chord`	1
`val1`	Low 3 bits contain the register number High 4 bits contain the value
`n`	0
`val2`	Offset: –128 to +127 seqItems
`res`	0
`trk`	0
`d`	0

`incRegister` command

Increments the value of the specified register.

`cmd`	8
`chord`	1
`val1`	Low 3 bits contain the register number
`n`	0
`val2`	0
`res`	0
`trk`	0
`d`	0

`setRegister` command

Sets the specified register to the specified value.

`cmd`	6
`chord`	1
`val1`	Low 3 bits contain the register number High 4 bits contain the value
`n`	0
`val2`	0
`res`	0
`trk`	0
`d`	0

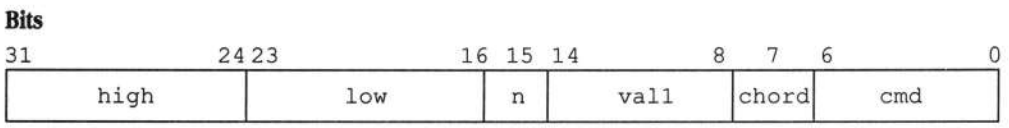
MIDI commands

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 to 1 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, which is not described in this documentation. See Figure 40-5 for the format of MIDI commands.

- **Figure 40-5** MIDI command format



`cmd`	Command number.
`chord`	The `chord` bit should be set to 1 in a MIDI command. Setting the `chord` bit to 0 can sometimes cause unwanted delays in playing a sequence.
`val1`	Contains data specific to each command.
`n`	Always set this bit to 0 for MIDI commands. Setting the `n` bit 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.

midiChnlPress command

Sends a MIDI Channel Pressure command to the channel specified in `val1`. The new pressure value is specified by the `low` byte.

`cmd`	15
`chord`	1
`val1`	Bits 8 through 11 specify the MIDI channel number ($0–$0F)
`n`	0
`low`	Channel pressure
`high`	0

midiCtlChange command

Sends a MIDI Control Change command to the channel specified in `val1`. The control number is specified in the `low` byte, and the new value of the control in the `high` byte.

`chord`	1
`val1`	Bits 8 through 11 specify the MIDI channel number ($0–$0F)
`n`	0
`low`	Control number
`high`	Control value

midiNoteOff command

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.

`cmd`	10
`chord`	1
`val1`	Bits 8 through 11 specify the MIDI channel number ($0–$0F)
`n`	0
`low`	Note number
`high`	Velocity

midiNoteOn command

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.

`cmd`	11
`chord`	1
`val1`	Bits 8 through 11 specify the MIDI channel number ($0–$0F)
`n`	0
`low`	Note number
`high`	Velocity

midiPitchBend command

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.

`cmd`	16
`chord`	1
`val1`	Bits 8 through 11 specify the MIDI channel number ($0–$0F)
`n`	0
`low`	Pitch bend least significant byte
`high`	Pitch bend most significant byte

midiPolyKey command

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.

`cmd`	12
`chord`	1
`val1`	Bits 8 through 11 specify the MIDI channel number ($0–$0F)
`n`	0
`low`	Note number
`high`	Key pressure

midiProgChange command

Sends a MIDI Program Change command to the channel specified in `val1`. The program number is specified in the `low` byte.

`cmd`	14
`chord`	1
`val1`	Bits 8 through 11 specify the MIDI channel number ($0–$0F)
`n`	0
`low`	Program number
`high`	0

midiSelChnlMode command

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.

`cmd`	17
`chord`	1
`val1`	Bits 8 through 11 specify the MIDI channel number ($0–$0F)
`n`	0
`low`	First data byte
`high`	Second data byte

midiSetSysExl command

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, whereas the low word is specified by the `midiSysExclusive` command. The `midiSetSysExl` command must precede the `midiSysExclusive` command. See the following discussion of that command for more information about the format and content of the MIDI packet.

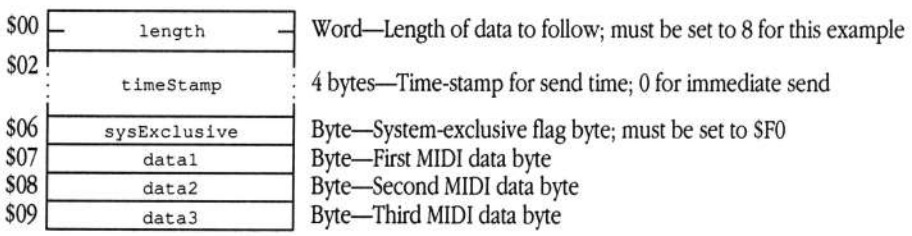
`cmd`	21
`chord`	1
`val1`	0
`n`	0
`low`	Low byte of high word
`high`	High byte of high word

midiSysExclusive 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, whereas the high word is specified by the `midiSetSysExl` command. The `midiSetSysExl` command must precede the `midiSysExclusive` command.

`cmd`	18
`chord`	1
`val1`	0
`n`	0
`low`	Least significant byte of low word of MIDI packet address
`high`	Most significant byte of low word of MIDI packet address

Here is an example of a 3-byte system-exclusive command:



midiSysCommon command

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.

`cmd`	19
`chord`	1
`val1`	Bits 10 through 8—low nibble of status byte value varies from 1 through 7 Bits 12 and 11—number of data bytes: 00 = 0 data bytes 01 = 1 data byte 10 = 2 data bytes 11 = Invalid value
`n`	0
`low`	First data byte
`high`	Second data byte (if appropriate)

`midiSysRealTime` command

Sends a MIDI System Real-Time command. The real-time number is specified in the low 3 bits of the `low` byte.

`cmd`	20
`chord`	1
`val1`	0
`n`	0
`low`	Real-time number ($01–$07)
`high`	0

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. Because 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 long word 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 long word 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, the program 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 phrase done flag marking the end of the 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.

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. Here is an 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                ; don't 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
                _SetInstTable
                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'                           ; all notes off
                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();
```

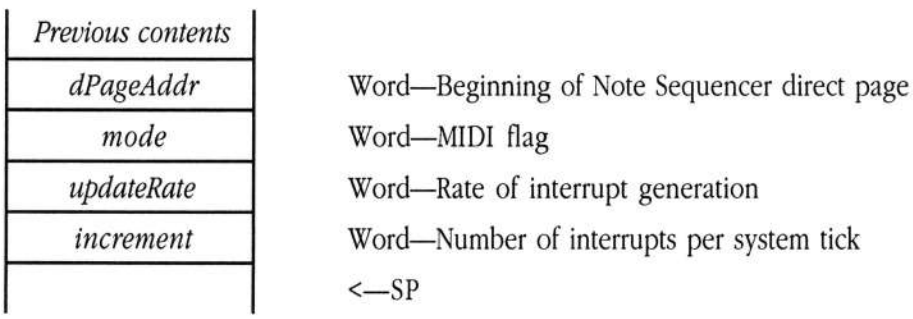
SeqStartUp $021A

Starts up the Note Sequencer and performs all the necessary initializations for the tool set. This call 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.

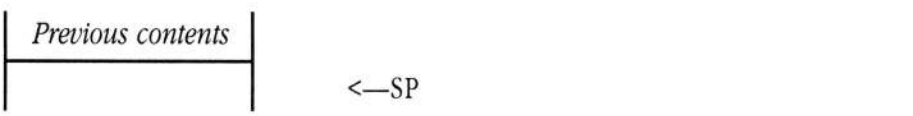
Your application must make sure that the MIDI Tool Set has been started before issuing this call.

Parameters

Stack before call



Stack after call



Errors

$1A03	`startedErr`	The Note Sequencer is already started.
$1A07	`nsWrongVer`	The version of the Note Synthesizer 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 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 Hz, or every 5 milliseconds. A value of 250 means that interrupts will run at 100 Hz, 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 * *updateRate*) / (*increment* * T)

where 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 the tempo of a sequence and smoother envelopes. However, a higher *updateRate* also requires more processor time.

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* to arrive at the duration you want.

Once you have set the *updateRate* value, it remains in effect; you can change it only 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 terminating.

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

Errors

$1923	`nsNotInit`	The Note Synthesizer was not started.
$1A05	`noStartErr`	The Note Sequencer was not started.
$0812	`noSAppInitErr`	The Sound Tool Set was not started.

C

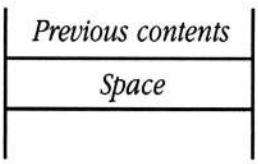
```
extern pascal void SeqShutDown();
```

SeqVersion $041A

Returns the version number of the Note Sequencer that is currently in use. Refer to Appendix A, "Writing Your Own Tool Set," in Volume 2 of the *Toolbox Reference* for information on the format and content of the returned *versionNum* value.

Parameters

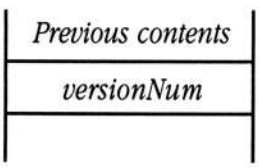
Stack before call



Word—Space for result

<—SP

Stack after call



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();
```

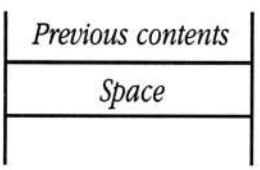
SeqStatus $061A

Returns a Boolean flag indicating whether or not the Note Sequencer is active. If the tool set is active, the flag is TRUE (nonzero); otherwise, it is FALSE (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

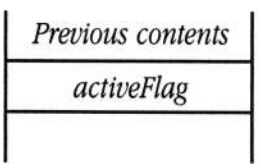
Stack before call



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.

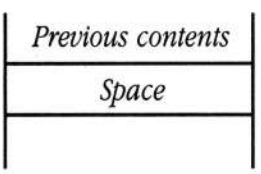
`ClearIncr` $0A1A

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

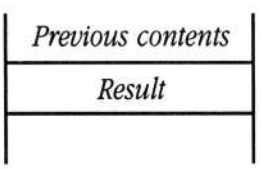
Stack before call



Word—Space for result

<—SP

Stack after call



Word—Previous increment value

<—SP

Errors None

C

```
extern pascal Word ClearIncr();
```

GetLoc $0C1A

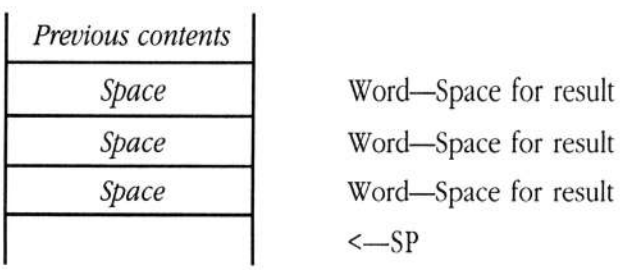
Returns certain information about the sequence that is playing. This call provides an index to the seqItem that is executing, the current pattern, and the nesting level. The nesting level indicates how deeply control has passed into a structure with phrases nested within phrases. A nesting level value 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:

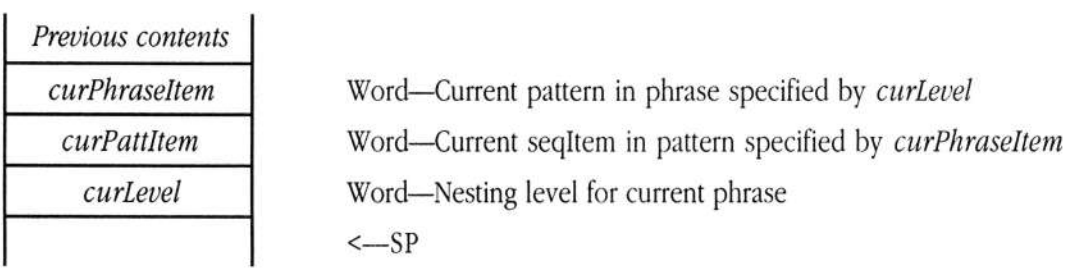
curPattItem = 3
curPhraseItem = 1
curLevel = 1

Parameters

Stack before call



Stack after call



Errors None

C

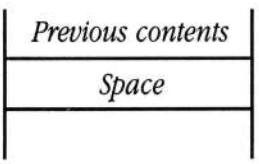
```
extern LocRec GetLoc();
```

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

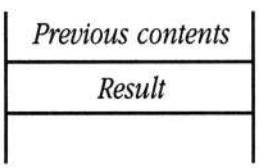
Stack before call



Word—Space for result

<—SP

Stack after call



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 silence all instrument voices temporarily while a sequence is active. If the high bit of the *mode* parameter to the `SeqStartUp` call was set to 1, 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

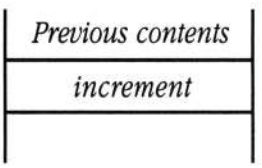
```
extern pascal void SeqAllNotesOff();
```

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* parameter is set to 0, the sequence will halt.

Parameters

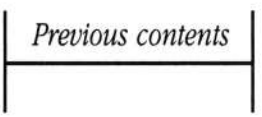
Stack before call



Word—Desired increment value

<—SP

Stack after call



<—SP

Errors None

C

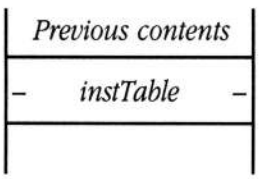
```
extern pascal SetIncr(increment);

Word      increment;
```

SetInstTable $121A

Sets the current instrument table to the one specified in *instTable.*

Parameters

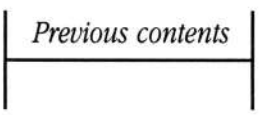
Stack before call



Long—Handle to instrument table

<—SP

Stack after call



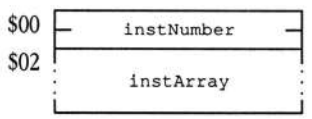
<—SP

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:



Word—Number of instruments in table

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. Before starting to play a sequence, an application should call `SetTrkInfo` for each track it uses.

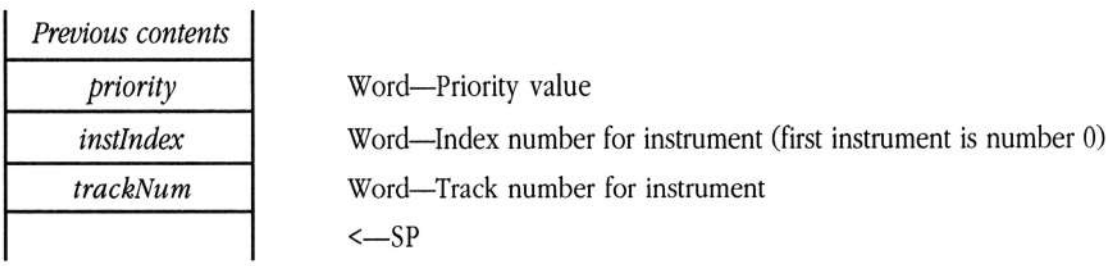
If MIDI was enabled when the Note Sequencer was started up (see "`SeqStartUp` $021A" earlier in this chapter), then `SetTrkInfo` can be used to enable MIDI output on particular tracks. If the most significant bit of the *trackNum* parameter is set to 1, then everything played on the specified track will 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 to 1.

You must make a `SetInstTable` call before issuing this call.

Parameters

Stack before call



Stack after call

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

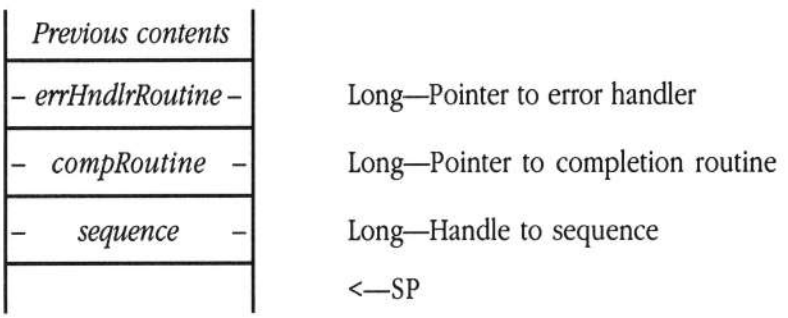
```
extern pascal StartInts();
```

StartSeq $0F1A

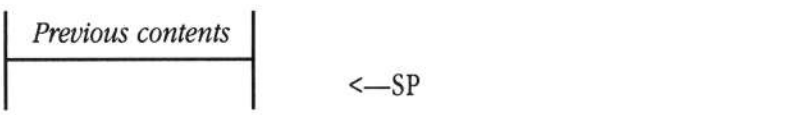
Starts interpretation of a series of seqItems stored at the address specified by the *sequence* parameter.

Parameters

Stack before call



Stack after call



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 was not started.
$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 *errHndlrRoutine* parameter is a pointer to an error-handling routine supplied by the application programmer. If *errHndlrRoutine* is set to NIL, then the Note Sequencer will not invoke a routine. For information about error-handling routines for the Note Sequencer, see "Error Handlers and Completion Routines" earlier 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 the Note Sequencer will not invoke a routine. For information about completion routines for the Note Sequencer, see "Error Handlers and Completion Routines" earlier 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 simply starts 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.

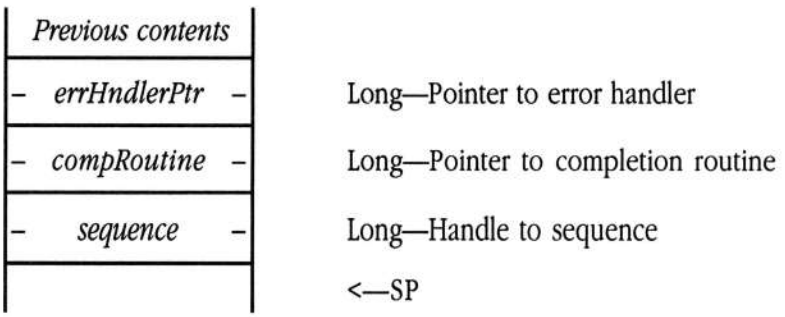
StartSeqRel $151A

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.

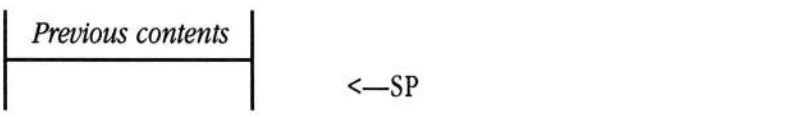
The Note Sequencer uses the dereferenced value of *sequence* as the base address for all phrases and patterns. It does not check for overflow and does not support negative offsets from the specified base address.

Parameters

Stack before call



Stack after call



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 was not started.
$2004	`miToolsErr`	Required tools not active or wrong version.
$2007	`miNoBufErr`	No MIDI output buffer is allocated.

C

```
extern pascal void StartSeqRel(errHndlrPtr,
          compRoutine, sequence);

Pointer   errHndlerPtr, compRoutine;
Handle    sequence;
```

errHndlrPtr The *errHndlrPtr* parameter is a pointer to an error-handling routine supplied by the application programmer. If *errHndlrPtr* is set to NIL, then the Note Sequencer will not invoke a routine. For information about error-handling routines for the Note Sequencer, see "Error Handlers and Completion Routines" earlier 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 the Note Sequencer will not invoke a routine. For information about completion routines for the Note Sequencer, see "Error Handlers and Completion Routines" earlier 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, a sequence presented in 65816 assembly language, shows 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+0'  ; 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 calls affect 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();
```

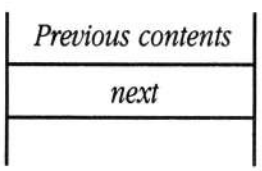
StopSeq $111A

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" earlier 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

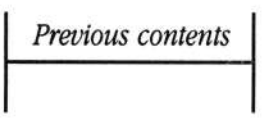
Stack before call



Word—Boolean; TRUE to process remaining phrases

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal StopSeq(next);

Boolean   next;
```

Note Sequencer error codes

Table 40-1 lists the error codes that may be returned by Note Sequencer calls.

■ **Table 40-1** Note Sequencer error codes

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.
$1A04	`noNoteErr`	Can't find the note for a `noteOff` command.
$1A05	`noStartErr`	The Note Sequencer was not started.
$1A06	`instBndsErr`	The specified instrument number is out of the bounds of the instrument table.
$1A07	`nsWrongVer`	The version of the Note Synthesizer 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 does, and it supports looping within a sound sequence and enveloping a sound.

- *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.

The following list summarizes the capabilities of the Note Synthesizer. The tool calls are grouped according to function. Later sections of this chapter discuss the tool set in greater detail and define the precise syntax of the Note Synthesizer tool calls.

Routine	**Description**
Housekeeping routines	
`NSBootInit`	Called only by the Tool Locator—must not be called by an application
`NSStartUp`	Initializes the Note Synthesizer for use by an application and establishes values for many important operational parameters
`NSShutDown`	Informs the Note Synthesizer that an application is finished using its tool calls
`NSVersion`	Returns the Note Synthesizer version number
`NSReset`	Called only when the system is reset—must not be called by an application
`NSStatus`	Returns the operational status of the Note Synthesizer

Note Synthesizer tool calls

`AllNotesOff`	Turns off all Note Synthesizer generators
`AllocGen`	Requests a sound generator
`DeallocGen`	Frees a sound generator
`NoteOff`	Lets a note die out
`NoteOn`	Starts a note
`NSSetUpdateRate`	Sets the update rate for the Note Synthesizer
`NSSetUserUpdateRtn`	Sets the user update routine

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 DOC generators for its use with `AllocGen`. It can play musical notes by making individual calls to `NoteOn` and `NoteOff` for each note that it plays. The `NoteOn` call starts a generator and a 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.

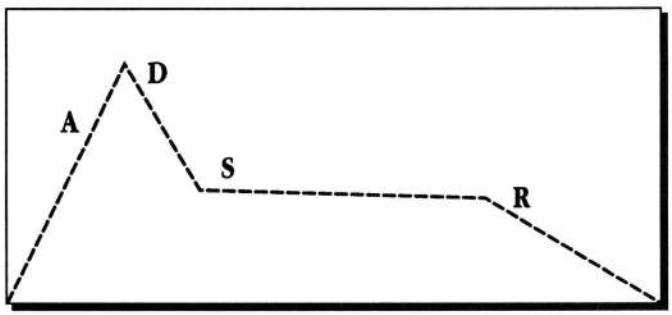
The Note Synthesizer requires that the Sound Tool Set be loaded and started up. One page of bank zero memory must be allocated to the Sound Tool Set for use as a direct page. The Note Synthesizer shares this direct-page space with the Sound Tool Set.

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 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 during which the sound increases 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 during which the sound falls off from its peak loudness to the level at which it stays, that 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, creates a blare like that of 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, by contrast, might have a very flat attack and decay, and a long, high sustain.

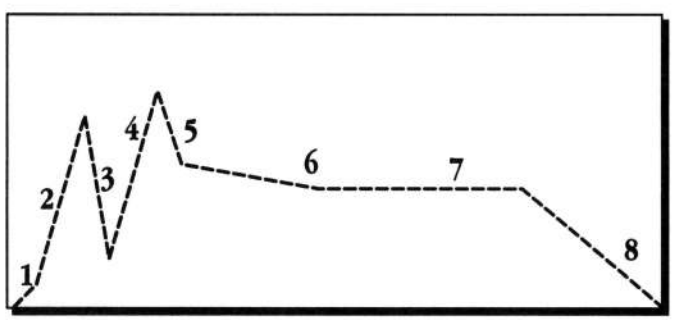
The release is the portion of a note as it dies away. A long release can produce a nice ringing quality but can also be a problem if the note is still sounding when another and dissonant note begins to sound.

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. The segments are defined as a series of breakpoints and increments. During each segment, the note's loudness slopes from its starting value toward its defined breakpoint value. 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. If the volume of any individual segment goes to 0 before the end of the segment, the Note Synthesizer considers the note done.

The breakpoint represents the loudness of the sound as 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 indicating the amount by which the current volume is to be adjusted at each update (the default rate is 100 updates per second; you can use the `NSStartUp` and `NSSetUpdateRate` tool calls to set other values). The low-order byte contains the numerator for a fractional increment. For example, an increment value of 1 translates to a fractional increment of 1/256. In this case, the volume is incremented once every 256 interrupts. The Note Synthesizer processes the segment until its volume reaches the specified breakpoint value. At that time, the Note Synthesizer moves to the next segment.

The length of time that an envelope segment lasts is given by the following formula:

$$T = \frac{|(L-N)| * 256}{(0.4)*(I*R)}$$

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

$$T = \frac{|(30-40)| * 256}{(0.4)*(25*100)} = \frac{2560}{(0.4)2500} = 2.56 \text{ seconds}$$

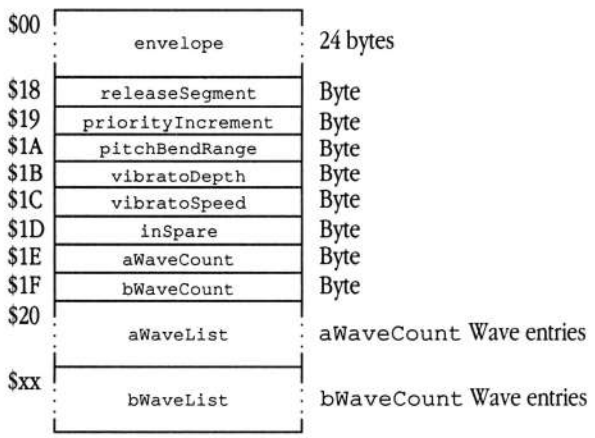
Thus, with the given parameters, the specified segment will last 2.56 seconds.

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 is sustained, then the note continues to play until no available generators are left, and the generator producing the note is reallocated to another note.

Instruments

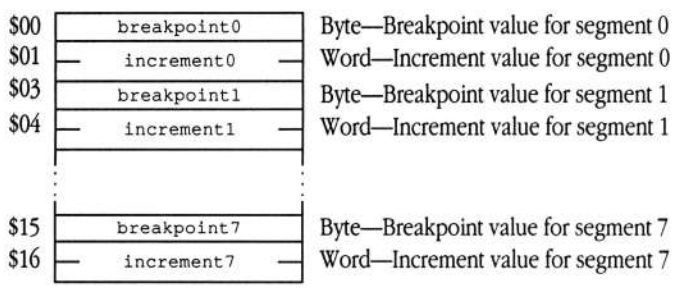
The Note Synthesizer's basic functional unit is an instrument. This is a data structure stored somewhere in the memory of the Apple IIGS that defines the sound characteristics of a played note. When a program makes the `NoteOn` call, it passes a pointer to an instrument, and that instrument is used while the sound is generated. Figure 41-3 shows the format of the instrument data structure.

■ **Figure 41-3** Instrument data structure



`envelope` Specifies the envelope for the sound as a series of eight segments, each a breakpoint and increment value pair (see "Note Synthesizer Envelopes" earlier in this chapter for detailed information on these concepts). Each breakpoint is a 1-byte value specifying a target volume level in the range from 0 through 127. Each increment is a 2-byte value that determines the amount of time the Note Synthesizer will spend reaching the breakpoint volume (and, therefore, the slope of the segment).

The `envelope` array has the following format:



`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 the 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 progresses through segments 5, 6, and 7 before ending the note.

`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" later in this chapter.

`pitchBendRange`

Specifies the maximum pitch bend that is possible for the note. The maximum possible value for a pitch bend is 127; `pitchBendRange` specifies how many semitones the pitch is raised when the pitch bend value is 127. The legal values are 1, 2, and 4 semitones. Note that the only way to change the pitch bend value of a note that is playing is to change the `pitchBendRange` field of the appropriate Generator Control Block (GCB) (see "Generators" later in this chapter for information on the format and content of the GCB).

The `pitchBendRange` field is used mainly 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.

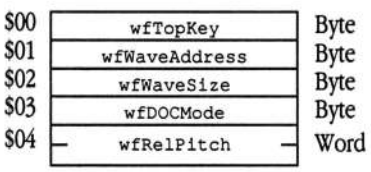
`vibratoSpeed` Controls the rate of vibrato. Higher values produce faster vibrato. The actual speed of vibrato effect depends on the update rate, which defaults to 100 updates per second. You can use the `NSStartUp` and `NSSetUpdateRate` tool calls to set other rates.

`inSpare` Must be set to zero.

`aWaveCount, bWaveCount`
Specify the number of wavelist entries (up to 255) that follow the wavecounts.

`aWaveList, bWaveList`
A wavelist is an array of variable length. The elements of the array are 6-byte structures. The corresponding wavecount field indicates the number of entries in each wavelist.

An entry in a wavelist data structure specifies wave data that is intelligible to the DOC. The Note Synthesizer places the data into the DOC registers.



`wfTopKey` When the Note Synthesizer plays a note, it examines the `wfTopKey` 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 `wfTopKey` value is the one it plays. For this reason, waveforms should be stored in increasing order of `wfTopKey` value. The last waveform in a wavelist should have a value of 127, the maximum valid pitch value.

`wfWaveAddress` The high byte of the waveform's address in sound RAM. Its value is copied into the Address Pointer register of the DOC.

`wfWaveSize` Sets the size of the DOC's wave table and the frequency resolution of the DOC. This data is copied directly to the DOC's Bank-Select/TableSize/Resolution register. The resolution and table size should normally be equal.

`wfDocMode` Sets the mode of the DOC. This field 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.

`wfRelPitch` A word value used to tune the waveform. The high-byte value is the semitone, and the low-byte value 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 `wfRelPitch` field. Such a wavelist specifies 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," and the *Apple IIGS Hardware Reference.*

DOC memory

An application that uses the Note Synthesizer must use the Sound Tool Set call `WriteRAMBlock` to load into DOC memory any waveforms that it can use. You must not place a 0 in the first 256 bytes of DOC memory because doing so halts the timer oscillator and causes a system failure. If the application uses the clock function of the MIDI Tool Set, 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.

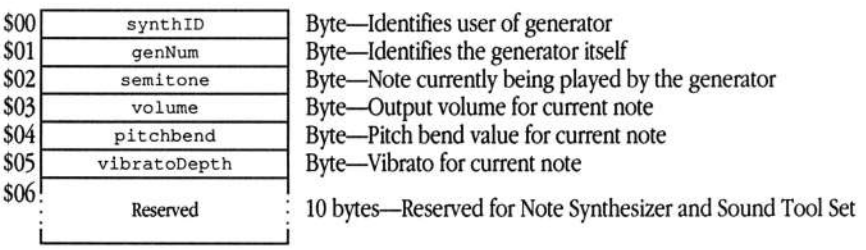
The Note Synthesizer allocates generators to all the different sound tools that may need them. It therefore requires a priority scheme for allocating generators in the event that a generator is requested when all generators are in use. When a generator is allocated, it receives a priority. A generator's priority may range from 0 through 128. A priority of 0 means the generator is not being used and will be allocated to any sound tool that requests it. A priority of 128 indicates that the generator is locked and cannot be reallocated. The Note Synthesizer uses remaining values in a generator's range 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. Your application specifies a priority when requesting a generator. The Note Synthesizer then allocates a generator to your application if it finds one with a lower priority value (see the description of the `AllocGen` tool call later in this chapter for more information).

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 "dials" 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 layout (`GCBRecord`)



`synthID` Identifies who is currently using the generator. Valid values are

0	Not used
1	Sound Tool Set free-form 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 specifies the maximum allowable pitch bend in semitones (see "Instruments" earlier in this chapter).

`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();
```

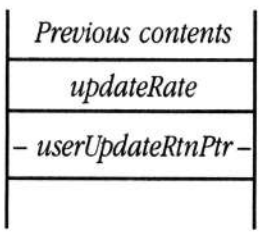
NSStartUp $0219

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 *userUpdateRtnPtr* 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

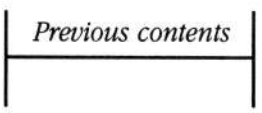


Word—Rate of envelope generation

Long—Pointer to custom interrupt routine

<—SP

Stack after call



<—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,
          userUpdateRtnPtr);

Word      updateRate;
Pointer   userUpdateRtnPtr;
```

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

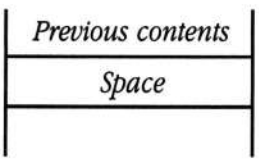
```
extern pascal void NSShutDown();
```

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

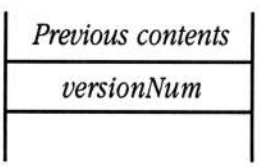
Stack before call



Word—Space for result

<—SP

Stack after call



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();
```

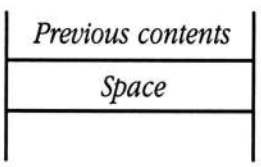
NSStatus $0619

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

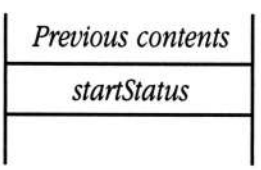
Stack before call



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 Sound Tool Set free-form synthesizer.

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

Errors None

C

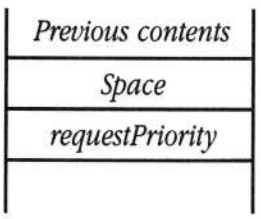
```
extern pascal void AllNotesOff();
```

AllocGen $0919

Requests a sound generator. Returns a generator number from 0 to 13. The call reallocates a generator if all generators are allocated and the specified *requestPriority* exceeds that of one of the previously allocated generators.

Parameters

Stack before call

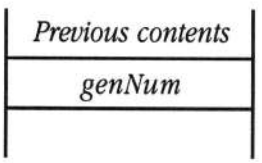


Word—Space for result

Word—Desired generator priority

<—SP

Stack after call



Word—Number of allocated generator

<—SP

Errors	$1921	`nsNotAvail`	No generators available to allocate.
	$1923	`nsNotInit`	Note Synthesizer not started up.

C

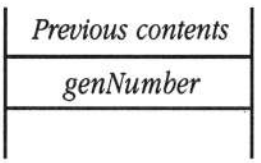
```
extern pascal Word AllocGen(requestPriority);

Word      requestPriority;
```

DeallocGen $0A19

Sets the named generator's allocation priority to 0 and halts its oscillators. Any subsequent allocation request with a valid *requestPriority* will then succeed.

Parameters

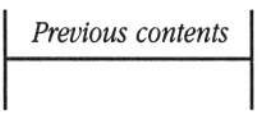
Stack before call



Word—Number of generator to deallocate

<—SP

Stack after call



<—SP

Errors

$1922	`nsBadGenNum`	Invalid generator number.

C

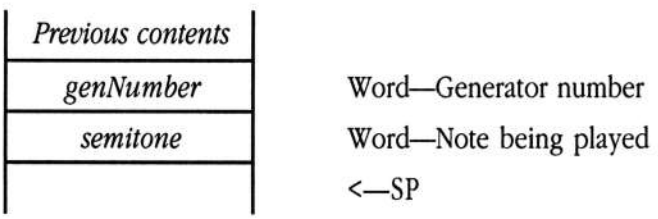
```
extern pascal void DeallocGen(genNumber);

Word       genNumber;
```

NoteOff $0C19

Switches the specified generator to release mode, causing 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 *genNumber* and *semitone* parameters should be set to the same values specified in the corresponding `NoteOn` call.

Parameters

Stack before call



Stack after call

Errors None

C

```
extern pascal void NoteOff(genNumber, semitone);

Word      genNumber, semitone;
```

NoteOn $0B19

Initiates the generation of a note on a specified generator. The *genNumber* 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 is inaudible even if everything else is working correctly. The dynamic range of the DOC is 48 decibels.

Parameters

Stack before call

Previous contents	
genNumber	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(genNumber, semitone,
          volume, instrumentPtr);

Word      genNumber, 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.

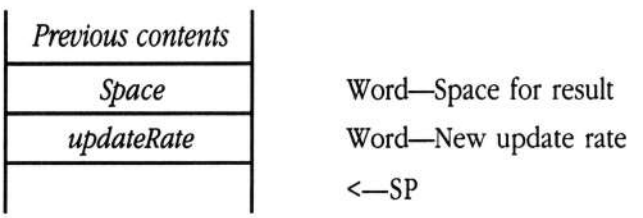
```
            pushword #0             ;space for GenNum
            pushword #64            ;priority of this note
            _AllocGen               ;retrieve an allocated generator
            pla                     ;get the generator number
            sta GenNum              ;store it

            pushword GenNum         ;push parameters:generator
            pushword Semitone       ;note
            pushword #127           ;maximum volume
            pushlong #Instrument    ;LONG pointer to instrument
definition
            _NoteOn
                                    •
                                    •
                                    •
            pushword GenNum         ;push parameters: generator
            pushword Semitone       ;note
            _NoteOff                ;turn off the note
```

NSSetUpdateRate $0E19

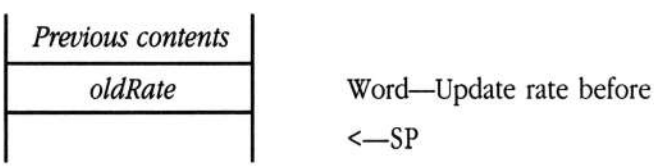
Sets the Note Synthesizer's *updateRate* parameter, as described under `NSStartUp` in "Note Synthesizer Housekeeping Calls" earlier in this chapter. The specified *updateRate* value becomes the new *updateRate*, and the old value is returned.

Parameters

Stack before call



Stack after call



Errors

$1923	`nsNotInit`	Note Synthesizer not started up.

C

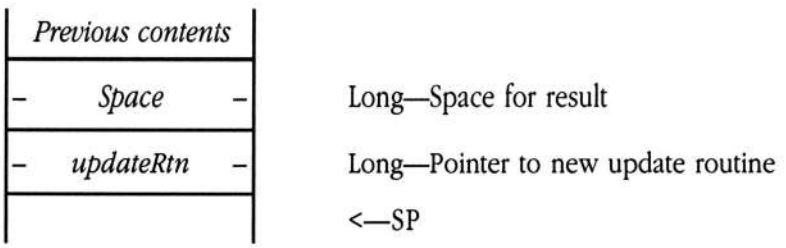
```
extern pascal Word NSSetUpdateRate(updateRate);

Word      updateRate;
```

NSSetUserUpdateRtn $0F19

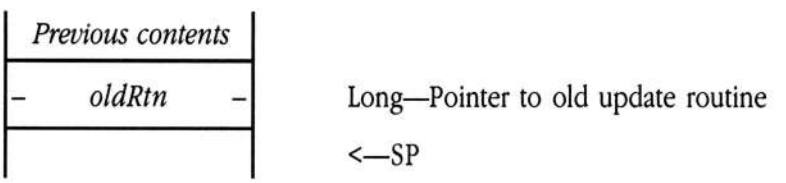
Sets the user update routine described under `NSStartUp` in "Note Synthesizer Housekeeping Calls" earlier in this chapter. The update routine pointer is set to the value passed in the *updateRtn* parameter, and the address of the old update routine is returned. If there is no user update routine when this call is made, it returns a NIL pointer. A NIL *updateRtn* value disables the current update routine.

Parameters

Stack before call



Stack after call



Errors

$1923	`nsNotInit`	Note Synthesizer not started up.

C

```
extern pascal VoidProcPtr
            NSSetUserUpdateRtn(updateRtn);

Pointer     updateRtn;
```

Note Synthesizer error codes

Table 41-1 lists the error codes that may be returned by Note Synthesizer calls.

■ **Table 41-1** Note Synthesizer error codes

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 Volume 1 of the *Toolbox Reference.*

- The diagram for the job subrecord, Figure 15-10 on page 15-14 of Volume 1 of the *Toolbox Reference,* shows that the `fFromUsr` field is a word. This is incorrect. The `fFromUsr` field is actually a byte. Note that as a result the offsets for all fields following this one are incorrect. This error is also reflected in the tool set summary at the end of the chapter.
- The description of the `PrJobDialog` tool call includes this incorrect statement: "The initial settings displayed in the dialog box are taken from the printer driver." The sentence should begin "The initial settings displayed in the dialog box are taken from the print record."

Clarifications

The following items provide additional information about features previously described in Volume 1 of the *Toolbox Reference.*

- 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 Volume 2 of the *Toolbox Reference* for the layout of the `locInfo` record).
- The port driver auxiliary file type of an AppleTalk driver is $0003. Its file type remains $BB.

New features of the Print Manager

The following functions have been added to the Print Manager:

- 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 deletes such files and creates new ones 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 passes control to a routine that (1) determines what call was being made to the driver, (2) pops the parameters off the stack, and (3) returns a `missingDriver` error ($1301). The Print Manager also displays an alert asking the user to make sure a printer and port driver are selected, if your application calls `PrJobDialog` and `PrStlDialog`.
- 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 displays an alert directing the user to use the Control Panel. New applications should never issue this call and should not include the Choose Printer command in the file menu. Note that `PMStartup` still loads the List Manager if it has not already been loaded.
- The Print Manager now allows you to assign a name to a document. This feature is primarily applicable to documents destined for AppleTalk printers and is used by AppleShare® print servers for the print log.
- 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 (`bjDocLoop`) and copy count field (`iCopies`) 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 (such as a Macintosh computer).

New Print Manager calls

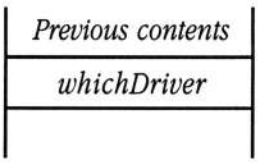
The following sections discuss new Print Manager tool calls.

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

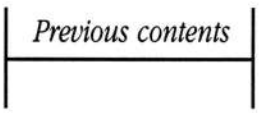
Stack before call



Word—Printer driver to load

<—SP

Stack after call



<—SP

Errors	$1309	`badLoadParam`	The specified parameter is invalid.
	Loader errors		Returned unchanged

C

```
extern pascal void PMLoadDriver(whichDriver);

Word       whichDriver;
```

whichDriver Specifies which printer driver to load. Legal values for the driver parameter include

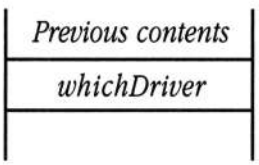
0 Load both drivers.
1 Load printer driver.
2 Load port driver.

PMUnloadDriver $3413

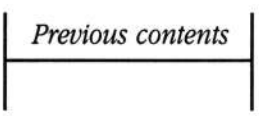
Unloads the current port driver, printer driver, or both, depending on the input parameter.

Parameters

Stack before call



Stack after call



Errors	$1309	`badLoadParam`	The specified parameter is invalid.
	Loader errors		Returned unchanged

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.

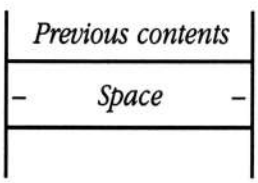
PrGetDocName $3613

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

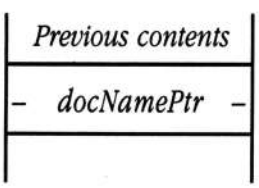
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to document name string (Pascal string)

<—SP

Errors None

C

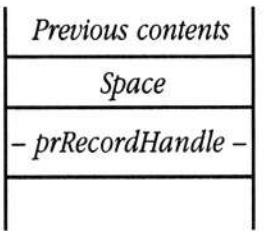
```
extern pascal Pointer PrGetDocName();
```

PrGetPgOrientation $3813

Returns a value indicating the current page orientation for the specified document.

Parameters

Stack before call

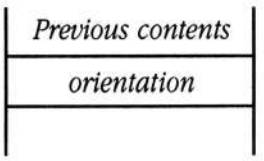


Word—Space for result

Long—Handle to print record for document

<—SP

Stack after call



Word—Page orientation: 0 = portrait, 1 = landscape

<—SP

Errors None

C

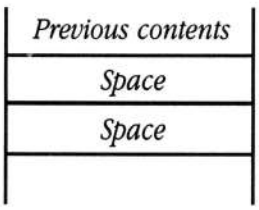
```
extern pascal Word
          PrGetPgOrientation(prRecordHandle);

Handle    prRecordHandle;
```

PrGetPrinterSpecs $1813

Returns information about the currently selected printer.

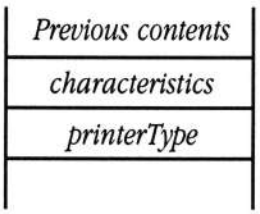
Parameters

Stack before call



Word—Space for result
Word—Space for result
<—SP

Stack after call



Word—Word defining printer characteristics
Word—Word indicating the type of printer connected
<—SP

Errors None

C

```
extern pascal PrinterSpecs PrGetPrinterSpecs();
```

characteristics Defines the features of the particular printer.

Reserved bits 15–2 Must be set to 0.

`color` bits 1–0 Indicates color capability.
00 = Can't determine
01 = Black and white only
10 = Color capable
11 = Reserved

printerType Indicates the type of printer selected.

0 Undefined
1 ImageWriter I or II
2 ImageWriter LQ
3 LaserWriter family printer that supports PostScript (LaserWriter, LaserWriter Plus, and LaserWriter IINT and IINTX)
4 Epson

PrSetDocName $3713

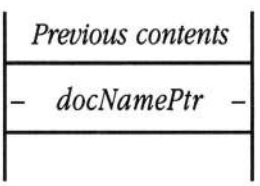
Sets the document name for use with AppleTalk printers. The Print Manager passes this name when connecting to printers and spoolers, allowing the destination printer to report the name properly.

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

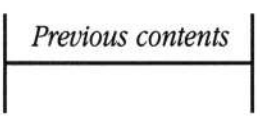
Stack before call



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

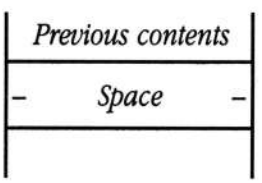
The following calls, not previously documented, may be useful to application programmers.

`PrGetNetworkName` $2B13

Returns the AppleTalk network name for the currently selected printer. If the user has selected a nonnetworked printer, the call returns a NIL pointer.

Parameters

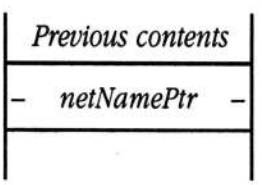
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to printer network name string (Pascal string)

<—SP

Errors None

C

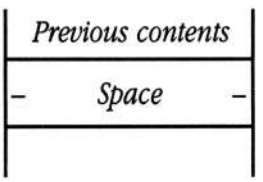
```
extern pascal Pointer PrGetNetworkName();
```

PrGetPortDvrName $2913

Returns the name string for the currently selected port driver.

Parameters

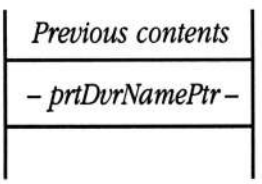
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to port driver name string (Pascal string)

<—SP

Errors None

C

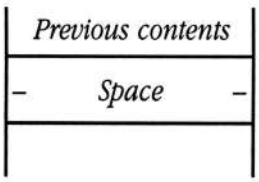
```
extern pascal Pointer PrGetPortDvrName();
```

PrGetPrinterDvrName $2813

Returns the name string for the currently selected printer driver.

Parameters

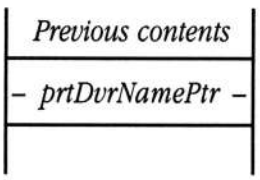
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to printer driver name string (Pascal string)

<—SP

Errors None

C

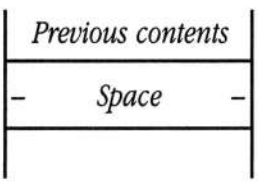
```
extern pascal Pointer PrGetPrinterDvrName();
```

PrGetUserName $2A13

Returns the user name as entered in the Control Panel.

Parameters

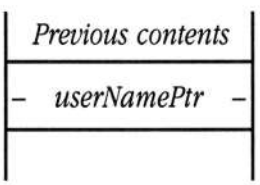
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to user name string (Pascal string)

<—SP

Errors None

C

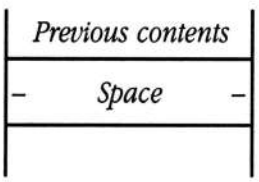
```
extern pascal Pointer PrGetUserName();
```

PrGetZoneName $2513

Returns the name string for the currently selected AppleTalk print zone. If the user has selected a nonnetworked printer, the call returns a NIL pointer.

Parameters

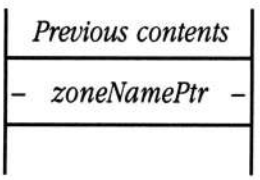
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to zone name string (Pascal string)

<—SP

Errors None

C

```
extern pascal Pointer PrGetZoneName();
```

Print Manager error codes

Table 42-1 lists all valid Print Manager error codes.

■ **Table 42-1** 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 with the LaserWriter cannot be established.
$1307	`papReadWriteErr`	Read-write error on the LaserWriter.
$1308	`ptrConnFailed`	Connection with the ImageWriter cannot be established.
$1309	`badLoadParam`	The specified parameter is invalid.
$130A	`callNotSupported`	Tool call is not supported by current version of the driver.
$1321	`startUpAlreadyMade`	`LLDStartUp` call already made.

Chapter 43 **QuickDraw II Update**

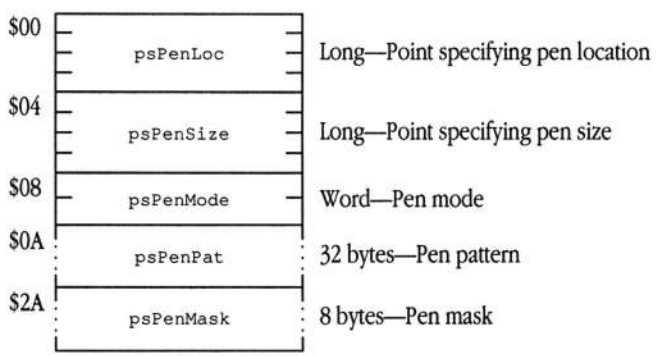
This chapter documents new features of QuickDraw II. The complete reference to QuickDraw II is in Volume 2, Chapter 16 of the *Apple IIGS Toolbox Reference.*

Error corrections

The following items provide corrections to the documentation for QuickDraw II in Volume 2 of the *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.
- Chapter 16 states that the coordinates passed to the `LineTo` and `MoveTo` calls should be expressed as global coordinates. In fact, the coordinates must be local and must refer to the GrafPort in which the drawing or moving takes place.
- The pen state record shown in Figure 16-38 on page 16-238 of Volume 2 of the *Toolbox Reference* is incorrect. The correct record layout is shown in Figure 43-1.

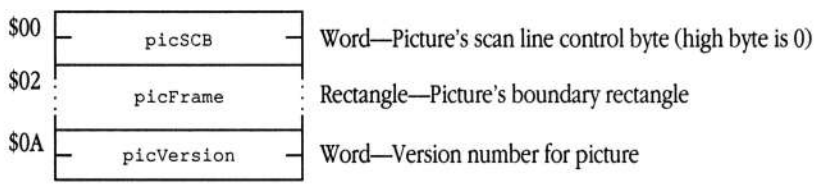
- **Figure 43-1** Pen state record



Clarification

QuickDraw pictures are described by a series of QuickDraw operation codes that record the commands by which the picture was created. When these pictures are stored as data structures, the actual picture data (the operation codes) is preceded by control information, some of which may be of interest to Apple IIGS developers. Figure 43-2 shows some of this control information. Note that the layout of this control information is subject to change.

■ **Figure 43-2** QuickDraw picture header



New features of QuickDraw II

The following information describes new features in this version of QuickDraw II.

- QuickDraw II now supports 16-by-8 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. Because the `OpenPort` and `InitPort` tool calls clear this bit, existing applications will work fine.
- The `PointInRect` call 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. Although 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. This ensures that FASTFONT is properly installed.

QuickDraw II speed enhancement

In addition to FASTFONT, several other changes that improve drawing performance have been made to QuickDraw II. First, pattern filling in `modeCopy` and `modeXOR` now operates between two and four 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 back the *masterSCB* parameter and check shadowing status.

In addition, your application can further improve QuickDraw II performance by following some simple rules. First, your application must change GrafPort fields only 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 erasing at the same time, then all the redrawing. In this manner, QuickDraw II has to change its drawing pattern only 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`.

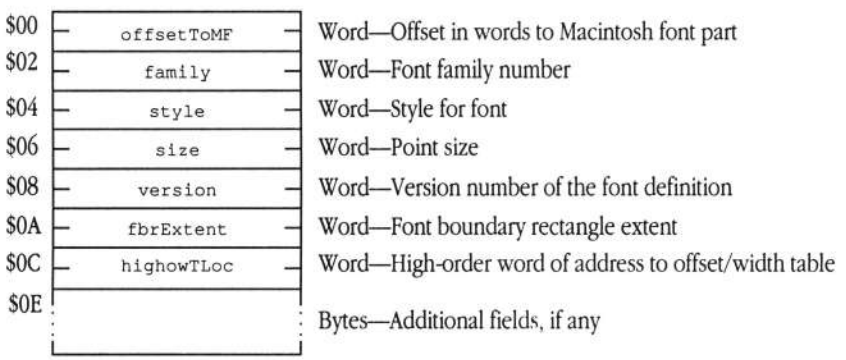
The *masterSCB* now has the following format:

`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 13–8	Must be set to 0.
SCB	bits 7–0	Use standard SCB values.

New font header layout

The font header has been expanded to include a new field containing additional addressing information. Figure 43-3 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 43-3** New font header layout



`highowTLoc` Defines the high-order word of the address of the offset/width table for the font. The `owTLoc` field defined in the old font header 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 feature of QuickDraw II Auxiliary

QuickDraw II now supports text justification within pictures. Note that QuickDraw II justifies the text only in the drawn 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 15–4	Must be set to 0.
`fTextJust`	bit 3	Controls text justification in pictures. 0 = Don't justify text 1 = Justify text
	bits 2–0	Use standard `fontFlags` values (see page 16-56 in Volume 2 of the *Toolbox Reference* for a description of these bits).

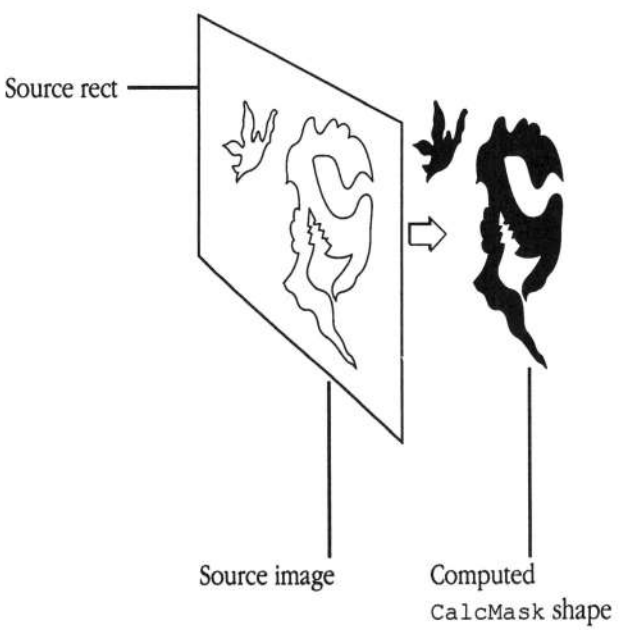
New QuickDraw II Auxiliary calls

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 boundary 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). Figure 44-1 shows an example of mask generation.

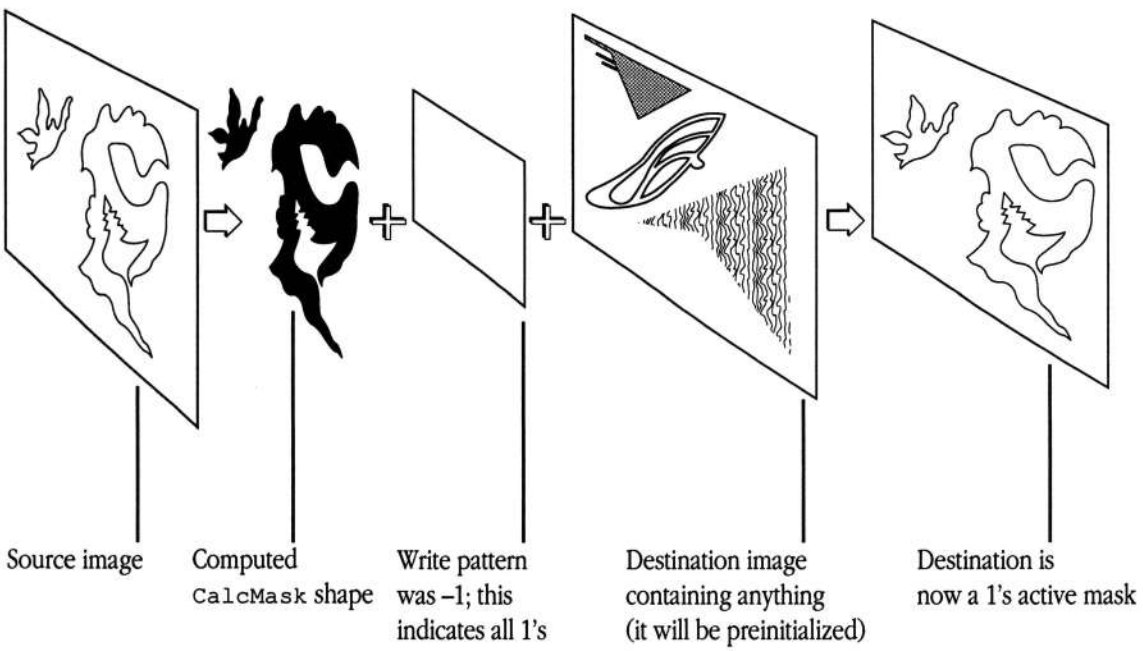
■ **Figure 44-1** Mask generation with `CalcMask`



This call differs from `SeedFill` only in that it works from the "outside in"; `SeedFill` goes "inside out," filling all enclosed areas starting from a specified interior point (see the description of the `SeedFill` tool call later in this chapter for details).

`CalcMask` is most commonly used to implement a lasso tool. `CalcMask` determines the selected shape by filling inward from the lasso rectangle. Figure 44-2 shows an example.

■ **Figure 44-2** Implementing a lasso tool with `CalcMask`



For this use, set the call parameters as follows:

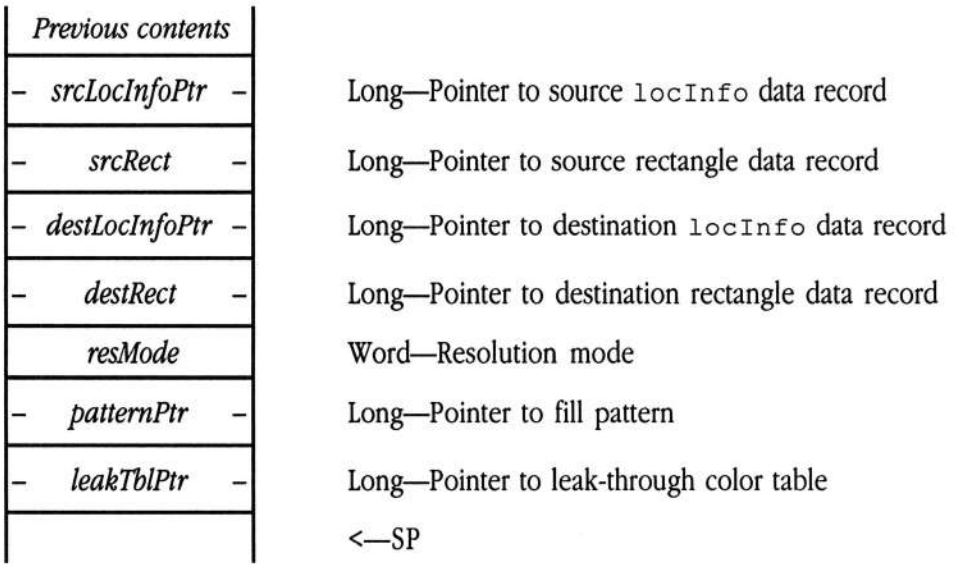
`destMode` portion of *resMode*	%0010 (clear destination to 0's before drawing)
patternPtr	$FFFFFFFF (use all 1's pattern when drawing to destination)

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 to ensure an accurate fill. △

Parameters

Stack before call



Stack after call

Errors

$0201	`memErr`	`NewHandle` error occurred.
$1211	`badRectSize`	Height or width is negative, *destRect* is not the same size as *srcRect*, or the source or destination rectangle is not within its boundary rectangle.
$1212	`destModeError`	`destMode` portion of *resMode* is 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 of 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 15–12 Indicates initialization and drawing options.
0000 = Copy source to destination (obliterating destination)
0001 = Leave destination alone (overlay source onto destination)
0010 = Initialize destination to 0's before drawing
0011 = Initialize destination to 1's before drawing
Other values are invalid.

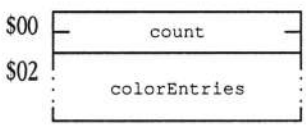
Reserved bits 11–2 Must be set to 0.

`res` bits 1–0 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 0's pattern when writing to destination
$FFFFFFFF	Use an all 1's 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. Valid values in 640 pure mode range from 0 through 3, inclusive; for 320 mode and 640 dithered mode valid values range from 0 through 15, inclusive.



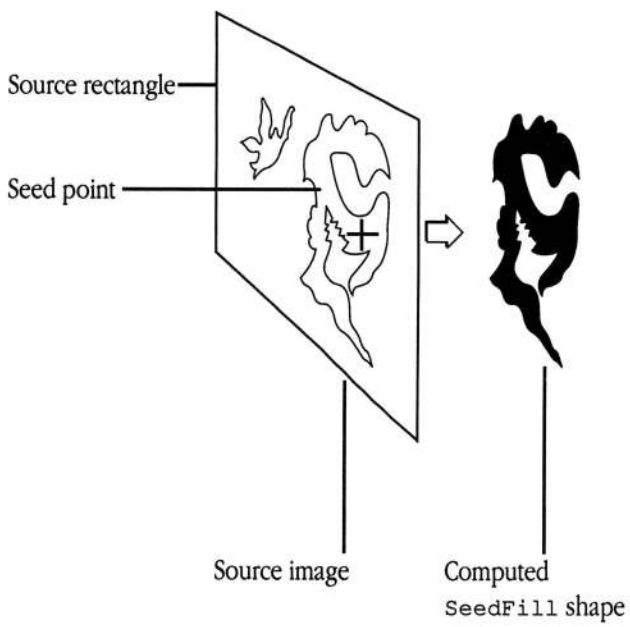
Word—Count of color entries to follow

`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. Figure 44-3 shows an example.

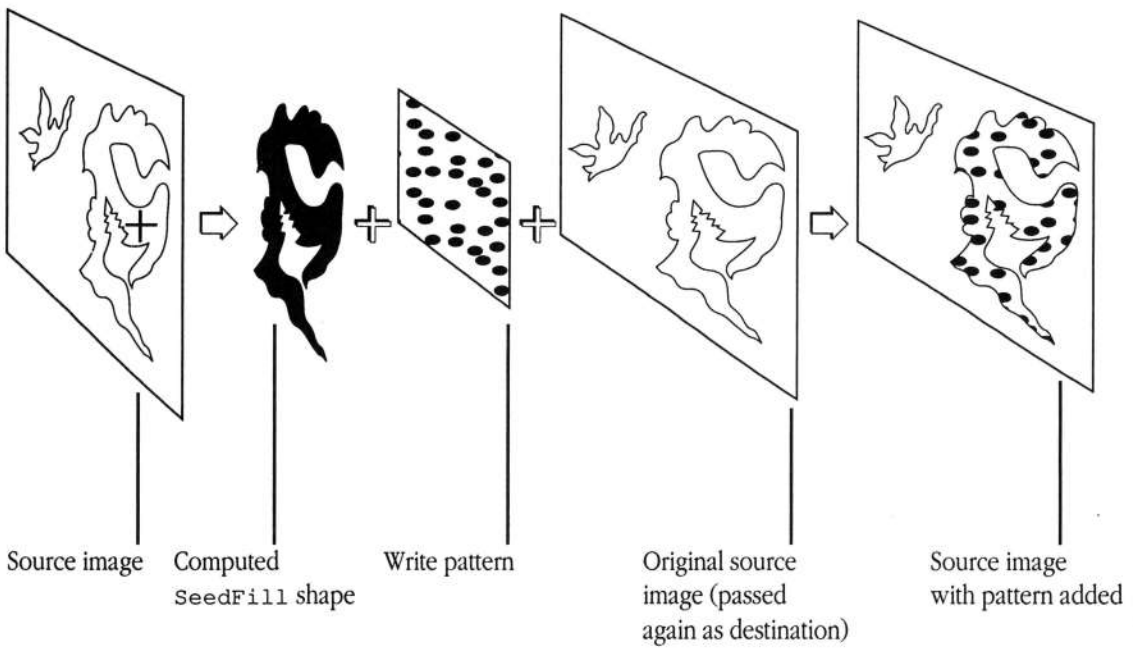
■ **Figure 44-3** Mask generation with `SeedFill`



This call differs from `CalcMask` only in that it works from the "inside out"; `CalcMask` goes "outside in" (see the description of the `CalcMask` tool call earlier in this chapter for details).

`SeedFill` is a versatile tool. Most simply, you can use it to implement a paint bucket tool, as in Figure 44-4.

■ **Figure 44-4** Implementing a paint bucket tool with `SeedFill`

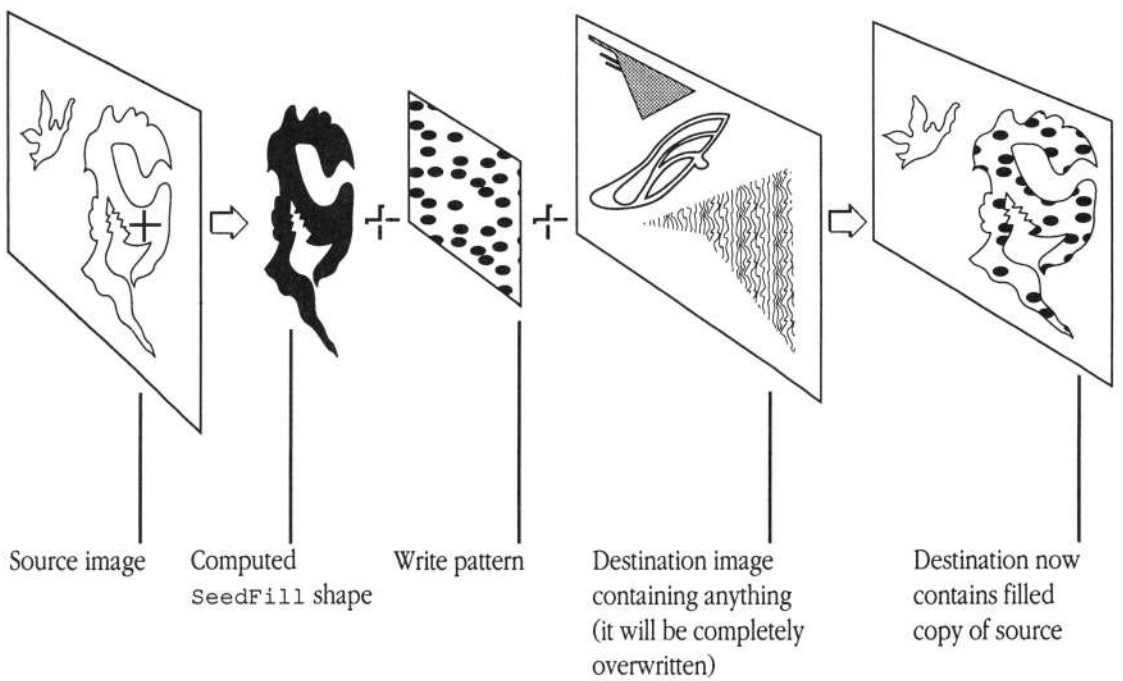


For this operation, use the following call parameter values:

`destMode` portion of *resMode*	%0001 (do not change destination image before drawing)
patternPtr	Pointer to fill color or pattern

To add an undo capability to the paint bucket, specify a different destination, as in Figure 44-5.

■ **Figure 44-5** Paint bucket tool with undo

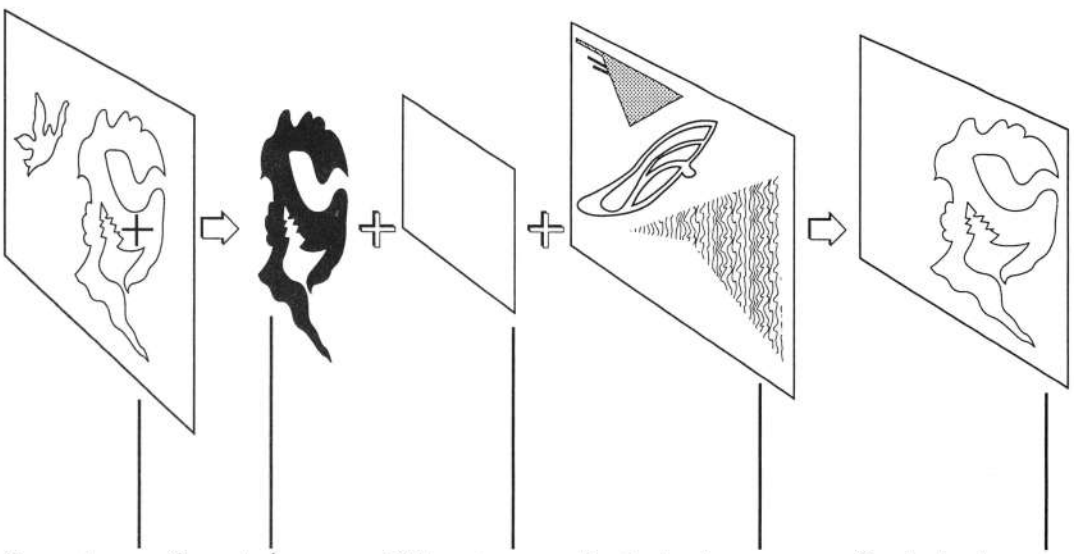


For this operation, use the following call parameter values:

`destMode` portion of *resMode*	%0000 (copy source to destination)
patternPtr	Pointer to fill color or pattern

Figure 44-6 shows a more complex example, illustrating the "from-the-inside" lasso tool.

■ **Figure 44-6** Implementing a "from-the-inside" lasso tool with `SeedFill`



For this operation, use the following call parameter values:

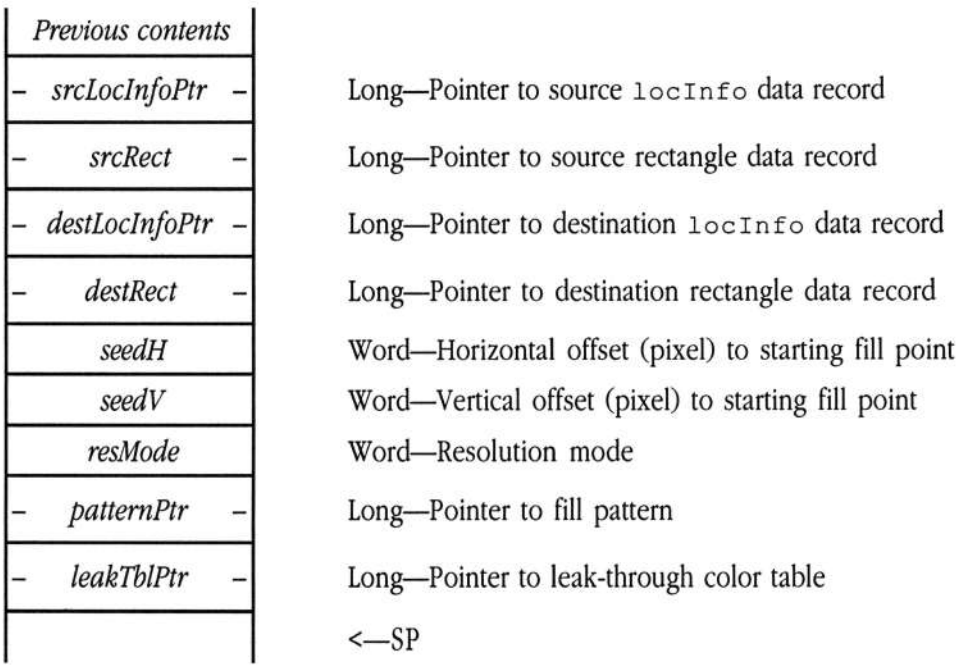
`destMode` portion of *resMode*	%0010 (clear destination to 0's before drawing)
patternPtr	$FFFFFFFF (use all 1's pattern when drawing to destination)

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 to ensure an accurate fill. △

Parameters

Stack before call



Stack after call

Errors

$0201	`memErr`	`NewHandle` error occurred.
$1211	`badRectSize`	Height or width is negative, *destRect* is not the same size as *srcRect*, or the source or destination rectangle is not within its boundary rectangle.
$1212	`destModeError`	`destMode` portion of *resMode* is 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 of 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 15–12 Indicates initialization and drawing options.
0000 = Copy source to destination (obliterating destination)
0001 = Leave destination alone (overlay source onto destination)
0010 = Initialize destination to 0's before drawing
0011 = Initialize destination to 1's before drawing
Other values are invalid.

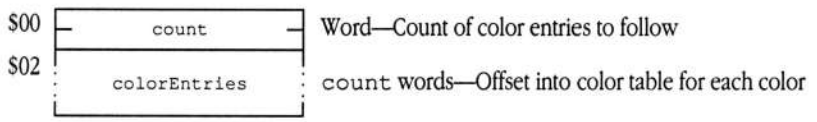
Reserved bits 11–2 Must be set to 0.

`res` bits 1–0 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 0's pattern when writing to destination
$FFFFFFFF	Use an all 1's 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.



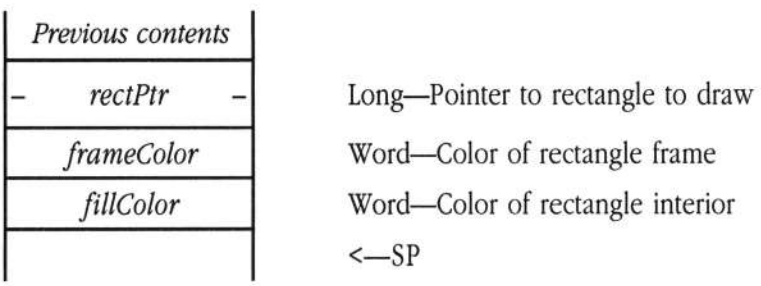
SpecialRect $0C12

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



Stack after call

Errors None

C

```
extern pascal void SpecialRect(rectPtr, frameColor,
                fillColor);

Pointer   rectPtr;
Word      frameColor, fillColor;
```

frameColor, fillColor

The low-order 4 bits of each of these parameters specify the color.

Chapter 45 **Resource Manager**

This chapter documents the features of the 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 can define 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, that 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 list summarizes the capabilities of the Resource Manager. The tool calls are grouped according to function. Later sections of this chapter discuss resources in greater detail and define the precise syntax of 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
`ResourceVersion`	Returns the Resource Manager version number
`ResourceReset`	Called only when the system is reset—must not be called by an application
`ResourceStatus`	Returns the operational status of the Resource Manager

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
`GetIndType`	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 in the case of 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 of 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. Because 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 list 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 meaningful only in the context of a given resource type, the system does place restrictions on the ID values you can assign. The following list 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 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 list summarizes the allowable ranges:

ID range	Lowest possible ID returned	Highest possible ID returned
$0000	$00000001 (zero 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 to identify the resource uniquely. In some cases, this may be more convenient 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, be careful to use the standard data structures for defining those names.

Using resources

In most cases, applications use the Resource Manager only indirectly, that is, 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 make far greater use of the Resource Manager. The next several paragraphs describe the steps your program must follow to use its predefined resources.

1. Unlike most other tool sets, the Resource Manager need not be started up by your program. 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, your 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, however, 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 automatically closes any open resource files. Once your program issues a `ResourceShutDown` call, it should not make any other Resource Manager calls, except for `ResourceStartUp`.

Resource attributes

Every resource is associated with a set of attributes that define the current state of the resource and place 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 attributes 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, 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 call description in Chapter 12, "Memory Manager," in Volume 1 of the *Toolbox Reference* for the format and content of the memory attributes word.

Here are the contents of the attributes word for a resource:

`attrLocked`	bit 15	Passed to Memory Manager `NewHandle` tool call when memory is allocated for the resource. 0 = Memory for resource not locked 1 = Memory for resource locked; cannot be moved or purged
`attrFixed`	bit 14	Passed to Memory Manager `NewHandle` tool call when memory is allocated for the resource. 0 = Memory for resource need not be fixed 1 = Memory for resource is fixed and cannot be moved
Reserved	bits 13–12	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). 0 = Resource does not require a converter routine 1 = Resource requires 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. 0 = Resource need not be loaded at a specific location 1 = Resource to be loaded at specific location
`attrPurge`	bits 9–8	Passed to Memory Manager `NewHandle` tool call when memory is allocated for the resource. 00 = Purge level 0 01 = Purge level 1 10 = Purge level 2 11 = Purge level 3
`resProtected`	bit 7	Indicates whether the resource is write-protected. If this bit is set to 1, then applications may not update the resource on disk. 0 = Resource is not write-protected 1 = Resource is write-protected
`resPreLoad`	bit 6	Specifies whether the Resource Manager should load the resource into memory at `OpenResourceFile` time. If this bit is set to 1, then this resource is loaded into memory when the resource file is opened, rather than when the resource itself is accessed. 0 = Do not preload the resource 1 = Preload the resource
`resChanged`	bit 5	Indicates whether the resource has been changed. If this bit is set to 1 for a non–write-protected resource, the Resource Manager updates the resource on disk at `CloseResourceFile` time. 0 = Resource has not been changed in memory 1 = Resource has been changed in memory and therefore differs from the version stored on disk
`attrNoCross`	bit 4	Passed to Memory Manager `NewHandle` tool call when memory is allocated for the resource. 0 = Memory may cross bank boundary 1 = Memory may not cross bank boundary
`attrNoSpec`	bit 3	Passed to Memory Manager `NewHandle` tool call when memory is allocated for the resource. 0 = May use special memory 1 = May not use special memory

`attrPage`	bit 2	Passed to Memory Manager `NewHandle` tool call when memory is allocated for the resource. 0 = Memory need not be page-aligned 1 = Memory must be page-aligned
Reserved	bits 1–0	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 constitute the resource data. The resource map is a directory to those resources, containing information on both location and size. 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

When 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 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 of a resource file.

Resource file search sequence

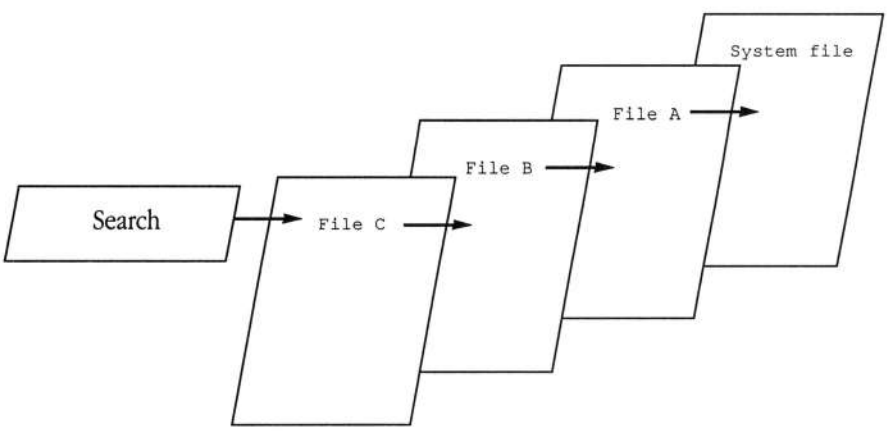
As your program opens resource files, the Resource Manager adds those files to the head of the resource file search chain for your application. The Resource Manager uses this search chain for many of its operations, such as locating a resource. The system resource file is always the last file in the 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 tool calls

```
OpenResourceFile        File A
OpenResourceFile        File B
OpenResourceFile        File C
```

the Resource Manager builds the search chain shown in Figure 45-1 for your application.

■ **Figure 45-1** A resource file search chain



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), because it is the 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.

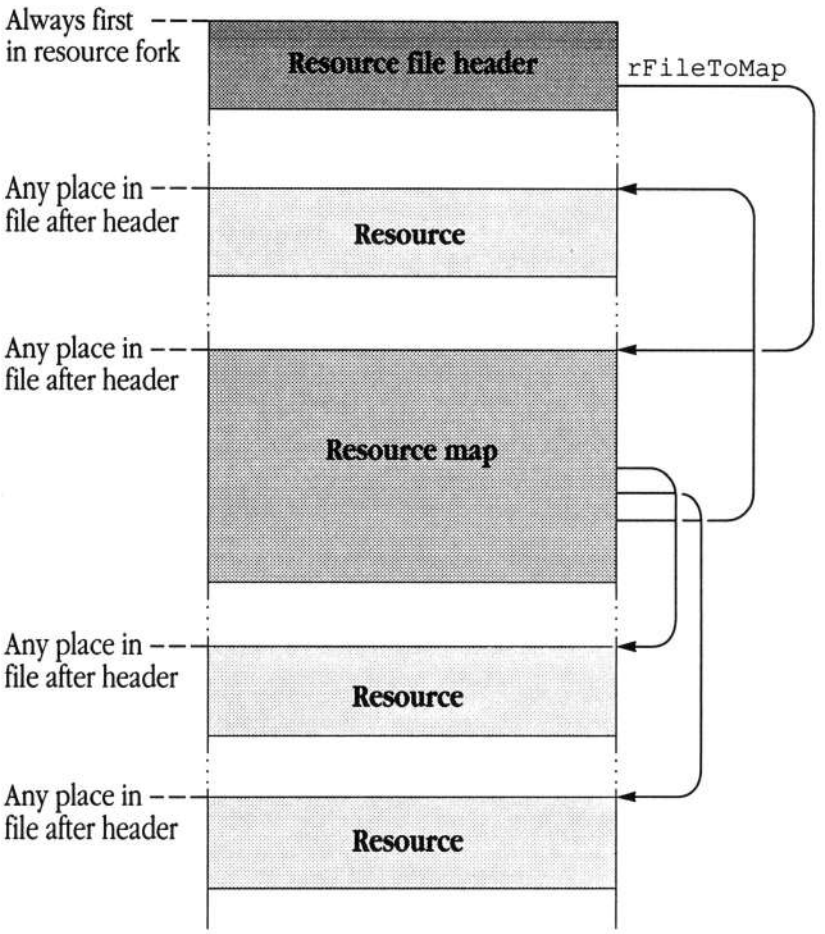
The Resource Manager provides tool calls that allow your program to control the search sequence for the resource file chain. The `SetCurResourceFile` tool call changes the current resource file, so that any resource file, including the System file, can be the first file searched, though the search still terminates when the Resource Manager either finds the desired resource or encounters the end of the file chain. The `SetResourceFileDepth` tool call controls the number of files the Resource Manager searches before giving up. By using these calls, your program can fine-tune resource searches for performance or can inhibit access to some resource files during 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 45-2 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 45-2** Resource file internal layout



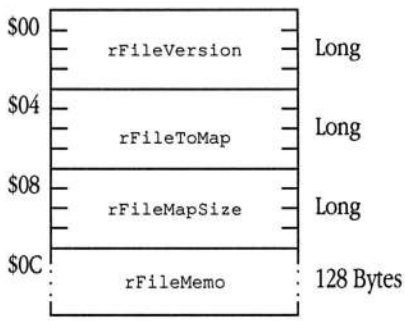
The Resource Manager controls the relative positions of all elements of the resource fork. It moves or resizes the map or resources as required. Therefore, your program should never rely on the location of any element in the fork, except for the resource file header.

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, shown in Figure 45-3, is the first data block in every resource fork.

■ **Figure 45-3** Resource file header (`ResHeaderRec`)



`rFileVersion` Version number defining layout of resource file. Currently, only version 0 is supported. This field allows Apple IIGS resource files to be distinguished from Macintosh resource files; the first long in Macintosh resource files must have a value greater than 127.

`rFileToMap` Offset, in bytes, to beginning of the resource map. This offset starts from the beginning of the resource file.

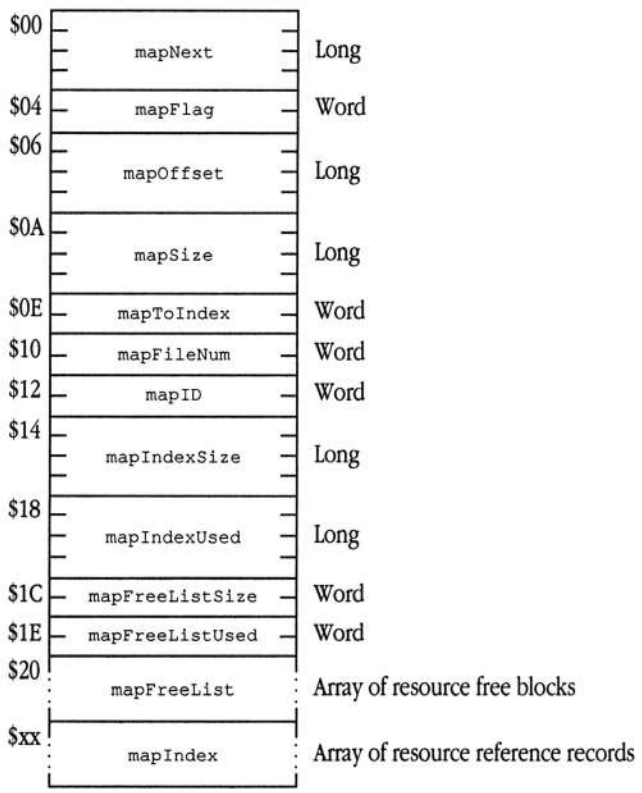
`rFileMapSize` Size, in bytes, of the resource map.

`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; Figure 45-4 shows the layout of the resource map.

■ **Figure 45-4** Resource map (`MapRec`)



`mapNext` Handle to resource map of next resource file in the search chain. Set to NIL if last file in chain. This field is valid only when the map is in memory.

`mapFlag` Contains control flags defining the state of the resource file.

Reserved	bits 15–2	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. 0 = Map not changed 1 = Map 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 changes in the resource or resource file made during program execution.

`mapToIndex` Offset, in bytes, from the beginning of the 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`.

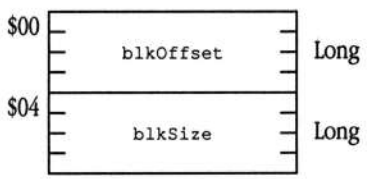
`mapFreeList` Array of resource free blocks, which describe free space in the resource file.

`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. Figure 45-5 shows the format of the resource free block.

■ **Figure 45-5** Resource free block (`FreeBlockRec`)



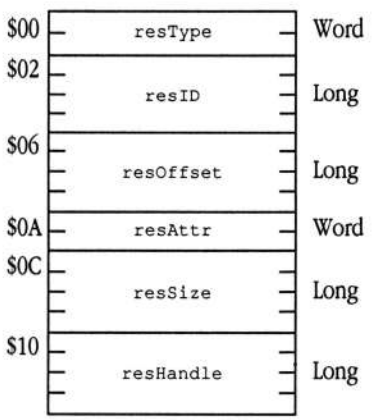
`blkOffset` Offset, in bytes, to the free block from the start of the resource fork. A NIL value indicates the end of the used blocks in the array.

`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 the `mapToIndex` field of the resource map (`MapRec`). Figure 45-6 shows the format of the resource reference record.

■ **Figure 45-6** Resource reference record (`ResRefRec`)



`resType`	Resource type. A NIL value indicates the 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. A 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 uses the `ResourceConverter` tool call to register a converter routine. 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 functions only 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 as changed, since the converter may write them back to disk in an unexpected format.

ReadResource

Reads 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. 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
```

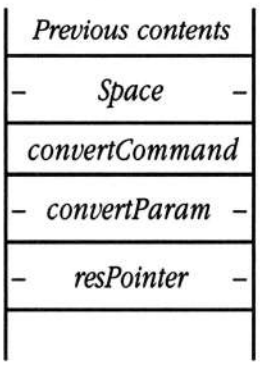
The *resPointer* parameter contains a pointer to the resource reference record, which contains location and size information about the resource in memory (see "Resource File Format" earlier in this chapter for information on the format and content of the resource reference record). Your program should 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 whether 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. In case of error, the routine should return a non-NIL result. It must also free the memory referenced by the `resHandle` field in the resource reference record and set that field to NIL.

Parameters

Stack before call



Long—Space for result

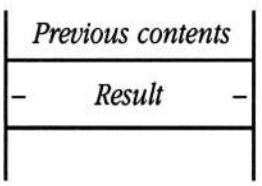
Word—Command to be performed (will be 0: `ReadResource`)

Long—Pointer to GS/OS read file parameter block

Long—Pointer to resource reference record

<—SP

Stack after call



Long—NIL if successful; error code if error (low-order word)

<—SP

WriteResource

Writes a resource from memory to 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 calls the `ReturnDiskSize` function in the converter routine to determine how much disk space the resource requires.

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* parameter 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 disposes of the handle to the resource after calling `WriteResource`.

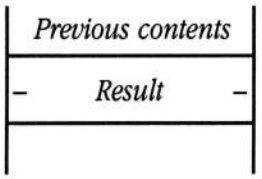
This function must return a NIL result.

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 disk space a resource will require and returns that value to the caller. Note that this call is not valid for resources that are loaded into absolute memory, because the size of these resources cannot change.

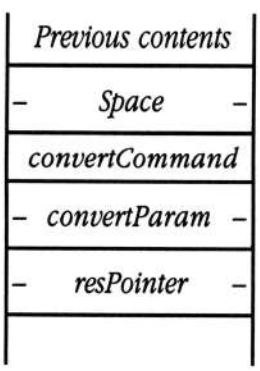
The *convertParam* parameter is undefined.

The *resPointer* parameter 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



Long—Space for result

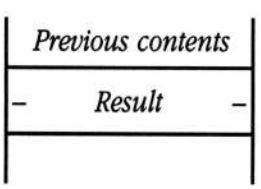
Word—Command to be performed (will be 4: `ReturnDiskSize`)

Long—Undefined

Long—Pointer to resource reference record

<—SP

Stack after call



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 responsibility of the application-switching program to use these calls.

In the following example, the Resource Manager is already active, and the application switcher has previously used the `ResourceStartUp` tool call to register itself with the Resource Manager. 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 order by call number.

`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 or output parameters.

Errors None

C This call must not be made by an application.

ResourceStartUp $021E

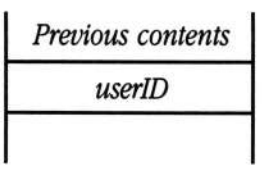
Notifies the Resource Manager that an application wishes to open and use its own resource files. Unlike other tool set `StartUp` calls, this call is not required in all circumstances. If your application uses only system resources (located in the system resource file), then it does not have to issue a `ResourceStartUp` tool call. By contrast, if your application uses nonsystem 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

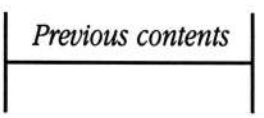
Stack before call



Word—Application user ID (obtained at program startup)

<—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 after other tool set shutdown calls, after this call the Resource Manager is still active. However, after calling `ResourceShutDown`, your application can access only 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

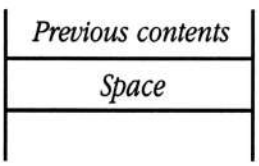
```
extern pascal void ResourceShutDown();
```

ResourceVersion $041E

Retrieves the Resource Manager version number. The *versionInfo* result contains the information in the standard format defined in Appendix A, "Writing Your Own Tool Set," in Volume 2 of the *Toolbox Reference.*

Parameters

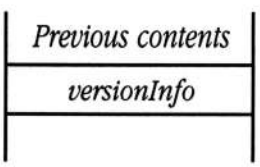
Stack before call



Word—Space for result

<—SP

Stack after call



Word—Resource Manager version number

<—SP

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 or output parameters.

Errors None

C This call must not be made by an application.

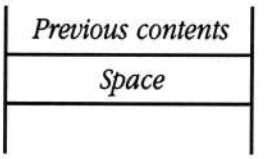
ResourceStatus $061E

Returns a flag indicating whether the Resource Manager is active. If the Resource Manager was loaded and initialized successfully at system startup, then this function returns a value of TRUE. If the Resource Manager was not successfully loaded or initialized, 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

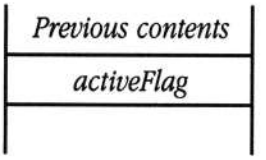
Stack before call



Word—Space for result

<—SP

Stack after call



Word—Boolean; TRUE if Resource Manager is active

<—SP

Errors $0002 `funcNotFoundErr` Resource Manager not active.

C

```
extern pascal Boolean ResourceStatus();
```

Resource Manager tool calls

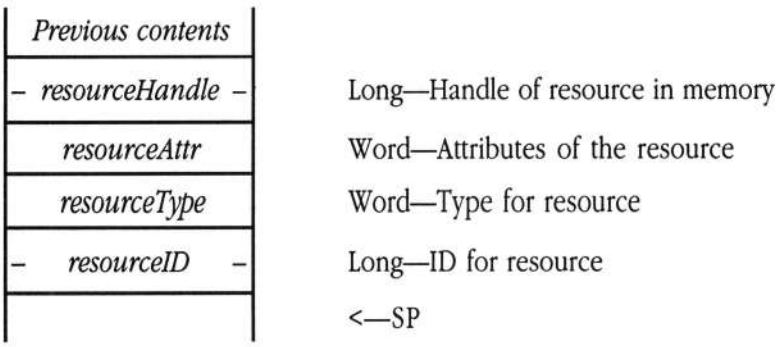
This section discusses the Resource Manager tool calls, in order by call name.

`AddResource` $0C1E

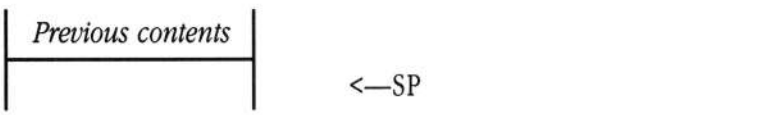
Adds a resource to the current resource file. The Resource Manager marks the new resource as changed and writes the new resource to disk when the file is updated. Your program specifies the attributes of the new resource in a flag word passed to `AddResource`. Some of these attributes control how memory 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



Stack after call



Errors

$1E04	`resNoCurFile`	No current resource file.
$1E05	`resDupID`	Specified resource ID is already in use.
$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` $181E" later in this chapter).

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 of 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

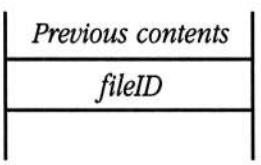
Updates a specified resource file, frees any memory used by the resource map for the file and any resources currently loaded, and closes the file. Your program passes the file ID of the resource file to be closed. This file ID is obtained from the `OpenResourceFile` tool call.

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 if you set the search depth so shallow that the system resource file is inaccessible (see the description of the `SetResourceFileDepth` tool call later in this chapter).

◆ *Note:* When quitting, your program need not issue `CloseResourceFile` calls for all open resource files. The `ResourceShutDown` call automatically updates and closes any open resource files.

Parameters

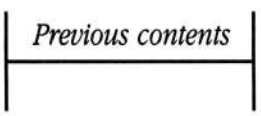
Stack before call



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;
```

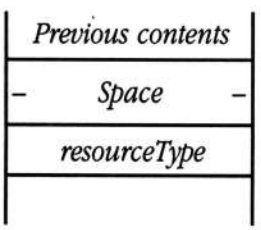
CountResources $221E

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, if it is in the search sequence.

◆ *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

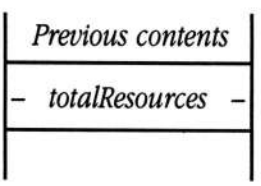


Long—Space for result

Word—Resource type to be counted

<—SP

Stack after call



Long—Number of resources of specified type

<—SP

Errors None

C

```
extern pascal Long CountResources(resourceType);

Word      resourceType;
```

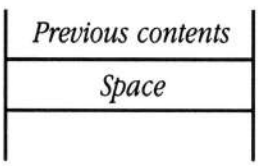
CountTypes $201E

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, if it is in the search sequence.

◆ *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

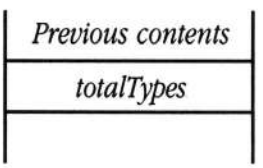
Stack before call



Word—Space for result

<—SP

Stack after call



Word—Number of different resource types

<—SP

Errors	Memory Manager errors	Returned unchanged.

C

```
extern pascal Word CountTypes();
```

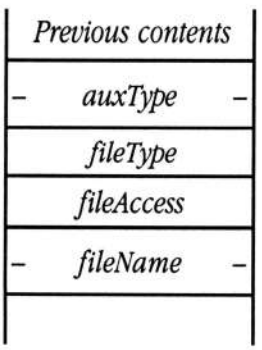
CreateResourceFile $091E

Initializes a resource fork with no resources. If necessary, `CreateResourceFile` creates the file to contain the resource fork. The specific actions performed by this call depend on the state of the specified input file.

No file of specified name	Create file with specified *auxType*, *fileType*, *fileAccess*, and *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 nonempty resource fork	Return `resForkUsed` error.

Parameters

Stack before call



Long—GS/OS auxiliary file type (used only if file does not exist)

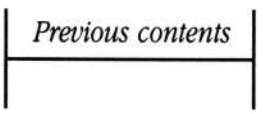
Word—GS/OS file type (used only if file does not exist)

Word—GS/OS file access (used only if file does not exist)

Long—Pointer to GS/OS class 1 input pathname for resource file

<—SP

Stack after call



<—SP

Errors

$1E01	`resForkUsed`	Resource fork not empty.
GS/OS errors		Returned unchanged.

C

```
extern pascal void CreateResourceFile(auxType,
          fileType, fileAccess, fileName);

Long      auxType, fileName;
Word      fileType, fileAccess;
```

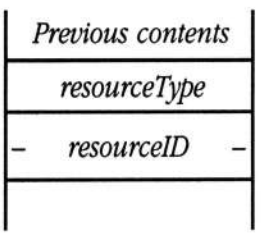
DetachResource $181E

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 you issue `DetachResource`, add the resource to the new resource file by calling `AddResource`. After you issue the `AddResource` call, the Resource Manager is again responsible for the resource handle.

Parameters

Stack before call

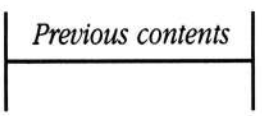


Word—Type of resource to be detached

Long—ID of resource to be detached

<—SP

Stack after call



<—SP

Errors

$1E06	`resNotFound`	Specified resource not found.
$1E0C	`resHasChanged`	Resource has been changed and has not been updated.

C

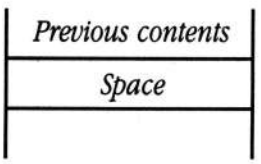
```
extern pascal void DetachResource(resourceType,
          resourceID);

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).

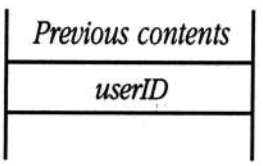
Parameters

Stack before call



Word—Space for result
<—SP

Stack after call



Word—User ID of current application; $401E if none
<—SP

Errors None

C

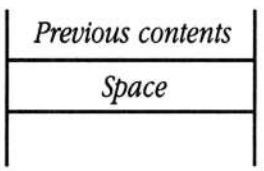
```
extern pascal Word GetCurResourceApp();
```

GetCurResourceFile $121E

Returns the file ID of the current resource file. This call returns a NIL value if there is no current file.

Parameters

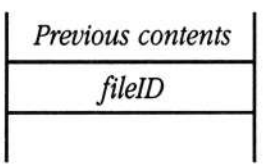
Stack before call



Word—Space for result

<—SP

Stack after call



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

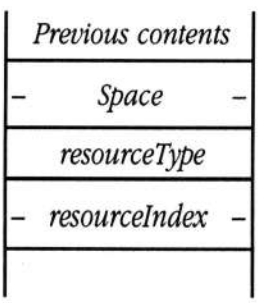
Finds a resource of a specified type by means of its index and returns the resource ID for that resource. The index value corresponds to the position of the desired resource among all resources of the specified type in all resource files available to the calling program in its search sequence; the first 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.

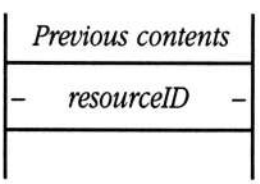
Parameters

Stack before call



Long—Space for result
Word—Type of resource to find
Long—Index of resource to find
<—SP

Stack after call



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

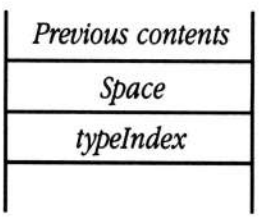
Finds a resource type value by means of its index. The index value corresponds to the 1-relative position of the desired resource type among all types in all resource files available to the calling program in its search sequence.

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.

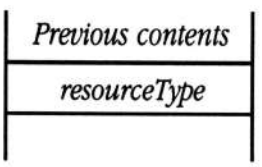
Parameters

Stack before call



Word—Space for result
Word—Index of type to find
<—SP

Stack after call



Word—Type matching index
<—SP

Errors

$1E0A	`resIndexRange`	Index is out of range (no resource found).
Memory Manager errors		Returned unchanged.

C

```
extern pascal Word GetIndType(typeIndex);

Word       typeIndex;
```

GetMapHandle $261E

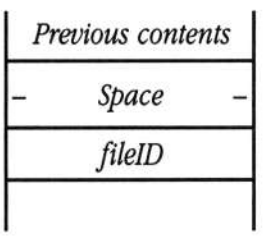
Returns a handle to the resource map for a specified resource file. Your program specifies the desired resource file by passing its file ID to `GetMapHandle`. This call searches all open resource files, irrespective of the search sequence in effect.

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

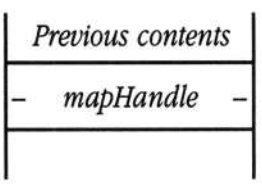


Long—Space for result

Word—ID of resource file to find

<—SP

Stack after call



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 of a particular resource file. However, this field also supports the following special values:

NIL	Returns handle to map of current resource file
$FFFF	Returns handle to map of 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.

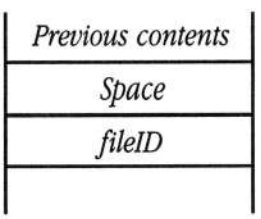
Your program may use this reference number to read data from the resource file. However, your program should be very careful to maintain the structure of the fork during write operations; careless writing could destroy the resource fork. Further, your program should never directly close the file using 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

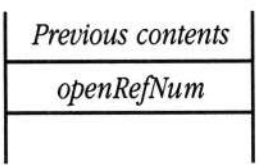


Word—Space for result

Word—ID of resource file to find

<—SP

Stack after call



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 of a particular resource file. However, this field also supports the following special values:

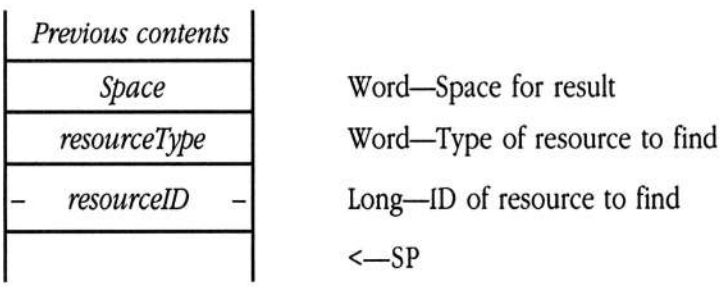
NIL	Returns reference number of current resource file
$FFFF	Returns reference number of system resource file

GetResourceAttr $1B1E

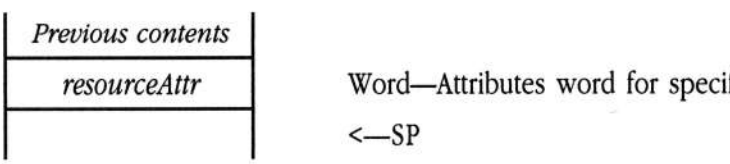
Returns the attributes word for a specified resource. Your program specifies the type and ID of the desired resource. For more information about the format and content of the attributes word, see "Resource Attributes" earlier in this chapter.

Parameters

Stack before call



Stack after call



Errors

$1E06	`resNotFound`	Specified resource not found.

C

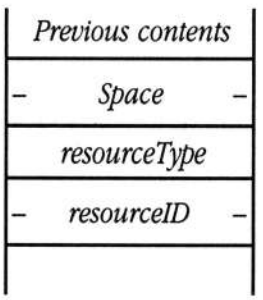
```
extern pascal Word GetResourceAttr(resourceType,
          resourceID);

Word      resourceType;
Long      resourceID;
```

GetResourceSize $1D1E

Returns the size of the specified resource. Your program specifies the type and ID of 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



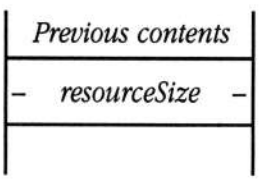
Long—Space for result

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;
```

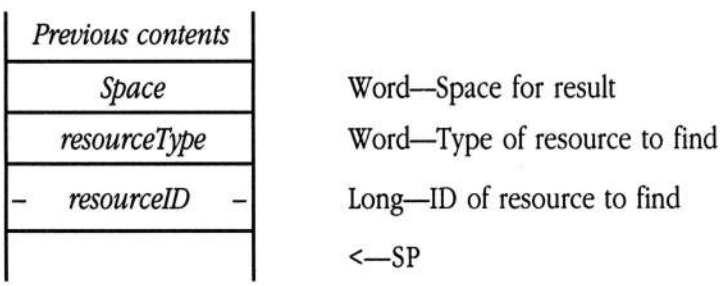
HomeResourceFile $151E

Returns the file ID of the resource file that contains a specified resource. Your program specifies the type and ID of the resource in question.

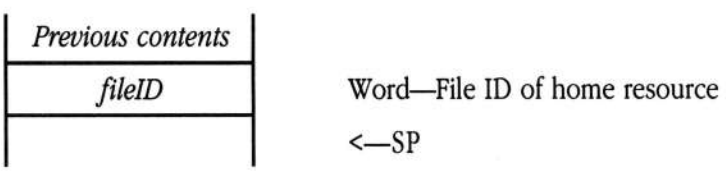
- *Note:* If multiple resources share the specified type and ID values, and your program has changed the resource search sequence (with the `SetCurResourceFile` or `SetResourceFileDepth` tool calls), the result of this call may be different from those of previous calls.

Parameters

Stack before call



Stack after call



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. To use this call you must have a thorough understanding of 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 found for resource type.
$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.

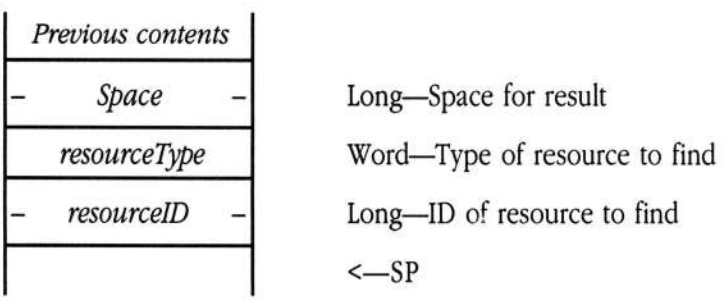
The `LoadResource` call searches both memory and disk for the specified resource. If the resource is already in memory, `LoadResource` returns a handle to that memory location. If the resource has been purged from memory, `LoadResource` reloads the resource and returns its handle. If the resource has 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 0 bytes). If you want the changes to be reflected in the resource file, use the `MarkResourceChange` tool call to set 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.

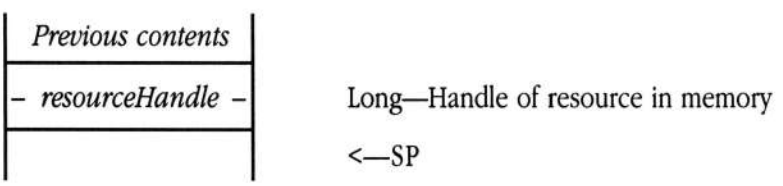
Note that your program should not dispose of the handle; only the Resource Manager should free the memory that it allocates.

Parameters

Stack before call



Stack after call



Errors

$1E03	`resNoConverter`	No converter routine found for resource type.
$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;
```

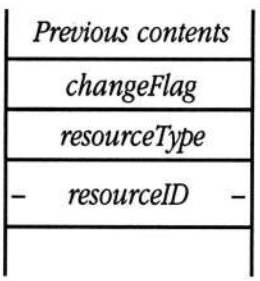
MarkResourceChange $101E

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 to make permanent the in-memory changes you have made to a resource.

Parameters

Stack before call



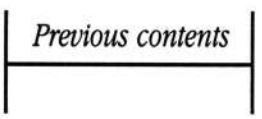
Word—Boolean; TRUE for changed, FALSE for not changed

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;
```

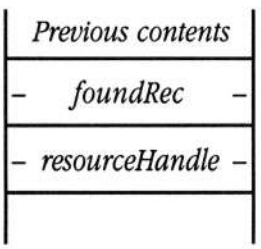
MatchResourceHandle $1E1E

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. As a consequence of the search algorithm used by the Resource Manager, the type and ID values returned by this call are unreliable if your program subsequently alters the resource search path (with the `SetCurResourceFile` or `SetResourceFileDepth` tool calls).

◆ *Note:* The Resource Manager has been optimized to access resources by type and ID, irrespective of the number of resources in the system. Although `MatchResourceHandle` works well with relatively small numbers of resources (less than 100), this call can be very slow when applied to files with large numbers of resources. To avoid this overhead, consider storing the resource type and ID in the resource structure, so that your program can access this information directly.

Parameters

Stack before call

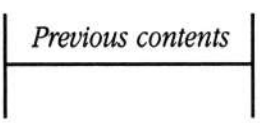


Long—Pointer to location in which to return type and ID

Long—Handle of resource

<—SP

Stack after call



<—SP

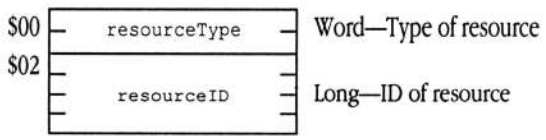
Errors

$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:

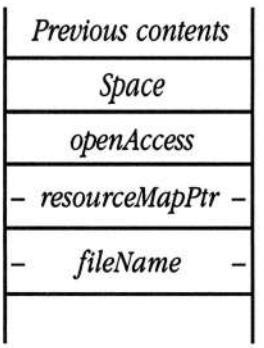


OpenResourceFile $0A1E

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 preloaded (`resPreLoad` flag is set to 1 in the attributes word for the resource).

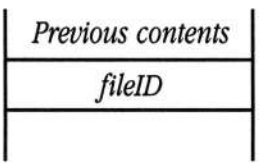
Parameters

Stack before call



Word—Space for result	
Word—File access	
Long—Pointer to resource map in memory	
Long—Pointer to GS/OS class 1 pathname of resource file	
<—SP	

Stack after call



Word—ID of open resource file	
<—SP	

Errors

$1E06	`resNotFound`	Specified resource not found.
$1E09	`resNoUniqueID`	No more resource IDs available.
$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,
          resourceMapPtr, fileName);

Word      openAccess;
Pointer   resourceMapPtr, fileName;
```

openAccess Contains GS/OS file access privileges for the resource file. See the *GS/OS Reference* for more information.

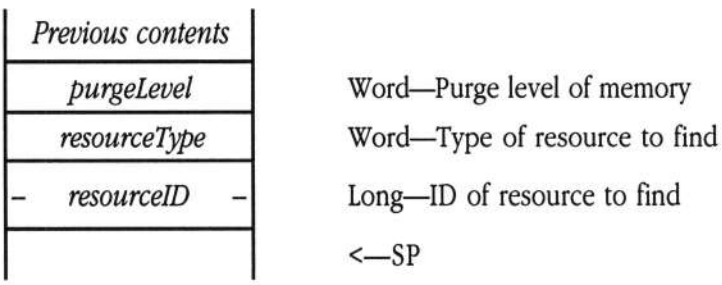
resourceMapPtr To open a resource file on disk, set this field to NIL. If the map is in memory, load this field with a pointer to that map. In this case, the Resource Manager opens the file that is already in memory.

ReleaseResource $171E

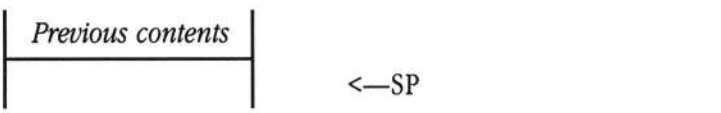
Sets the purge level of the memory used by a resource. Your program specifies the type and ID of the resource whose memory is to be freed and the purge level to be assigned to the memory. See Chapter 12, "Memory Manager," in Volume 1 of the *Toolbox Reference* for more information about purge levels and memory management. Note that this call does not unlock the handle.

Parameters

Stack before call



Stack after call



Errors

$1E06	`resNotFound`	Specified resource not found.
$1E0C	`resHasChanged`	Resource has been changed and has not been updated.

C

```
extern pascal void ReleaseResource(purgeLevel,
          resourceType, resourceID);

Word      purgeLevel, resourceType;
Long      resourceID;
```

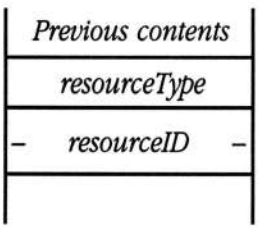
purgeLevel Specifies the Memory Manager purge level to be assigned to the freed memory. Valid Memory Manager purge levels lie in the range of 0 to 3. To direct the Resource Manager to dispose of the handle immediately, set this field to a negative value.

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

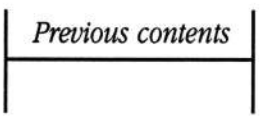


Word—Type of resource to find

Long—ID of resource to find

<—SP

Stack after call



<—SP

Errors

$1E06	`resNotFound`	Specified resource not found.
$1E0E	`resDiskFull`	Volume full.
Memory Manager errors		Returned unchanged.

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.

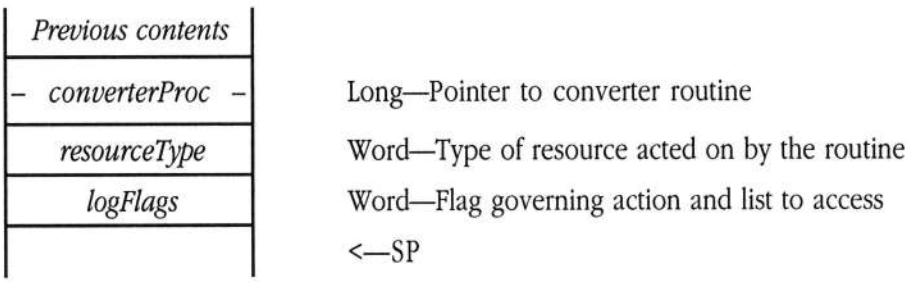
The Resource Manager maintains two classes of converter routine lists: one for your application and one for the system. Each application has its own converter routine list. All programs share access to the system list. When searching for a routine to convert a resource of a given type, the Resource Manager first searches the application list of 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



Stack after call

Errors

$1E0D	`resDiffConverter`	Another converter already logged in for this resource type.
Memory Manager errors		Returned unchanged.

C

```
extern pascal void ResourceConverter(converterProc,
          resourceType, logFlags);

Pointer   converterProc;
Word      resourceType, logFlags;
```

logFlags

Specifies whether to log the converter routine into or out of its list, and specifies which list (application or system) to access.

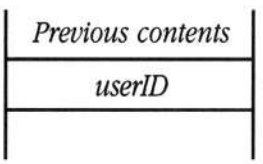
Reserved	bits 15–2	Must be set to 0.
`list`	bit 1	Indicates which routine list to access. 0 = Application converter list 1 = System converter list
`action`	bit 0	Specifies action to take. 0 = Log routine out of list 1 = Log routine into list

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 call `ResourceStartUp` to register itself with the Resource Manager.

Parameters

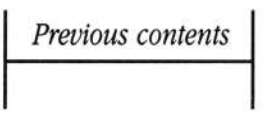
Stack before call



Word—User ID of application that will be using Resource Manager

<—SP

Stack after call



<—SP

Errors

$1E08	`resBadAppID`	User ID not found; calling program has not issued `ResourceStartUp` tool call.

C

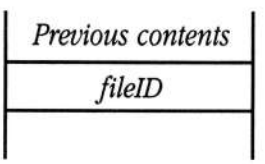
```
extern pascal void SetCurResourceApp(userID);

Word      userID;
```

SetCurResourceFile $111E

Makes a specified resource file the current file. Because Resource Manager searches typically start with the current resource file, your program can control the file search sequence by specifying a particular file as the current file. For more information about Resource Manager search processing, see "Using Resources" earlier in this chapter.

Parameters

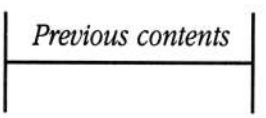
Stack before call



Word—File ID of resource file to be made current file

<—SP

Stack after call



<—SP

Errors

$1E07	`resFileNotFound`	Specified file ID does not match an open file.

C

```
extern pascal void SetCurResourceFile(fileID);

Word       fileID;
```

SetResourceAttr $1C1E

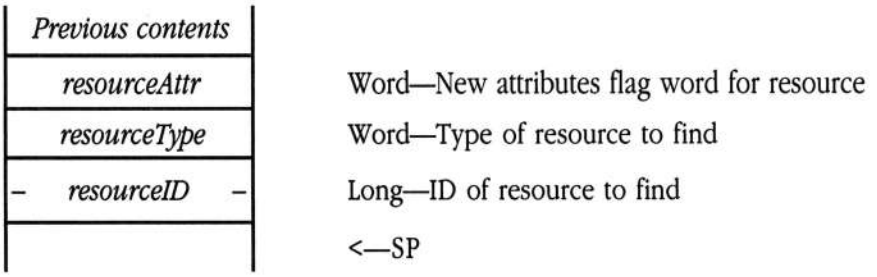
Sets the attributes of a resource. Your program specifies the type and ID of the desired resource and a new attributes word for the resource. The Resource Manager replaces the existing attributes word with the one provided to this call. For more information about the format and content of the attributes 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.

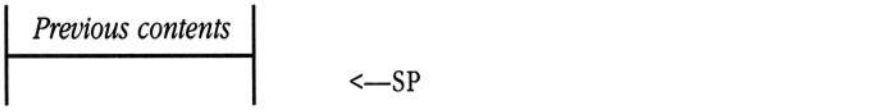
Note that these changes affect only future use of the resource. For example, if your program changes the attributes of a resource to indicate that it should be locked into memory (sets the `attrLocked` flag to 1), that action does not change the status of any current instances of that resource in memory. 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



Stack after call



Errors

$1E06	`resNotFound`	Specified resource not found.

C

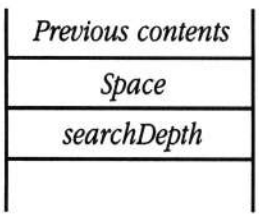
```
extern pascal void SetResourceAttr(resourceAttr,
          resourceType, resourceID);

Word      resourceAttr, resourceType;
Long      resourceID;
```

SetResourceFileDepth $251E

Sets the number of files the 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

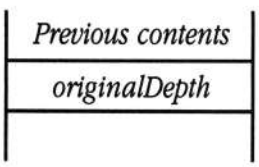


Word—Space for result

Word—Number of files to search

<—SP

Stack after call



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

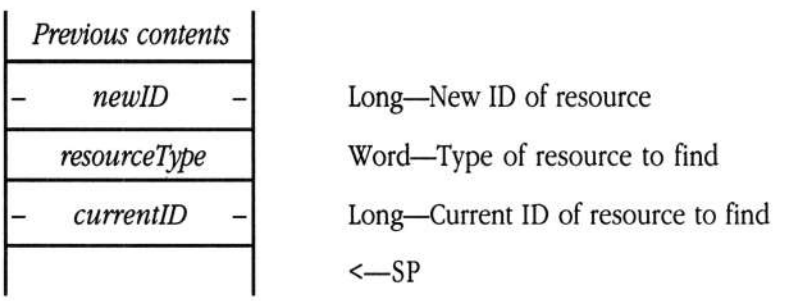
SetResourceID $1A1E

Changes the ID of a resource to a new value. Your program specifies the type and current ID of the resource to be changed.

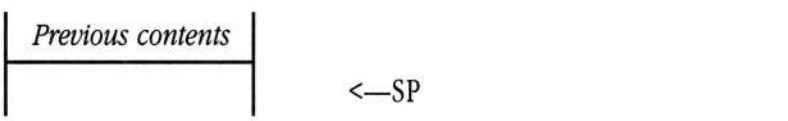
If your program changes the ID value of a resource, it should not mark the resource as changed. The Resource Manager automatically tracks these changes.

Parameters

Stack before call



Stack after call



Errors

$1E05	`resDupID`	Specified resource ID is already in use.
$1E06	`resNotFound`	Specified resource not found.

C

```
extern pascal void SetResourceID(newID,
          resourceType, currentID);

Long      newID, current ID;
Word      resourceType;
```

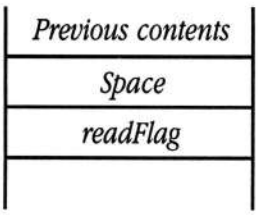
SetResourceLoad $241E

Controls Resource Manager access to the disk when resources are loaded. If you disable disk loading, the Resource Manager does not load resources from disk but instead allocates empty handles for requested resources. However, if a resource had been loaded into memory prior to the disabling of disk loading, the Resource Manager returns a valid handle. For example, a `LoadResource` tool call returns 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

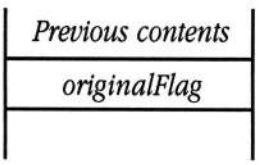


Word—Space for result

Word—Flag controlling Resource Manager disk access

<—SP

Stack after call



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.

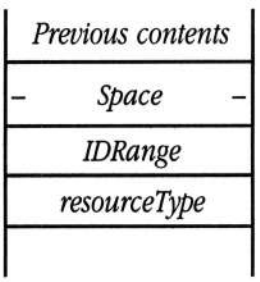
0	Do not read resources from disk
1	Read resources from disk, if necessary

UniqueResourceID $191E

Returns a unique resource ID for a specified resource type. Your program specifies the resource type of 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



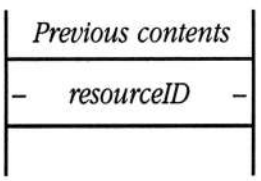
Long—Space for result

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	No current resource file.
$1E09	resNoUniqueID	No more resource IDs available.

C

```
extern pascal Long UniqueResourceID(IDRange,
              resourceType);

Word          IDRange, resourceType;
```

IDRange Specifies a 65,535-element range within which the 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 values. This facility is provided so that applications 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 list summarizes the valid values for *IDRange:*

IDRange	**Lowest possible ID returned**	**Highest possible ID returned**
$0000	$00000001 (zero 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

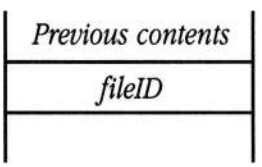
UpdateResourceFile $0D1E

Transfers modifications made to resources in memory to the appropriate 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` writes 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

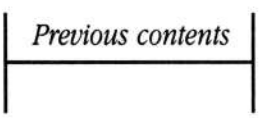
Stack before call



Word—ID of open resource file

<—SP

Stack after call



<—SP

Errors

$1E03	`resNoConverter`	No converter routine found for resource type.
$1E07	`resFileNotFound`	Specified file ID does not match an open file.
$1E0E	`resDiskFull`	Volume full.
GS/OS errors		Returned unchanged.

C

```
extern pascal void UpdateResourceFile(fileID);

Word       fileID;
```

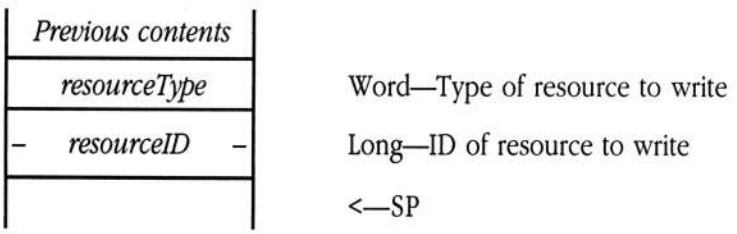
WriteResource $161E

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 word), the Resource Manager writes the resource to its resource file on disk. The `AddResource`, `MarkResourceChange`, or `SetResourceAttr` (with `resChanged` set to 1) tool calls cause a resource to be marked as changed.

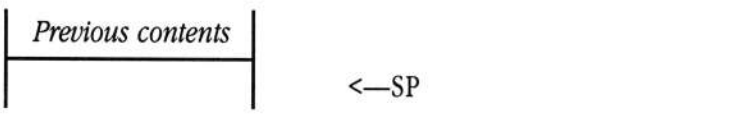
- *Note:* Most applications will not issue this call because the `ResourceShutDown`, `CloseResourceFile`, and `UpdateResourceFile` tool calls automatically write all changed resources to the appropriate resource file (unless the resource is write-protected).

Parameters

Stack before call



Stack after call



Errors

$1E03	`resNoConverter`	No converter routine found for resource type.
$1E06	`resNotFound`	Specified 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

Tables 45-1, 45-2, and 45-3 summarize the constants, data structures, and error codes (respectively) 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 `OpenResourceFile` should be used to load the resource into memory.
`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 a converter routine is required as the resource is loaded into memory or written to disk.
`resMemAttr`	$C31C	Flags passed to the `NewHandle` Memory Manager tool call when memory is allocated 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	Long	Format version of resource fork
`rFileToMap`	$0004	Long	Offset from start of fork to resource map record
`rFileMapSize`	$0008	Long	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	Long	Offset from start of fork to resource map record
`mapSize`	$000A	Long	Size, in bytes, of resource map
`mapToIndex`	$000E	Word	Offset from start of map to the `mapIndex` array
`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	Long	Total number of resource reference records in `mapIndex`
`mapIndexUsed`	$0018	Long	Number of used resource reference records
`mapFreeListSize`	$001C	Word	Total number of free block records in the `mapFreeList` array
`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`)

[continued]

■ **Table 45-2** Resource Manager data structures [continued]

Name	Offset/Value	Type	Description
`FreeBlockRec` (free block record)			
`blkOffset`	$0000	Long	Offset, in bytes, to start of this block of free space
`blkSize`	$0004	Long	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	Long	Resource ID
`resOffset`	$0006	Long	Offset, in bytes, from start of resource fork to this resource
`resAttr`	$000A	Word	Attribute bit flags for the resource
`resSize`	$000C	Long	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 found for resource type.
$1E04	`resNoCurFile`	No current resource file.
$1E05	`resDupID`	Specified resource ID is already in use.
$1E06	`resNotFound`	Specified resource not found.
$1E07	`resFileNotFound`	Specified ID does not match an open file.
$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 (no resource found).
$1E0B	`resSysIsOpen`	System resource file is already open.
$1E0C	`resHasChanged`	Resource has been changed and has not been updated.
$1E0D	`resDiffConverter`	Another 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 contains corrections to the documentation of the Sound Tool Set in Volume 2 of the *Toolbox Reference.*

- The documentation of the `FFSoundDoneStatus` call contains an error. You will note that the paragraph that describes the call does not agree with the diagram describing the stack after the call. 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.
- The description of the `GetSoundVolume` tool call is misleading with respect to the number of significant bits in the returned volume setting. The text accompanying the stack diagram is correct—only the high nibble of the low-order byte contains valid volume data.
- The `FFGeneratorStatus` tool call can return error code $0813, indicating that the *genNumber* parameter contains an invalid generator number.

Clarification

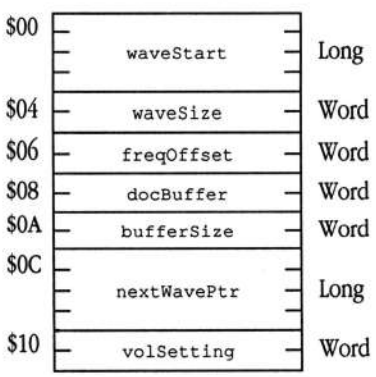
This section presents more complete information about the `FFStartSound` tool call, including further explanation of its parameters, a new error code, an example procedure for moving a sound from the Macintosh computer to the Apple IIGS computer, and some sample code demonstrating the use of the call. The original documentation for this call is in Chapter 21, "Sound Tool Set," in Volume 2 of the *Toolbox Reference.*

FFStartSound

The free-form synthesizer is designed to play back long waveforms. 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 4 KB buffers, then those buffers must be aligned on 4 KB memory boundaries.

Parameter block



`waveStart` The starting address of the wave to be played, not in Digital Oscillator Chip (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 discussion of those registers in "New Information" later in this chapter 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 3 bits set the values for the table-size and resolution portions of the DOC Bank-Select/Table-Size/Resolution register.
`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 computer to the Apple IIGS computer

To move a digitized sound from the Macintosh computer to the Apple IIGS computer and play the sound, you perform the following steps:

1. Save the sound as a pure data file on the Macintosh computer.
2. Transfer the file to the Apple IIGS computer (using Apple File Exchange, for example).
3. Filter all the 0 sample bytes out of the file by replacing them with bytes set to $01. This is very important, because the Apple IIGS computer interprets 0 bytes as the end of a sample.
4. Load the sound into memory with GS/OS calls.
5. Issue the `FFStartSound` tool call to play the sound. Set the `freqOffset` parameter to $01B7 to match the tempo at which the sound is played on the Macintosh computer, assuming that you recorded the original sound at the standard Macintosh sampling rate of 22 kHz.

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 param block
            _FFStartSound               ; Start free-form 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 their versions must be compatible. The currently required versions are

Note Synthesizer	version 1.3
Note Sequencer	version 1.3
MIDI Tool Set	version 1.2
Sound Tool Set	version 2.4

- 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. See the description in Volume 2 of the *Toolbox Reference*. Note also that your interrupt handler should check the synthesizer mode value to verify that it should handle the interrupt. This mode value is passed as an input parameter to the interrupt handler in the accumulator register.

  If your routine does not process the interrupt, it should jump 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 computer

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 Note Sequencer, the Note Synthesizer, the MIDI Tool Set, the Sound 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** plays 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 sound sample, including the ability to loop within the sample and control the sound envelope. The Note Synthesizer, however, is limited to sound samples 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 computer into a MIDI sound source with the capability 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, the tool set 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 of 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, the Note Sequencer immediately fetches the next seqItem for execution. If the `d` (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, 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 `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 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 accept as input a waveform stored as digital data, and generate an audio signal based on that 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 one 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 voice is 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 computer 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. The sample rate is determined by the formula

$$S = \frac{\left(\frac{C}{8}\right)}{(O+2)}$$

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, the Apple IIGS computer, 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 computer 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 allows 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. Normally frequency refers to the value of the frequency register, which determines, but is not equal to, the output frequency. Frequency directly determines the perceived pitch of a sound; higher frequencies result in higher pitches.

Sound RAM

The DOC has 64 KB of RAM dedicated to the 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

$$O = \left(\frac{S}{2^{(17+R)}}\right) * FHL$$

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) with all 32 oscillators enabled
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 the Frequency Low and Frequency High registers; 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:

$$O = \left(\frac{S}{SRi}\right) * \left(\frac{Fi * FHL}{2^{(9+R\text{-}TAB)}}\right)$$

where

O	is the output frequency in hertz
Fi	is the frequency of the sampled waveform in hertz
SRi	is the rate at which you sampled the original sound (in samples per second)
S	is the Apple IIGS sample rate (26.32 kHz) with all 32 oscillators enabled
FHL	is the combined values of the Frequency Low and Frequency High registers; 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	is the 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 what oscillator mode is in effect, whether the oscillator is halted, whether it will generate an interrupt at the end of its cycle, and what channel has been assigned to the oscillator.

- **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 21, "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 of 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 to other interrupt routines (see the discussion of the `SetUserSoundIRQV` tool call in Chapter 21, "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 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.

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 set to 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** The Channel value 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 Waveform Table Pointer register are accessible to the DOC for this operation; a larger table size reduces the number of Waveform Table Pointer register bits used in the address calculation and reduces the precision with which a particular sample can be located. If 8 bits of the Waveform Table Pointer register are used to locate the next sample, the DOC can distinguish twice as many starting points as it can if only 7 bits are used.

Each time the DOC cycles it adds the contents of the frequency registers to its 24-bit accumulator. It then appends a subrange of the accumulator's 24 bits to the value of the Waveform Table Pointer register 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 3 bits of the Bank-Select/Table-Size/Resolution register, determines the lowest bit of the accumulator value that will be appended to the Waveform Table Pointer register.

The table size value, which is the next 3 bits above the resolution, determines both the 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 8 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 where in memory to find the sounds 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 Waveform Table 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	Address Pointer Register								Accumulator Bits								1	1	1
	7							0	23 16							16 9	0	0	0
512																	1	1	1
	7							1	23 16							15 8	0	0	0
1024																	1	1	1
	7							2	23 16							14 7	0	0	0
2048																	1	1	1
	7							3	23 16							13 6	0	0	0
4096																	1	1	1
	7							4	23 16							12 5	0	0	0
8192																	1	1	1
	7							5	23 16							11 4	0	0	0
16,384																	1	1	1
	7						6		23 16							10 3	0	0	0
32,768																	1	1	1
	7								23 16							9 2	0	0	0

The resolution acts as an offset value, determining which bit is the lowest accumulator bit to be appended to the Waveform Table 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 following list shows the correspondence between table size values and the table sizes.

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 value and resolution value are copied into their respective bits in the Bank-Select/Table-Size/Resolution register from the lowest 3 bits of the *buffersize* parameter to the `FFStartSound` call.

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 is 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, a `JSL` and an `RTL`.

New Sound Tool Set calls

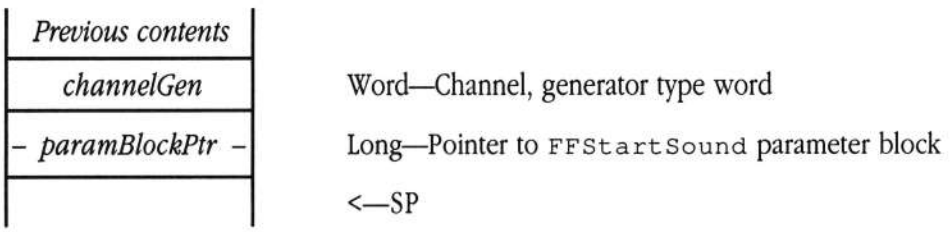
Four new tool calls provide greater flexibility for applications playing free-form sounds. The `FFSetUpSound` and `FFStartPlaying` calls allow you to schedule a sound for playback at a later time. The `ReadDOCReg` and `SetDOCReg` calls provide easy access to the DOC registers.

`FFSetUpSound` $1508

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



Stack after call

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 21, "Sound Tool Set," in Volume 2 of the *Toolbox Reference.*

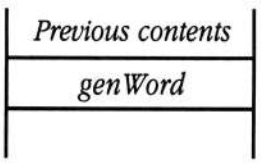
paramBlockPtr For complete information on the parameter block pointed to by the *paramBlockPtr* parameter, see "`FFStartSound`" earlier in this chapter.

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

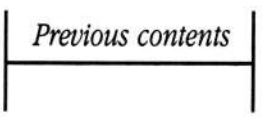
Stack before call



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. ▲

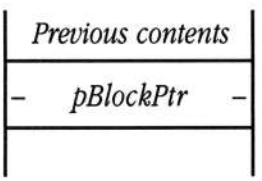
ReadDOCReg $1808

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 the 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. Do 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

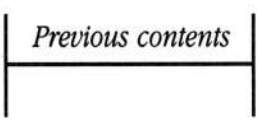
Stack before call



Long—Pointer to DOC register parameter block

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void ReadDOCReg(pBlockPtr);

Pointer   pBlockPtr;
```

pBlockPtr 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.
0 = Do not get the registers
1 = Get the registers

bit 14 Determines whether to get DOC registers for the second oscillator.
0 = Do not get the registers
1 = Get the registers

bits 13–12 Reserved; must be set to 0.

bits 11–8 Specify the generator number for the operation. Valid values lie in the range from $0 through $F.

bits 7–4 Reserved; must be set to 0.

bits 3–0 Specify who is using the generator (this value is returned).

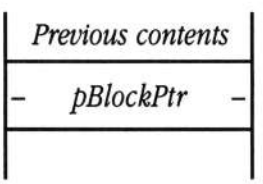
SetDOCReg $1708

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 that of the output from the `ReadDOCReg` tool call.

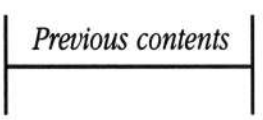
▲ Warning This is a very low-level call. Do not use it unless you have a thorough understanding of the DOC. This call may not be supported in future versions of the system hardware. ▲

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal void SetDOCReg(pBlockPtr);

Pointer   pBlockPtr;
```

pBlockPtr 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.
0 = Do not set the registers
1 = Set the registers

bit 14 Determines whether to set DOC registers for the second oscillator.
0 = Do not set the registers
1 = Set the registers

bits 13–12 Reserved; must be set to 0.

bits 11–8 Specify the generator number for the operation. Valid values lie in the range from $0 through $F.

bits 7–4 Reserved; must be set to 0.

bits 3–0 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 Update

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 of the Standard File Operations Tool Set

This section explains new features of the Standard File Operations Tool 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 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. Although 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 at least four pages of RAM available on the application stack (three for the List Manager and 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.

  Whenever the user changes the pathname, even in a Standard File dialog box that is subsequently canceled, 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 displays 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 displays a detailed information dialog box and allows 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 a slash (/), but now it is typically a colon (:).
- The system now disables the Save and New Folder buttons in Put dialog boxes referencing write-protected volumes. In addition, the system now displays a lock icon for such volumes.
- The Standard File Operations Tool Set now supports multiple file Get calls, which are collectively referred to as *multifile* calls. See "New or Changed Standard File Calls" later in this chapter for call syntax details.

  Multifile dialog boxes 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 returns the filename. If the file is a folder, clicking Open lists the folder contents, and clicking Accept returns the folder name to the calling program. Double-clicking a file has the same effect as clicking Accept; double-clicking a folder has the same effect as clicking Open.
  - When the user has selected multiple files, the Open button is disabled. The user must click Accept 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 box templates. The system automatically changes custom dialog box templates to use static text rather than user items. In addition, the system now uses a custom item-drawing routine for the path entry item. The system automatically changes input dialog box templates to call the Standard File Operations Tool Set's custom item-drawing 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 dialog boxes 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 the *GS/OS Reference.*

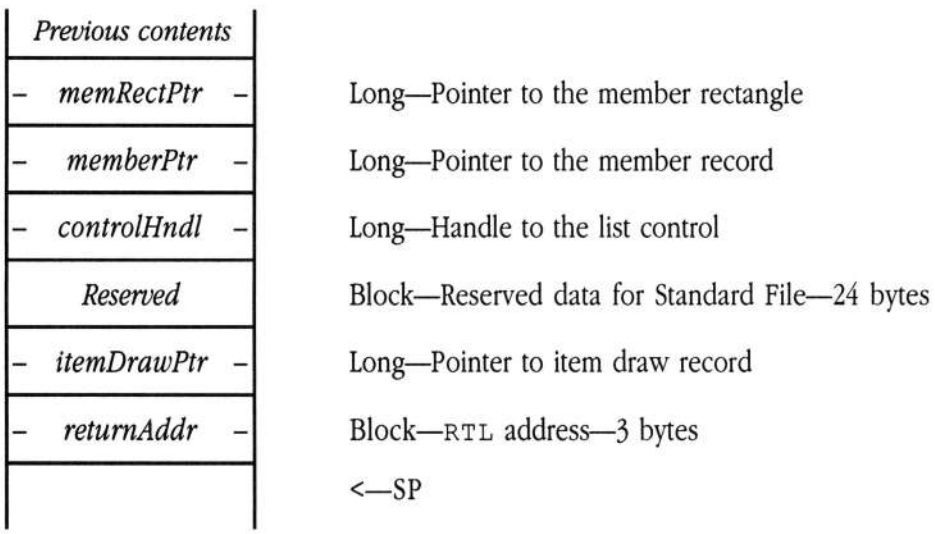
The exit interface from these filter procedures has not changed. Your program must remove the input pointer from the stack and return a response word indicating how the current file is to be displayed in the file list.

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

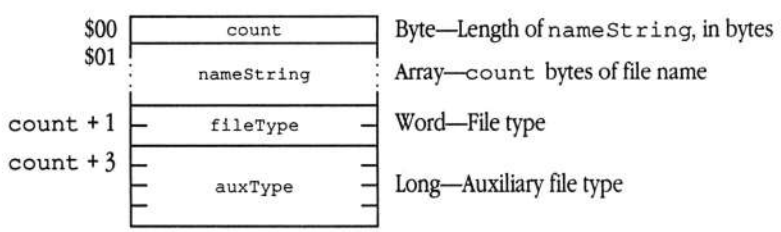
Custom item-drawing routines

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, the Standard File Operations Tool Set formats the stack as follows:



itemDrawPtr Pointer to a record formatted as follows:



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 item-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 the font or the character size is not recommended. Note that any icons must also comply with these restrictions (currently icons are eight lines high).

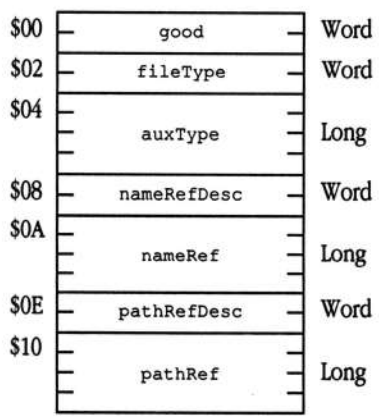
Standard File data structures

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



`good` Boolean indicating the status of the request. TRUE indicates that the user opened the file; FALSE indicates that the user canceled the request.

`fileType` The GS/OS file type information.

`auxType` The GS/OS auxiliary type information.

`nameRefDesc` 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 on 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` 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 on 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.

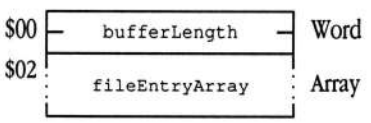
Multifile reply record

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

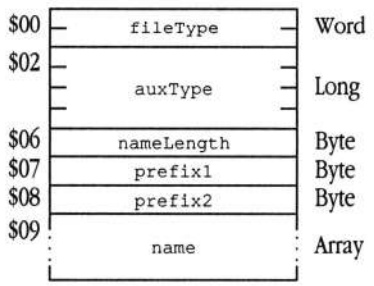
`good` Either the number of files selected, or FALSE if the user canceled the request.

`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` 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:

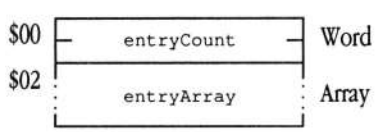


`fileType`	The GS/OS file type.
`auxType`	The GS/OS auxiliary type.
`nameLength`	The length of the following filename, including the volume prefix bytes (`prefix1` and `prefix2`).
`prefix1`	Volume prefix for the pathname, first byte. Always set to 8.
`prefix2`	Volume prefix for the pathname, second byte. Always set to a colon (`:`).
`name`	Filename string, containing (`nameLength` = 2) bytes of data, not to exceed 253 characters.

File type list record

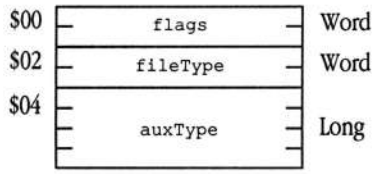
Figure 48-3 shows the layout of the Standard File type list record. You use this record with new Standard File calls that require a file type list as input (such as `SFGetFile2`).

■ **Figure 48-3** File type list record



`entryCount` 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 `auxType` when selecting files to be displayed.

bit 15	Controls whether Standard File cares about auxiliary types. 0 = Match only the specified `auxType` value 1 = Match any `auxType` value
bit 14	Controls whether Standard File cares about file types. 0 = Match only the specified `fileType` value 1 = Match any `fileType` value
bit 13	Disable selection. 0 = Make all displayed files selectable 1 = Display as dimmed, and thus unselectable, any files matching criteria specified in bits 14 and 15 (Note that the files 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. If you set both bits to 1, the Standard File Operations Tool Set will match all files.

`fileType` The GS/OS file type value to match, according to the settings of the `flags` bits.

`auxType` The GS/OS auxiliary type value to match, according to the settings of the `flags` bits.

Standard File dialog box 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 box template* to the appropriate Standard File routine (`SFPPutFile2`, `SFPGetFile2`, or `SFPMultiFile2`). The Standard File Operations Tool Set passes the dialog box template to the Dialog Manager (`GetNewModalDialog` call) when it establishes the user dialog box.

Although 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 box 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 of the rectangle in pixels 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 box 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       'Disk'
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.

  The files item (item 7) contains the boundary rectangle for the List Manager and serves no other purpose.

First, here are 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, here are 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.

  The files item (item 8) contains the boundary rectangle for the List Manager and serves no other purpose.

First, here are 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 a 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, here are 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 a 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

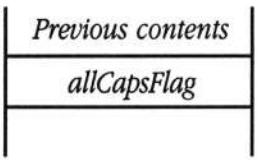
The following sections discuss several new or changed Standard File tool calls.

SFAllCaps $0D17

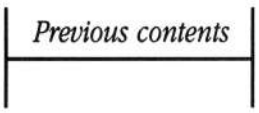
This call has been disabled so that filenames will appear exactly as entered. Existing programs may still issue the call, but it will have no effect.

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal void SFAllCaps(allCapsFlag);

Boolean   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—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 `nameRefDesc` value in the reply record.
$1705	`badReplyPathDesc`	Invalid `pathRefDesc` value in the reply record.
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 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

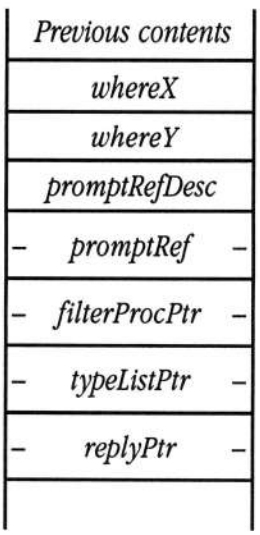
filterProcPtr Pointer to a new-style filter procedure, as described in "New Filter Procedure Entry Interface" earlier in this chapter.

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.

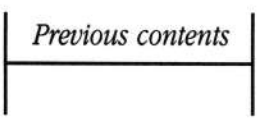
Parameters

Stack before call



Word—x coordinate of upper-left corner of dialog box
Word—y coordinate of upper-left corner of dialog box
Word—Type of reference in *promptRef*
Long—Reference to Pascal string for file prompt
Long—Pointer to filter procedure; NIL for none
Long—Pointer to type list record; NIL for none
Long—Pointer to multifile reply record
<—SP

Stack after call



<—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 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

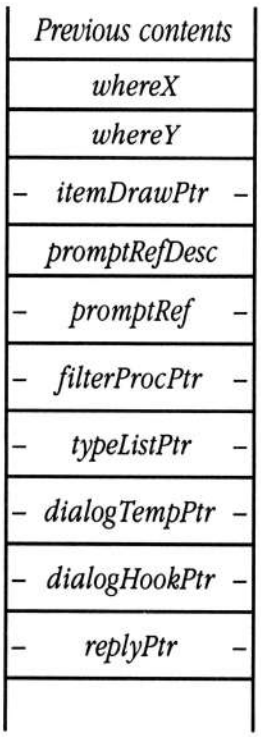
filterProcPtr Pointer to a new-style filter procedure, as described in "New Filter Procedure Entry Interface" earlier in this chapter.

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



Word—x coordinate of upper-left corner of dialog box

Word—y coordinate of upper-left corner of dialog box

Long—Pointer to item-drawing routine; NIL for none

Word—Type of reference in *promptRef*

Long—Reference to Pascal string for file prompt

Long—Pointer to filter procedure; NIL for none

Long—Pointer to type list record; NIL for none

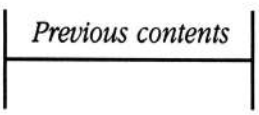
Long—Pointer to dialog box template

Long—Pointer to routine to handle item hits

Long—Pointer to new-style reply record

<—SP

Stack after call



<—SP

Errors

$1701	`badPromptDesc`	Invalid *promptRefDesc* value.
$1704	`badReplyNameDesc`	Invalid `nameRefDesc` value in the reply record.
$1705	`badReplyPathDesc`	Invalid `pathRefDesc` value in the reply record.
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 that the Standard File Operations Tool Set tried to return.

C

```
extern pascal void SFPGetFile2(whereX, whereY,
          itemDrawPtr, promptRefDesc, promptRef,
          filterProcPtr, typeListPtr, dialogTempPtr,
          dialogHookPtr, replyPtr);

Pointer   itemDrawPtr, filterProcPtr, typeListPtr,
          dialogTempPtr, 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 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.

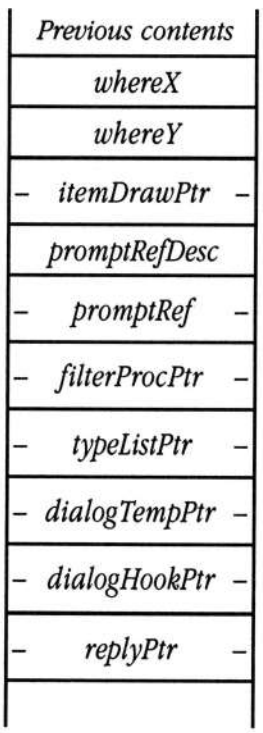
dialogTempPtr, dialogHookPtr
For more information about these fields, see the discussion of the `SFPPutFile` call in Chapter 22, "Standard File Operations Tool Set," in Volume 2 of the *Toolbox Reference.*

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



Word—x coordinate of upper-left corner of dialog box

Word—y coordinate of upper-left corner of dialog box

Long—Pointer to item-drawing routine; NIL for none

Word—Type of reference in *promptRef*

Long—Reference to Pascal string for file prompt

Long—Pointer to filter procedure; NIL for none

Long—Pointer to type list record; NIL for none

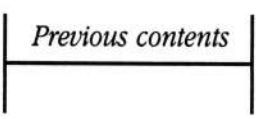
Long—Pointer to dialog box template

Long—Pointer to routine to handle item hits

Long—Pointer to multifile reply record

<—SP

Stack after call



<—SP

Errors

$1701	`badPromptDesc`	Invalid *promptRefDesc* value.

C

```
extern pascal void SFPMultiGet2(whereX, whereY,
          itemDrawPtr, promptRefDesc, promptRef,
          filterProcPtr, typeListPtr, dialogTempPtr,
          dialogHookPtr, replyPtr);

Pointer   itemDrawPtr, filterProcPtr, typeListPtr,
          dialogTempPtr, 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 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.

dialogTempPtr, dialogHookPtr
For more information about these fields, see the discussion of the `SFPPutFile` 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-drawing routine; NIL for none
promptRefDesc	Word—Type of reference in *promptRef*
promptRef	Long—Reference to Pascal string for file prompt
origNameRefDesc	Word—Type of reference in *origNameRef*
origNameRef	Long—Reference to GS/OS class 1 input string with default name
dialogTempPtr	Long—Pointer to dialog box 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 *origNameRefDesc* value.
$1704	`badReplyNameDesc`	Invalid `nameRefDesc` value in the reply record.
$1705	`badReplyPathDesc`	Invalid `pathRefDesc` value in the reply record.
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 that the Standard File Operations Tool Set tried to return.

C

```
extern pascal void SFPPutFile2(whereX, whereY,
          itemDrawPtr, promptRefDesc, promptRef,
          origNameRefDesc, origNameRef,
          dialogTempPtr, dialogHookPtr, replyPtr);

Pointer   itemDrawPtr, dialogTempPtr, 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" earlier in this chapter

promptRefDesc 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 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.

dialogTempPtr, dialogHookPtr

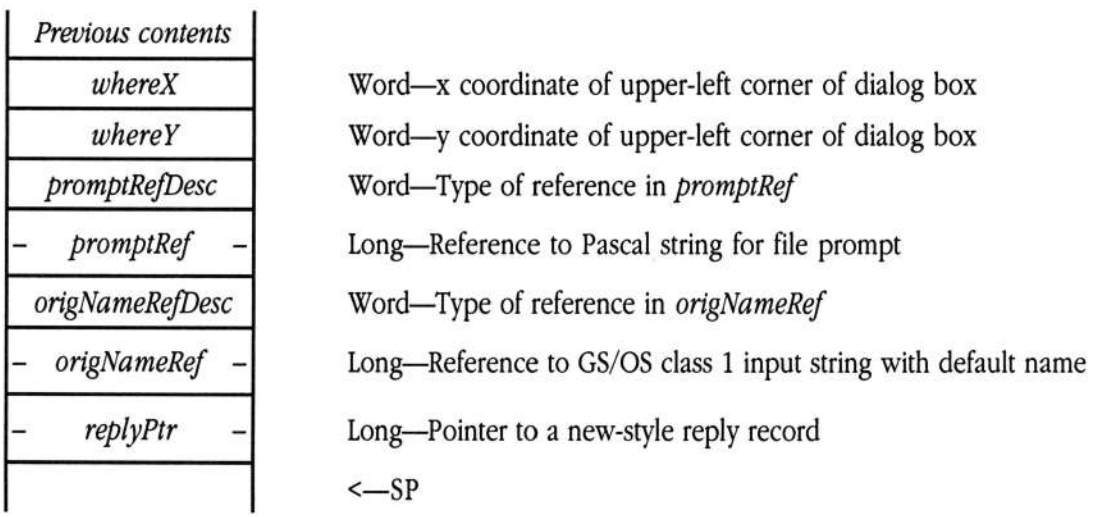
For more information about these fields, see the discussion of the `SFPPutFile` call in Chapter 22, "Standard File Operations Tool Set," in Volume 2 of the *Toolbox Reference.*

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 `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



Stack after call

Errors

$1701	`badPromptDesc`	Invalid *promptRefDesc* value.
$1702	`badOrigNameDesc`	Invalid *origNameRefDesc* value.
$1704	`badReplyNameDesc`	Invalid `nameRefDesc` value in the reply record.
$1705	`badReplyPathDesc`	Invalid `pathRefDesc` value in the reply record.
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 that 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 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 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.

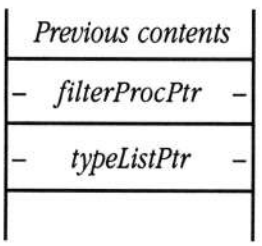
SFReScan $1317

Forces the system to rebuild and redisplay 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` is 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

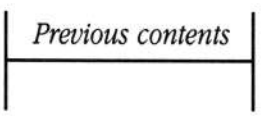


Long—Pointer to filter procedure; NIL for no change

Long—Pointer to type list record; NIL for no change

<—SP

Stack after call



<—SP

Errors

$1706	`badCall`	`SFPGetFile`, `SFPGetFile2`, and `SFPMultiGet2` are not active.

C

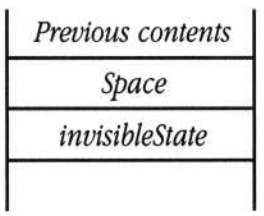
```
extern pascal void SFReScan(filterProcPtr,
          typeListPtr);

Pointer   filterProcPtr, typeListPtr;
```

SFShowInvisible $1217

Controls the 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

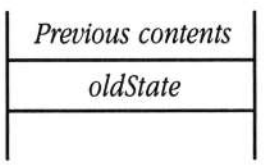


Word—Space for result

Word—Flag: 1 = display invisible files; 0 = no display (default)

<—SP

Stack after call



Word—Previous setting of invisible flag

<—SP

Errors None

C

```
extern pascal word SFShowInvisible(invisibleState);

Word      invisibleState;
```

Standard File error codes

Table 48-1 lists all valid Standard File error codes.

■ **Table 48-1** Standard File error codes

Value	Name	Definition
$1701	`badPromptDesc`	Invalid *promptRefDesc* value.
$1702	`badOrigNameDesc`	Invalid *origNameRefDesc* value.
$1704	`badReplyNameDesc`	Invalid `nameRefDesc` value in the reply record.
$1705	`badReplyPathDesc`	Invalid `pathRefDesc` value in the reply record.
$1706	`badCall`	`SFPGetFile`, `SFPGetFile2`, and `SFPMultiGet2` are not active.

Chapter 49 **TextEdit Tool Set**

This chapter documents the features of the 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. Although it has been loosely based on the Macintosh tool set of the same name, 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, and cutting and pasting, all with 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, automatic scrolling

TextEdit provides your program with the facilities to create, display, and manage one or more blocks of editable text. These blocks can be controls (such as the text in a dialog box or the text window for an editor) or they can be 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 blocks that are not controls. All TextEdit blocks, whether or not they are controls, are called **records** or **TextEdit records.**

Because many TextEdit records can be displayed at the same time, TextEdit provides a mechanism for distinguishing between them. This works much like the facility the Control Manager uses to move among controls in a window. The current or active TextEdit record is referred to as the **target** record. That record receives all user keystrokes and mouse clicks. 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 of a TextEdit record. All control information needed for the record is either stored in or 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` TextEdit tool call or the `NewControl2` Control Manager 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 can make the following adjustments to the current selection:

- When text is cut, TextEdit removes all leading spaces; if there are no leading spaces, it removes 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, 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 (but not characters) 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 record 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 list of tool calls, grouped according to function, summarizes the capabilities of the TextEdit Tool Set. Later sections of this chapter discuss TextEdit and its capabilities and data structures in greater detail and define the precise syntax of the TextEdit tool calls.

Routine	Description
Housekeeping routines	
`TEBootInit`	Initializes TextEdit; called only by the Tool Locator
`TEStartUp`	Initializes 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 display the blinking cursor and perform background tasks
`TEActivate`	Activates a specified TextEdit record
`TEDeactivate`	Deactivates a specified TextEdit record
`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 in the Clipboard
`TECopy`	Copies the current selection into the Clipboard
`TEPaste`	Pastes the contents of the Clipboard 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, or use `TENew` and allow TaskMaster to manage the control for you.
- Create a TextEdit control with the `NewControl2` or `TENew` 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. Note that the pseudocode presented in this discussion addresses only those issues of program logic that relate directly to TextEdit; much more logic is required to interact correctly with other tool sets or perform meaningful application-related work.

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 pseudocode describes the basic program logic for this technique.

```
Initialize all the tools including TextEdit.
Create a new window.
Call NewControl2 or TENew to allocate a new TextEdit control.
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 causes the cursor to blink. 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.
Shut down all the tools and exit.
```

Your application does not need to do anything if the user presses a key, presses the mouse button, or chooses a command from a menu. TaskMaster and the TextEdit control definition procedure handle all these standard events. The Window Manager disposes 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. To exert 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 pseudocode shows sample logic for this approach:

```
Initialize all the tools including TextEdit.
Create a new window.
Call NewControl2 or TENew to allocate a new TextEdit control.
while( quitFlag != TRUE )
      {
            Call TEIdle. This causes the cursor to blink 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
                                    Clipboard.
                        case CopyItem:
                              Call TECopy. This tells TextEdit to copy
                                    the current selection into the
                                    Clipboard.
```

```
                        case PasteItem:
                              Call TEPaste. This tells TextEdit to
                              replace the current selection with the
                              Clipboard.
                        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"
                        (such as Delete, Control-Y, arrow keys, and so
                        on).
            }
      }
Dispose of the window. This deallocates the TextEdit record and all
other controls in the window.
Shut down 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 pseudocode, but must also manage the TextEdit window itself. You must use the `TENew` call to create TextEdit records that are not controls. Because these TextEdit records are not inserted into the control list, 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 pseudocode presents sample logic for this approach:

```
Initialize all the tools including TextEdit.
Create a new window.
Call TENew to allocate a new TextEdit record that is not a control.
while( quitFlag != TRUE )
{
      Call TEIdle. This causes the cursor to blink 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 Clipboard.
                  case CopyItem:
                        Call TECopy. This tells TextEdit to copy the
                              current selection into the Clipboard.
                  case PasteItem:
                        Call TEPaste. This tells TextEdit to replace
                              the current selection with the Clipboard.
                  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" (such as
                Delete, Control-Y, arrow keys, and so on).
    }
}
Dispose of the window. This deallocates the TextEdit record and all
other controls in the window.
Shut down all the tools and exit.
```

When you use 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 noncontrol TextEdit records in the same window. ▲

All TextEdit tool calls require that you 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 NIL value for the `TERecord` handle. TextEdit will then access the currently active TextEdit control (the target TextEdit record).

▲ **Warning** Never pass a NIL `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, recall that 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 list 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 and places it in 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 is not 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

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 are 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` but 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 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, generic filter routines can provide three functions:

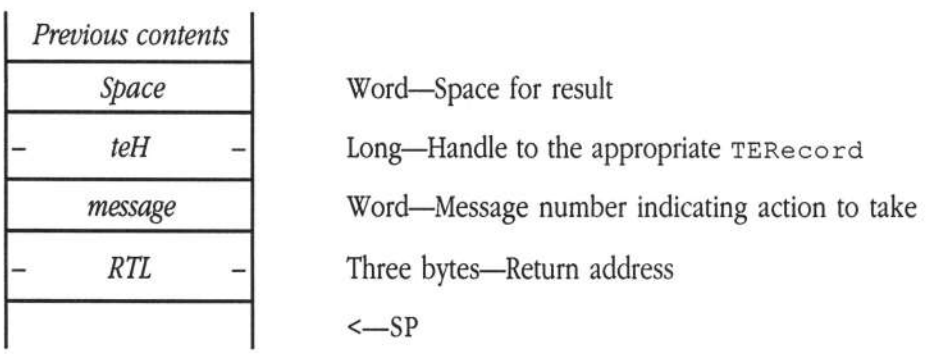
- erasing rectangles in the display port
- erasing rectangles in the off-screen TextEdit buffer
- updating the TextEdit record screen display

The `filterProc` field of the `TERecord` contains a pointer to the generic filter procedure. If this field has a non-NIL value, TextEdit calls 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.

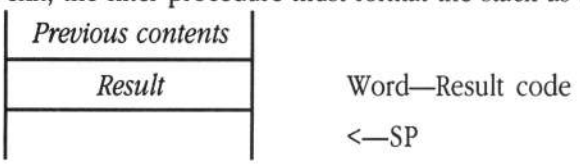
The routines in the filter procedure must adhere to the following rules:

- 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:



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 performs 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

`theFilterRect` The rectangle to erase, expressed in local coordinates for the port

Output `TERecord` field 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

`theFilterRect`	The rectangle to erase, expressed in local coordinates for the off-screen 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 `TERecord` field 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

`theFilterRect`	The rectangle that changed, expressed in local coordinates for the port

Output `TERecord` field 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 filter procedures* may support different keyboard mappings or macro facilities.

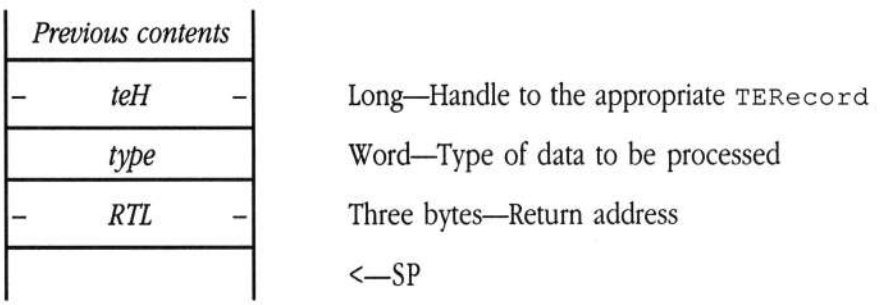
The `keyFilter` field of the `TERecord` can contain a pointer to a keystroke filter procedure. If `keyFilter` is NIL, TextEdit uses its standard keystroke routine. If `keyFilter` is non-NIL, TextEdit calls the routine pointed to by `keyFilter` to process all user keyboard input.

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.
- Keystroke filter procedures may move memory.

TextEdit invokes the keystroke filter procedure in full native mode by executing a `JSL`. Fields in the `KeyRecord` substructure in the `TERecord` contain information defining the data to be processed.

TextEdit loads additional control information onto the stack. On entry to the filter procedure, the stack is formatted as follows:



type The type of data to be processed.

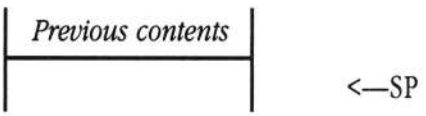
$0001	The character to be processed is stored in the `theChar` field of the `KeyRecord`
$0002	Reserved

The keystroke filter procedure is now free to perform whatever processing is appropriate. For example, it may change the input keystroke value 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).

One of the returned `KeyRecord` fields (`theOpCode`) specifies what action TextEdit is to take upon return from the filter procedure. This code in turn governs how TextEdit interprets the remainder of the returned `KeyRecord`. Here are the valid `theOpCode` values:

`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 the size of `theInputHandle` is 0, 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 cuts the current selection and places it in the Clipboard
`opCopy`	$000C	TextEdit copies the current selection to the Clipboard
`opPaste`	$000E	TextEdit replaces the current selection with the contents of the Clipboard
`opClear`	$0010	TextEdit clears the current selection

On exit, the filter procedure must format the stack as follows:



Input `KeyRecord` fields

`theChar`	The keystroke to process
`theModifiers`	Flags indicating the state of the modifier keys at the time the key was pressed; the keystroke is contained in `theChar` and in the location referred to by `theInputHandle`
`theInputHandle`	Handle to a copy of `theChar`

Output `KeyRecord` fields

`theChar`	Unchanged
`theModifiers`	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`	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.

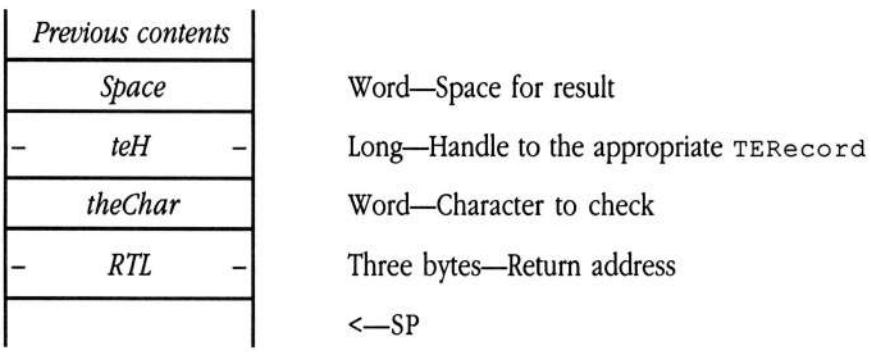
TextEdit determines whether to call a custom word wrap routine by examining the `wordWrapHook` field of the `TERecord`. If that field is NIL, TextEdit uses its standard word wrap routine. If that field is non-NIL, TextEdit calls the routine pointed to by that field whenever a word wrap decision is required. 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.

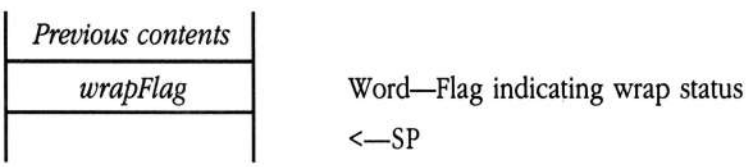
- The routine must preserve the direct-page and data bank registers, and must return in full native mode.
- Word wrap routines must not issue TextEdit tool calls, move memory, or cause memory to be moved.

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:



On exit, the filter procedure must format the stack as follows:



wrapFlag Wrap status of the current character.

$0000	Not a word wrap (TextEdit leaves the word on the current line)
$FFFF	Word wrap (TextEdit moves 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 (the 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.

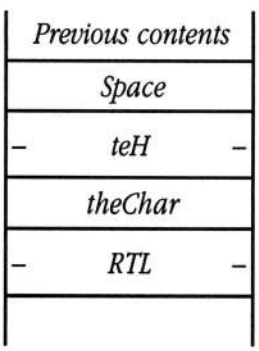
TextEdit determines whether to call a custom word wrap routine by examining the `wordBreakHook` field of the `TERecord`. If that field is NIL, TextEdit uses its standard word break routine. If that field is non-NIL, TextEdit calls the routine pointed to by that field whenever a word break decision is 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 constitutes a word break.

On entry to the procedure, the stack is formatted as follows:



Word—Space for result

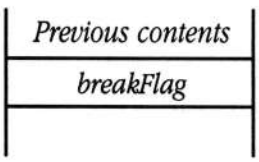
Long—Handle to the appropriate `TERecord`

Word—Character to check

Three bytes—Return address

<—SP

On exit, the filter procedure must format the stack as follows:



Word—Flag indicating break status

<—SP

breakFlag Break status of the current character.

$0000	Not a word break (TextEdit does not extend the selection)
$FFFF	Word break (TextEdit extends the selection to include the 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 Volume 1 of 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 of the 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," in Volume 1 of the *Toolbox Reference.*

Refer to Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* 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* 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, by contrast, are not relevant to most application programmers. However, if your program uses the low-level features of TextEdit or must for some other reason access TextEdit data structures directly, you should familiarize yourself with the information in that section.

In most cases, it is not necessary for your program to modify fields in these structures directly. However, if your program manipulates TextEdit structures, note that many of these data structures refer to or depend on 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.

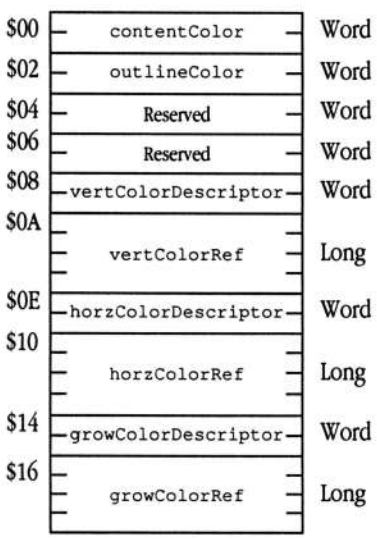
TEColorTable

Defines color attributes for a TextEdit record.

The `TEParamBlock` and `TERecord` structures 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," in Volume 2 of the *Toolbox Reference* for more information on QuickDraw II patterns and dithered colors. Figure 49-1 depicts the layout of the `TEColorTable` structure.

■ **Figure 49-1** `TEColorTable` layout



`contentColor` The color of the entire boundary rectangle (in essence, defining the background color of the text window).

`outlineColor` The color of the box that surrounds the text in the display window.

`vertColorDescriptor`
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 of the color table for the vertical scroll bar (resource type of `rCtlColorTbl`, $800D)

`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," in Volume 1 of the *Toolbox Reference.*

`horzColorDescriptor`
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 of the color table for the horizontal scroll bar (resource type of `rCtlColorTbl`, $800D)

`horzColorRef` Reference to the color table for the horizontal scroll bar. The *horzColorDescriptor* parameter indicates the type of reference stored here. This field must refer to a scroll bar color table, as defined in Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference.*

`growColorDescriptor`
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 of the color table for the size box (resource type of `rCtlColorTbl`, $800D)

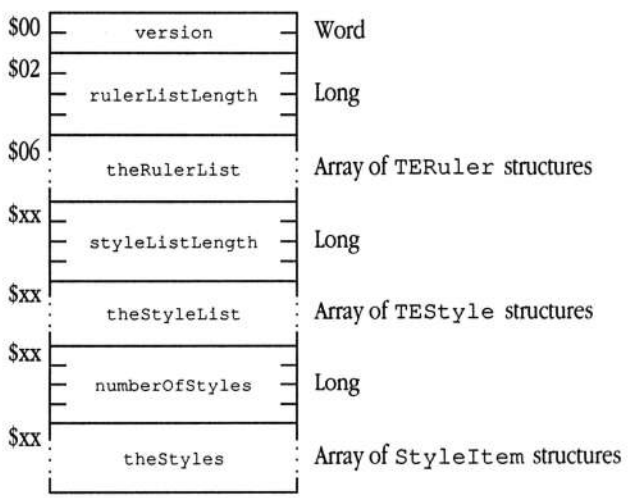
`growColorRef` Reference to the color table for the size box. The `growColorDescriptor` field indicates the type of reference stored here. This field must refer to a size box color table, as defined in Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference.*

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 shows the layout of the `TEFormat` structure.

■ **Figure 49-2** `TEFormat` layout



`version` Version number corresponding to the layout of this `TEFormat` structure. The number of this version of the structure is $0000.

`rulerListLength`
The length of `theRulerList` in bytes.

`theRulerList` Ruler data for the text record. The `TERuler` structure is embedded in 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 in the `TEFormat` structure at this location. Each `TEStyle` structure must define a unique style—there must be no duplicate style entries. 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 styles (stored in `theStyleList`) apply to which text in the TextEdit record.

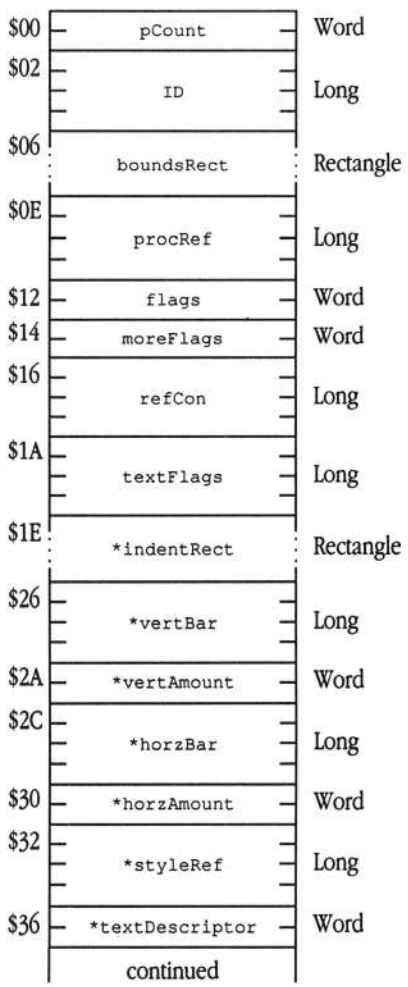
TEParamBlock

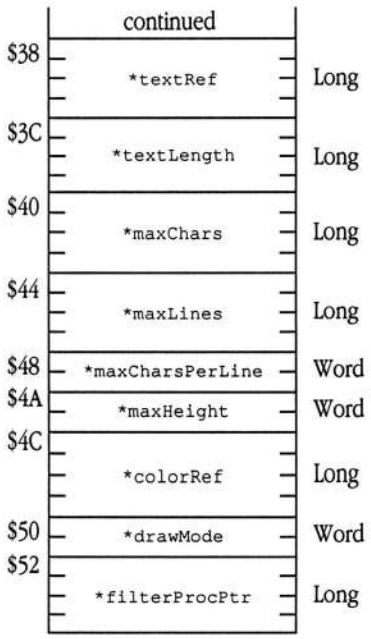
This structure contains the parameters necessary to create a new TextEdit record. In it your program defines many of the attributes of the record. The format of this structure corresponds directly to the format of the TextEdit control template accepted by the NewControl2 Control Manager call when creating 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 of TERecord fields.

In Figure 49-3, showing the layout of TEParamBlock, optional fields are marked with an asterisk(*).

■ **Figure 49-3** TEParamBlock layout





`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 identify a particular TextEdit record uniquely.

`boundsRect` Boundary rectangle for the TextEdit record. This rectangle contains the entire record, including its scroll bars and outline. If you set the lower-right corner of this rectangle to (0,0), TextEdit uses the lower-right corner of the window that contains the record as a default. Note that this rectangle must be large enough to display completely a single character in the largest allowed font.

`procRef` Type of control. Must be set to $85000000.

`flags` Control flags for the TextEdit record. Defined bits for `flags` are as follows:

Reserved	bits 15–8	Must be set to 0.
`fCtlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–0	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 a TextEdit record is created.
`fCtlCanBeTarget`	bit 14	Must be set to 1; TextEdit sets this bit to 0 if the `fDisableSelection` flag in `textFlags` is set to 1.
`fCtlWantEvents`	bit 13	Must be set to 1; TextEdit sets this bit to 0 if the `fDisableSelection` flag in `textFlags` is set to 1.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	If set to 1, TextEdit creates a size box in the lower-right corner of the window; whenever the window is resized, the edit text is resized and redrawn.
`fCtlIsMultiPart`	bit 10	Must be set to 1.
Reserved	bits 9–4	Must be set to 0.
Color table reference	bits 3–2	Defines type of reference in `colorRef`. 00 = Color table reference is by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Style reference	bits 1–0	Defines type of style reference in `styleRef`. 00 = Style reference is by pointer 01 = Style reference is by handle 10 = Style reference is by resource ID (resource type of `rStyleBlock`, $8012) 11 = Invalid value

△ **Important** Do not set `fCtlTellAboutSize` to 1 unless the TextEdit record also has a vertical scroll bar. This flag works only for TextEdit records that are controls. △

`textFlags` Specific TextEdit control flags. Valid values for `textFlags` are as follows:

`fNotControl`	bit 31	Indicates whether the TextEdit record to be created is a control. 0 = TextEdit record is a control 1 = TextEdit record is not a control
`fSingleFormat`	bit 30	Must be set to 1.
`fSingleStyle`	bit 29	Allows you to restrict the style options available to the user. 0 = Do not restrict the number of styles in the text 1 = Allow only one style in the text
`fNoWordWrap`	bit 28	Allows you to control TextEdit word wrap behavior. 0 = Perform word wrap to fit the ruler 1 = Do not use word wrap; break lines only on CR ($0D) characters
`fNoScroll`	bit 27	Controls user access to scrolling. 0 = Allow scrolling 1 = Do not allow either manual or automatic scrolling
`fReadOnly`	bit 26	Restricts the text in the window to read-only operations (copying from the window is still allowed). 0 = Editing permitted 1 = No editing allowed
`fSmartCutPaste`	bit 25	Controls TextEdit support for smart cut and paste. 0 = Do not use smart cut and paste 1 = Use smart cut and paste
`fTabSwitch`	bit 24	Defines behavior of the Tab key. 0 = Tab inserted in TextEdit document 1 = Tab to next control in the window
`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. 0 = Do not draw rectangle 1 = 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 this bit is set to 1, TextEdit automatically adjusts the right margin value for the ruler. 0 = Do not resize the ruler 1 = Resize the ruler
`fDisableSelection`	bit 20	Controls whether user can select text. 0 = User can select text 1 = User cannot select text
`fDrawInactiveSelection`	bit 19	Controls how inactive selected text is displayed. 0 = TextEdit does nothing special when displaying inactive selections 1 = TextEdit draws a box around inactive selections
Reserved	bits 18–0	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. 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, set this field to $FFFFFFFF.

`vertAmount` The number of pixels to scroll whenever the user presses the up or down arrow on the vertical scroll bar. 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. To use the default style and ruler information, set this field to NIL.

`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, set this field to NIL.

`textLength`
: If `textRef` is a pointer to the initial text, 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 limit the number of characters, then set this field to NIL.

`maxLines`
: Must be set to NIL.

`maxCharsPerLine`
: Must be set to NIL.

`maxHeight`
: Must be set to NIL.

`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`
: Text mode QuickDraw II uses to draw text. See Chapter 16, "QuickDraw II," in Volume 2 of the *Toolbox Reference* for details on valid text modes.

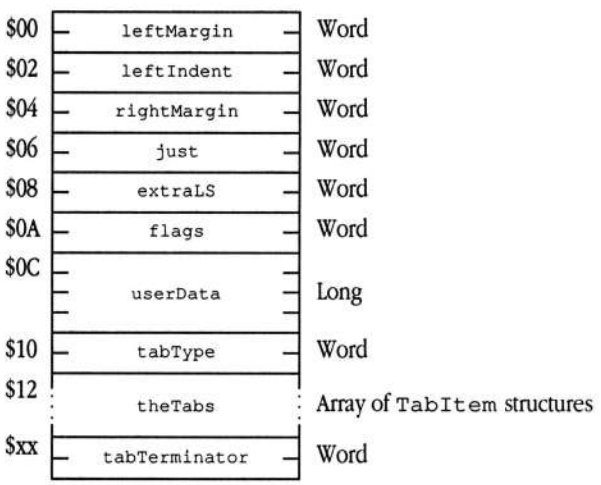
`filterProcPtr`
: Pointer to a filter routine for the control. For more information about TextEdit filter procedures, see "Generic Filter Procedure" earlier in this chapter. If you do not want to use a filter routine for the control, set this field to NIL.

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 shows the layout of the `TERuler` structure.

■ **Figure 49-4** `TERuler` layout



`leftMargin` 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` The number of pixels to indent from the left edge of the text rectangle for text lines that start paragraphs.

`rightMargin` Maximum line length, expressed as the number of pixels from the left edge of the text rectangle.

`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 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` Reserved

`userData` Application-specific data.

`tabType` The type of tab data, as 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, 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.

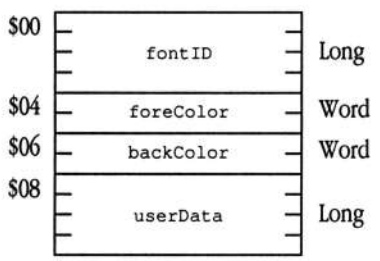
`tabTerminator` If `tabType` is set to 0, this field is omitted from the structure. If `tabType` is set to 1, then `theTabs` is omitted, and this field contains the number of pixels corresponding to the tab interval for the regular tabs. If `tabType` is set to 2, `tabTerminator` is set to $FFFF and marks the end of `theTabs` array.

TEStyle

This structure defines the font and color characteristics of a style of 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 shows the layout of the `TEStyle` structure.

■ **Figure 49-5** `TEStyle` layout



`fontID` Font Manager font ID record identifying the font of the text. See Chapter 8, "Font Manager," in Volume 1 of the *Toolbox Reference* for more information about font IDs.

`foreColor` Foreground color for the text. Note that all bits in TextEdit color words 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," in Volume 2 of the *Toolbox Reference* for more information on QuickDraw II patterns and dithered colors.

`backColor` Background color for the text.

`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 structures, you should be careful to maintain the correct relationships between fields that affect other TextEdit structures.

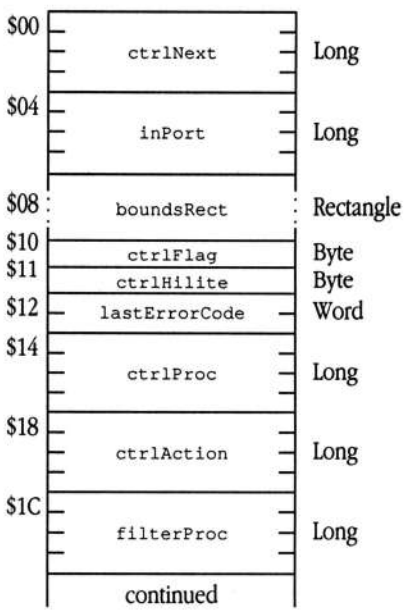
`TERecord`

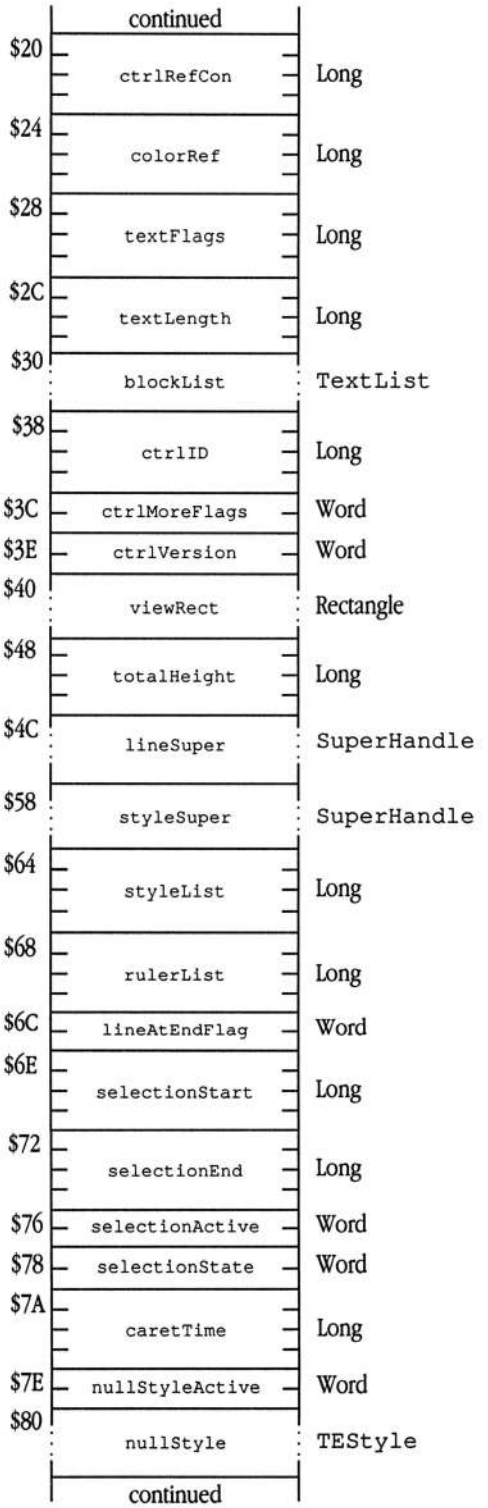
Figure 49-6 shows the main structure for a TextEdit record. The `TENew` tool call creates this structure partially based on the information specified in the `TEParamBlock` you supply to that call. In most cases, your program does not need to access fields in this structure directly. However, to use such advanced features as custom word wrap routines, your application will have to modify the `TERecord`.

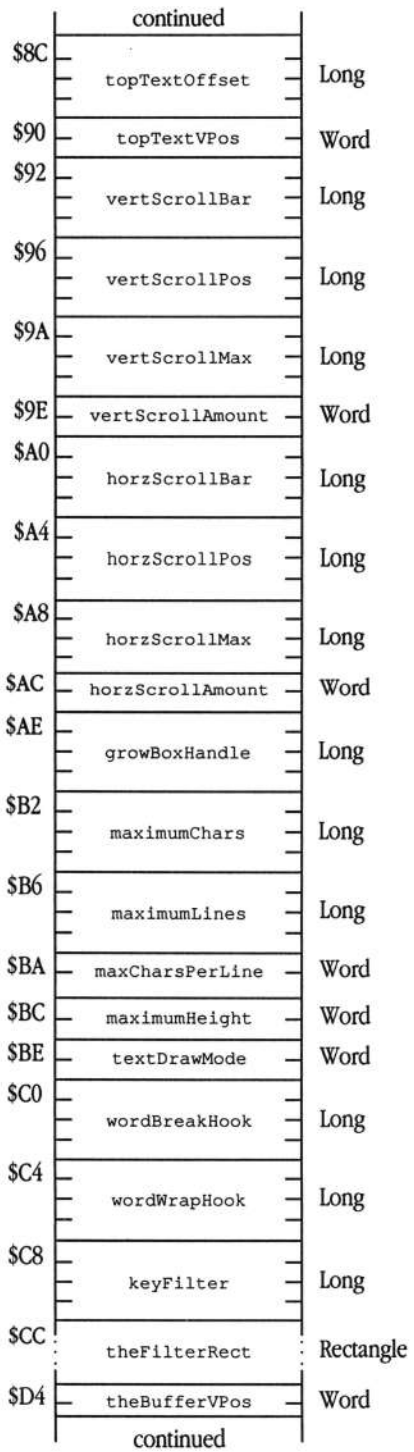
Note that this section describes only those `TERecord` fields that are currently defined and available to application programs. Your program should assume that there are more fields beyond those described here, and it should not try to move or copy a `TERecord` directly.

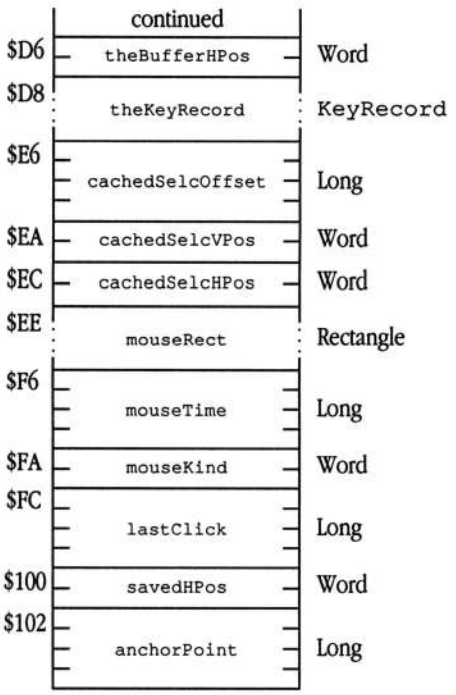
Most TextEdit tool calls require a handle to a `TERecord` as an entry parameter.

■ **Figure 49-6** `TERecord` layout




continued
$20
ctrlRefCon
Long
$24
colorRef
Long
$28
textFlags
Long
$2C
textLength
Long
$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
$76
selectionActive
Word
$78
selectionState
Word
$7A
caretTime
Long
$7E
nullStyleActive
Word
$80
nullStyle
TEStyle
continued


continued
$8C
topTextOffset
Long
$90
topTextVPos
Word
$92
vertScrollBar
Long
$96
vertScrollPos
Long
$9A
vertScrollMax
Long
$9E
vertScrollAmount
Word
$A0
horzScrollBar
Long
$A4
horzScrollPos
Long
$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
$D4
theBufferVPos
Word
continued



`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:

`fCtlInvis`	bit 7	0 = Visible, 1 = Invisible.
`fRecordDirty`	bit 6	Indicates whether text or style information for the record has changed (TextEdit sets this bit but never clears it—your application must set the bit to 0 whenever it saves the record). 0 = No text or style information has changed 1 = Text or style information has changed
Reserved	bits 5–0	Must be set to 0.

`ctrlHilite` Reserved

`lastErrorCode` The last error code generated by TextEdit. Note that this code may have been returned either from the TextEdit 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 NIL. See "Generic Filter Procedure" earlier in this chapter 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. 0 = TextEdit record is a control 1 = TextEdit record is not a control
`fSingleFormat`	bit 30	Must be set to 1.
`fSingleStyle`	bit 29	Allows you to restrict the style options available to the user. 0 = Do not restrict the number of styles in the text 1 = Allow only one style in the text
`fNoWordWrap`	bit 28	Allows you to control TextEdit word wrap behavior. 0 = Perform word wrap to fit the ruler 1 = Do not use word wrap; break lines only on CR ($0D) characters
`fNoScroll`	bit 27	Controls user access to scrolling. 0 = Allow scrolling 1 = Do not allow either manual or automatic scrolling
`fReadOnly`	bit 26	Restricts the text in the window to read-only operations (copying from the window is still allowed). 0 = Editing permitted 1 = No editing allowed

`fSmartCutPaste`	bit 25	Controls TextEdit support for smart cut and paste. 0 = Do not use smart cut and paste 1 = Use smart cut and paste
`fTabSwitch`	bit 24	Defines behavior of the Tab key. 0 = Tab inserted in TextEdit document 1 = Tab to next control in the window
`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. 0 = Do not draw rectangle 1 = 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 this bit is set to 1, TextEdit automatically adjusts the right margin value for the ruler. 0 = Do not resize the ruler 1 = Resize the ruler
`fDisableSelection`	bit 20	Controls whether user can select text. 0 = User can select text 1 = User cannot select text
`fDrawInactiveSelection`	bit 19	Controls how inactive selected text is displayed. 0 = TextEdit does nothing special when displaying inactive selections 1 = TextEdit draws a box around inactive selections
Reserved	bits 18–0	Must be set to 0.

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

`blockList` Cached link to 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; this bit must be set to 0 when a TextEdit record is created.
`fCtlCanBeTarget`	bit 14	Must be set to 1.
`fCtlWantEvents`	bit 13	Must be set to 1.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	If this bit is set to 1, TextEdit creates a size box in the lower-right corner of the window; whenever the window is resized, the edit text is resized and redrawn.
`fCtlIsMultiPart`	bit 10	Must be set to 1.
Reserved	bits 9–4	Must be set to 0.
Color table reference	bits 3–2	Defines type of reference in `colorRef`. 00 = Color table reference is by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Reserved	bits 1–0	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, in pixels, of the text in the TextEdit record.

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

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

`styleList` Handle to array of `TEStyle` structures, containing the unique styles for the record. The array is terminated with a long word 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 word 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 or equal to `selectionEnd`.

`selectionEnd` Ending text offset for the current selection. Must always be greater than or equal to `selectionStart`.

`selectionActive`
State information (active or inactive) about the current selection (defined by `selectionStart` and `selectionEnd`).

$0000	Active
$FFFF	Inactive

`selectionState`
State information about the current selection range.

$0000	Off screen
$FFFF	On screen

`caretTime` Blink interval for caret, expressed in system ticks.

`nullStyleActive`
State information about the null style 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.

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

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

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

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

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

`horzScrollBar` Currently not supported.

`horzScrollPos` Currently not supported.

`horzScrollMax` Currently not supported.

`horzScrollAmount`
Currently not supported.

`growBoxHandle` Handle of size box control record.

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

`maximumLines` Currently not supported.

`maxCharsPerLine`
Currently not supported.

`maximumHeight` Currently not supported.

`textDrawMode` QuickDraw II drawing mode for the text. See Chapter 16, "QuickDraw II," in Volume 2 of 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` 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" earlier in this chapter 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 commands. If the user clicks more than once in the region defined by this rectangle within the time period defined for multiclicks, then TextEdit interprets those clicks as multiclick sequences (double clicks or triple clicks). The user sets the time period with the Control Panel.

`mouseTime` System tick count when the user last released the mouse button.

`mouseKind` Type of last click.

0	Single click
1	Double click
2	Triple click

`lastClick` Location of last user 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 scroll arrow.

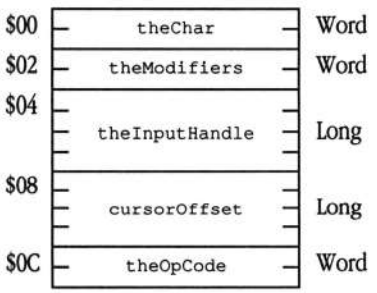
`anchorPoint` The character from which the user began to select text for the current selection. When TextEdit expands the current selection (as a result of user keyboard or mouse commands, or at the direction of a custom keystroke filter procedure), it always does so from the `anchorPoint`, not from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

KeyRecord

Defines the entry and exit parameter blocks 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 shows the layout of the `KeyRecord` structure.

■ **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," in Volume 1 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.

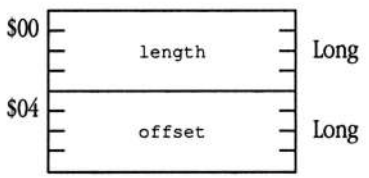
`theOpCode` On return from the filter routine, this field contains an operation code indicating what TextEdit is to do next and how it is to interpret the `KeyRecord`.

`opNormal`	$0000	TextEdit performs its standard processing on the character stored in the location referred to by `theInputHandle`, according to the value of `theModifiers`
`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 the size of `theInputHandle` is 0, 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 Clipboard and then clears the selection
`opCopy`	$0006	TextEdit copies the current selection to the Clipboard
`opPaste`	$0007	TextEdit inserts the contents of the Clipboard 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 series of characters. Taken consecutively, the array of StyleItem structures completely defines the styles for the entire record. Figure 49-8 shows the layout of the StyleItem structure.

■ **Figure 49-8** StyleItem layout

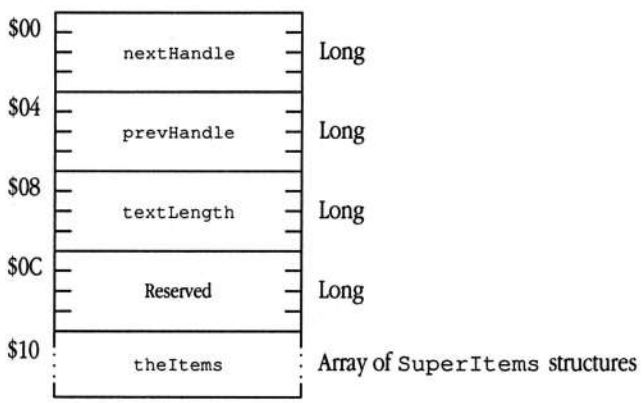


length	The total number of text characters that use this style. These characters begin where the previous StyleItem left off. A value of $FFFFFFFF indicates an unused entry.
offset	Offset, in bytes, into theStyleList array to the TEStyle record defining the characteristics of the style in question. The StyleList array is stored in the TEFormat record.

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 they 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 shows the layout of the SuperBlock structure.

■ **Figure 49-9** SuperBlock layout

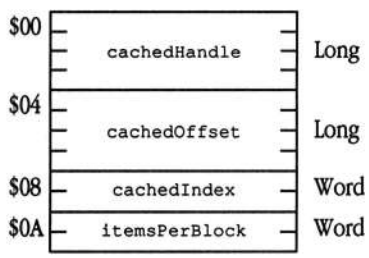


nextHandle	Handle to the next SuperBlock in this chain of blocks. A value of NIL indicates the end of the chain.
prevHandle	Handle to the previous SuperBlock in this chain of blocks. A value of NIL indicates the beginning of the chain.
textLength	The number of characters of text referred to by theItems.
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 to 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 shows the layout of the SuperHandle structure.

■ **Figure 49-10** SuperHandle layout

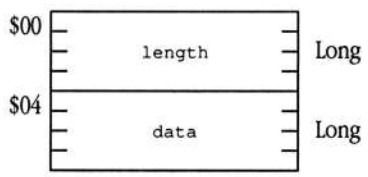


cachedHandle	Handle to the SuperBlock containing the current SuperItem.
cachedOffset	Byte offset to the current character within the text identified by the cached SuperItem.
cachedIndex	Byte offset to the start of the current SuperItem within the array of SuperItems stored in the SuperBlock referred to by cachedHandle.
itemsPerBlock	The number of SuperItems stored in each SuperBlock.

SuperItem

Defines an individual item within a `SuperBlock`. Figure 49-11 shows the layout of the `SuperItem` structure.

■ **Figure 49-11** `SuperItem` layout



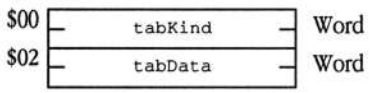
`length` The number of text characters in the TextEdit record that are affected by this `SuperItem`. A value of $FFFFFFFF indicates that this item is not currently used.

`data` 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 shows the layout of the `TabItem` structure.

■ **Figure 49-12** `TabItem` layout



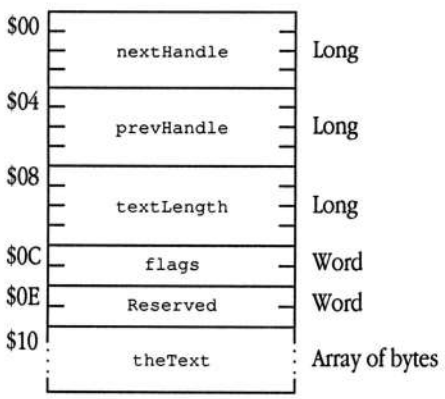
`tabKind` Must be set to $0000.

`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 that 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. The size of each of these `TextBlock` handles must be a multiple of 256 ($100), plus 16 ($10) (for example, 272 [$110], 528 [$210], and so on). Figure 49-13 shows the layout of the `TextBlock` structure.

■ **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`.

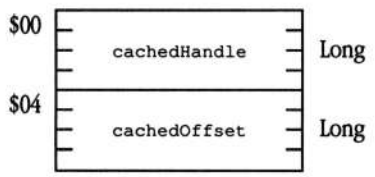
`flags` Reserved.

`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 shows the TextList structure.

■ **Figure 49-14** TextList layout



cachedHandle Handle to the TextBlock containing the text corresponding to the current location.

cachedOffset Byte offset from the start of the file to the start of the TextBlock described by this TextList entry.

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.

▲ **Warning** An application must never make this call. ▲

Parameters None

Errors None

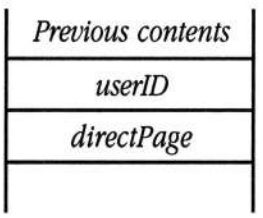
C Call must not be made by an application.

TEStartUp $0222

Starts up the TextEdit Tool Set and prepares TextEdit for application use by allocating memory and formatting direct-page space. Applications must issue this call before any other TextEdit tool calls. Before exiting, applications that issue the `TEStartUp` call must call `TEShutDown` to shut down TextEdit.

Parameters

Stack before call

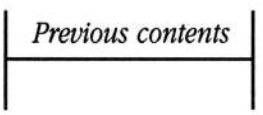


Word—Application's user ID (obtained at program start time)

Word—Address of one page of direct-page memory

<—SP

Stack after call



<—SP

Errors

$2201	`teAlreadyStarted`	TextEdit has already been started.
$220D	`teNeedsTools`	The Font Manager was not started.
Memory Manager errors		Returned unchanged.

C

```
extern pascal void TEStartUp(userID, directPage);

Word      userID, directPage;
```

TEShutDown $0322

Frees memory used by TextEdit, not including memory used by individual TextEdit records. It is the programmer's responsibility to issue the `TEKill` tool call at the end of processing for 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

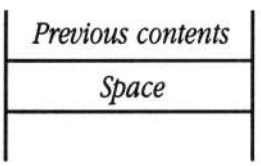
```
extern pascal void TEShutDown();
```

TEVersion $0422

Retrieves the TextEdit version number. This call returns valid information if TextEdit has been loaded; the tool set need not be active. The *versionInfo* result contains the information in the standard format defined in Appendix A, "Writing Your Own Tool Set," in Volume 2 of the *Toolbox Reference.*

Parameters

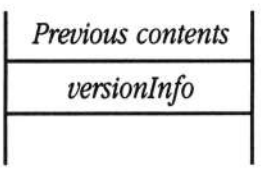
Stack before call



Word—Space for result

<—SP

Stack after call



Word—TextEdit version number

<—SP

Errors None

C

```
extern pascal Word TEVersion();
```

TEReset $0522

Resets TextEdit; called only when the system is reset.

▲ **Warning** An application must never make this call. ▲

Parameters None

Errors None

C Call must not be made by an application.

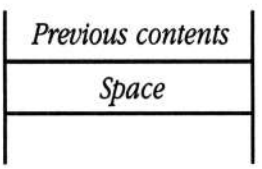
TEStatus $0622

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

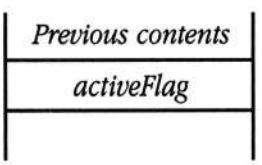
Stack before call



Word—Space for result

<—SP

Stack after call



Word—Boolean; TRUE if TextEdit is active

<—SP

Errors $0001 `toolNotFoundErr` TextEdit not loaded.

C

```
extern pascal Word TEStatus();
```

TextEdit tool calls

The following sections describe the TextEdit tool calls in order by call name.

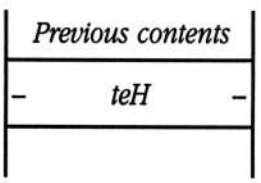
TEActivate $0F22

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

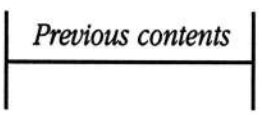
Stack before call



Long—Handle of `TERecord` in memory

<—SP

Stack after call



<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.

C

```
extern pascal void TEActivate(teH);

Long       teH;
```

teH

The TextEdit record for the operation. If your program specifies a NIL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

TEClear $1922

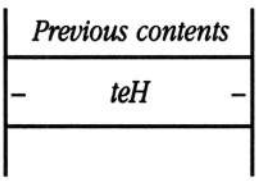
Clears the current selection in the active TextEdit record and redraws the screen. If there is no current selection, then this call does nothing and returns immediately. This call does not affect the Clipboard.

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

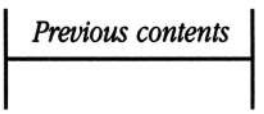
Stack before call



Long—Handle of `TERecord` in memory; NIL for active record

<—SP

Stack after call



<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.

C

```
extern pascal void TEClear(teH);

Long       teH;
```

teH

The TextEdit record for the operation. If your program specifies a NIL 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 pointer 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 causes the text to scroll in the proper direction if the user drags outside the view rectangle.

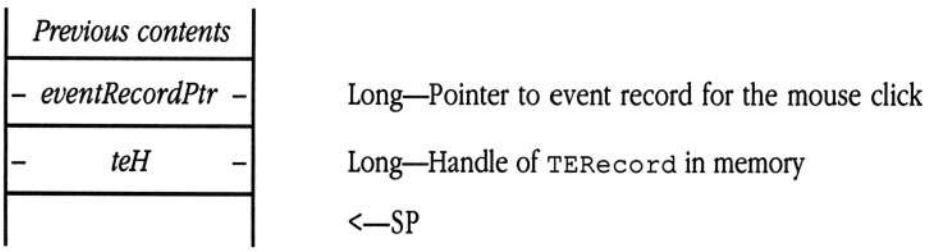
This call handles double and triple clicks as follows: double clicks select a word, and dragging thereafter lengthens or shortens the selection in word increments; triple clicks select a line, and dragging thereafter lengthens or shortens the selection in line increments.

If your program issues this call for a TextEdit record that is not currently active, TextEdit first makes that record active, and then proceeds to track the pointer.

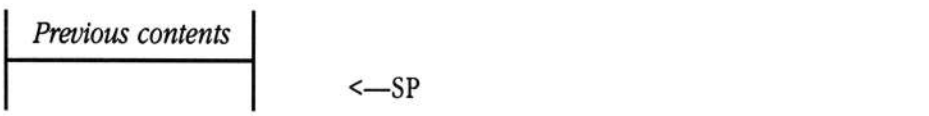
Your application need issue this call only if it is managing its own TextEdit records. If your program uses TextEdit controls with TaskMaster, it should not issue this call; TaskMaster manages the control automatically.

Parameters

Stack before call



Stack after call



Errors

$2202	teNotStarted	TextEdit has not been started.
$2203	teInvalidHandle	The *teH* parameter does not refer to a valid TERecord.
Memory Manager errors		Returned unchanged.

C

```
extern pascal void TEClick(eventRecordPtr, teH);

Pointer   eventRecordPtr;
Long      teH;
```

eventRecordPtr Pointer 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 Volume 1 of the *Toolbox Reference.*

teH The TextEdit record for the operation.

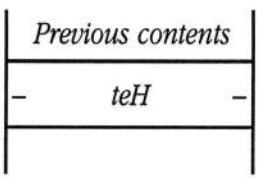
TECompactRecord $2822

Compresses 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. Although 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 NIL TextEdit record handle to this tool call.

Parameters

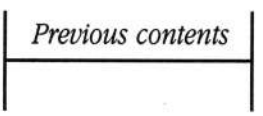
Stack before call



Long—Handle of `TERecord` to compact

<—SP

Stack after call



<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.
Memory Manager errors		Returned unchanged.

C

```
extern pascal void TECompactRecord(teH);

Long       teH;
```

teH The TextEdit record for the operation.

TECopy $1722

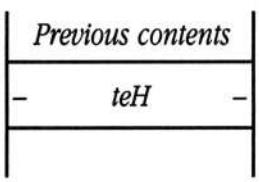
Copies the current selection from the active TextEdit record to the Clipboard, destroying the previous Clipboard contents. This call copies both the text and style information to the Clipboard. Note, however, that if there is no current selection, this call does nothing and does not affect the Clipboard.

This call does not automatically cause scrolling to the current selection.

Your application needs to 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

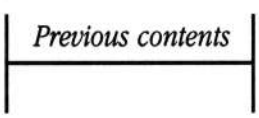
Stack before call



Long—Handle of `TERecord` in memory; NIL for active record

<—SP

Stack after call



<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.
Memory Manager errors		Returned unchanged.

C

```
extern pascal void TECopy(teH);

Long       teH;
```

teH

The TextEdit record for the operation. If your program specifies a NIL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

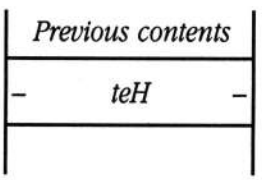
TECut $1622

Copies the current selection from the active TextEdit record to the Clipboard, destroying the previous Clipboard contents. `TECut` then scrolls to the beginning of the selection, deletes it, and redraws the screen. This call copies both the text and style information to the Clipboard. Note, however, that if there is no current selection, this call does nothing and does not affect the Clipboard.

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

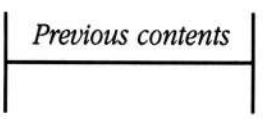
Stack before call



Long—Handle of `TERecord` in memory; NIL for active record

<—SP

Stack after call



<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.
Memory Manager errors		Returned unchanged.

C

```
extern pascal void TECut(teH);

Long       teH;
```

teH

The TextEdit record for the operation. If your program specifies a NIL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

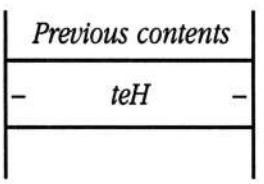
TEDeactivate $1022

Deactivates a TextEdit record. Your program specifies the `TERecord` for the record in question. `TEDeactivate` changes the highlighting of 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

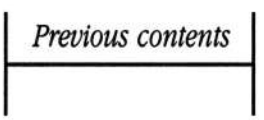
Stack before call



Long—Handle of `TERecord` in memory

<—SP

Stack after call



<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.

C

```
extern pascal void TEDeactivate(teH);

Long       teH;
```

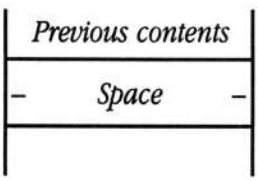
teH Specifies the TextEdit record for the operation.

TEGetDefProc $2222

Returns the address of the TextEdit control definition procedure. When the Control Manager starts up, the system issues this call to obtain the address of the TextEdit control definition procedure. This call is not intended for application use.

Parameters

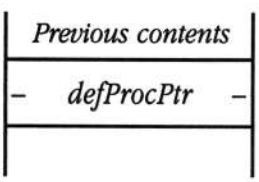
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to control definition procedure

<—SP

Errors None

C

```
extern pascal Pointer TEGetDefProc();
```

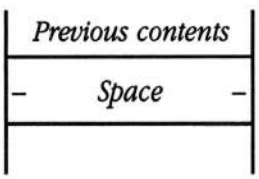
TEGetInternalProc $2622

Returns a pointer to the low-level procedure routine for TextEdit.

This call is reserved for future use by applications needing to access certain low-level TextEdit routines.

Parameters

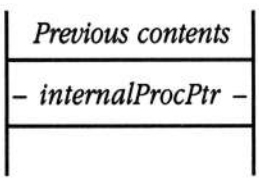
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to internal low-level procedure routine

<—SP

Errors None

C

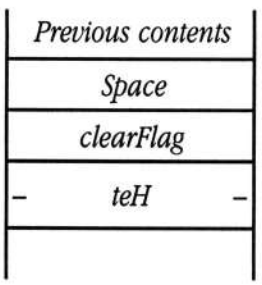
```
extern pascal Pointer TEGetInternalProc();
```

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



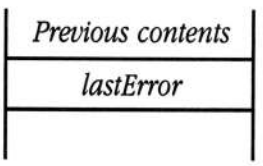
Word—Space for result

Word—Flag controlling disposition of last error field for record

Long—Handle of `TERecord` in memory; NIL for active record

<—SP

Stack after call



Word—Last error code generated for the record

<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.

C

```
extern pascal Word TEGetLastError(clearFlag, teH);

Word      clearFlag;
Long      teH;
```

clearFlag Flag controlling what TextEdit does with the last error field after servicing the call.

$0000	Leave the last error code intact
$FFFF	Clear the last error code to $0000

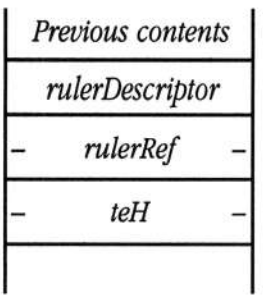
teH Specifies the TextEdit record for the operation.

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. The `TEGetRuler` call returns the `TERuler` record defining the ruler for the record in question.

Parameters

Stack before call



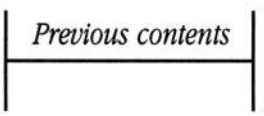
Word—Type of reference in *rulerRef*

Long—Reference to buffer to receive `TERuler` record

Long—Handle of `TERecord` in memory; NIL for active record

<—SP

Stack after call



<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.
Resource Manager errors		Returned unchanged.

C

```
extern pascal void TEGetRuler(rulerDescriptor,
          rulerRef, teH);

Word      rulerDescriptor;
Long      rulerRef, teH;
```

rulerDescriptor 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 that can be used to access a buffer to receive the `TERuler` structure (resource type of `rTERuler`, $8025)
`refIsNewHandle`	$0003	*rulerRef* contains a pointer to a 4-byte buffer to receive a handle to the `TERuler` structure; `TEGetRuler` allocates the new handle and returns it in the specified buffer

teH The TextEdit record for the operation. If your program specifies a NIL value, TextEdit works with the target TextEdit record. If there is no target record, TextEdit does nothing and returns immediately to your program.

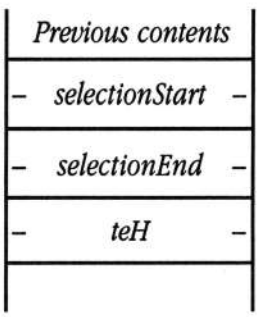
TEGetSelection $1C22

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 4-byte long values. If there is no current selection for the specified record, both the starting and ending offsets contain the current caret position.

Parameters

Stack before call



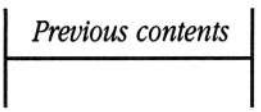
Long—Pointer to buffer to receive starting offset value

Long—Pointer to buffer to receive ending offset value

Long—Handle of `TERecord` in memory; NIL for active record

<—SP

Stack after call



<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.

C

```
extern pascal void TEGetSelection(selectionStart,
          selectionEnd, teH);

Pointer   selectionStart, selectionEnd;
Long      teH;
```

teH

The TextEdit record for the operation. If your program specifies a NIL 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

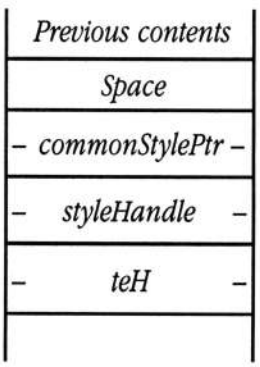
Returns all style information for the text in the current selection in a TextEdit record. Your program specifies the `TERecord` for the record in question and the addresses of buffers to receive the style data. `TEGetSelectionStyle` then loads the main output buffer with `TEStyle` structures describing 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 containing all style elements that 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 in which any text inserted at the current caret position will appear.

Parameters

Stack before call



Word—Space for result

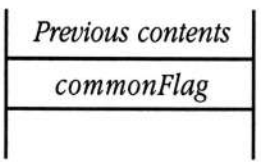
Long—Pointer to `TEStyle` buffer for common style record

Long—Handle to buffer for style information

Long—Handle of `TERecord` in memory; NIL for active record

<—SP

Stack after call



Word—Bit flag describing common style record contents

<—SP

Errors			
	$2202	`teNotStarted`	TextEdit has not been started.
	$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.

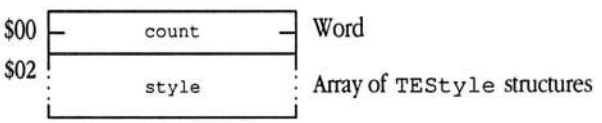
C

```
extern pascal Word
          TEGetSelectionStyle(commonStylePtr,
          styleHandle, teH);

Pointer   commonStylePtr;
Long      styleHandle, teH;
```

commonStylePtr Pointer to a buffer to receive a formatted `TEStyle` structure containing the style elements that are common to all text in the current selection. The *commonFlag* parameter 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 specify all the styles in the selection. If the buffer referenced by *styleHandle* cannot accommodate enough `TEStyle` structures, `TEGetSelectionStyle` automatically resizes the handle memory.

On return from `TEGetSelectionStyle`, the buffer referenced by *styleHandle* is formatted as follows:



`count` The number of `TEStyle` structures in the `styles` array.

`style` Array of `count` `TEStyle` structures.

teH The TextEdit record for the operation. If your program specifies a NIL 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 Flag indicating which portions of the common style record pointed to by *commonStylePtr* are relevant.

Reserved	bits 15–6	Will be set to 0.
`fUseFont`	bit 5	Indicates whether the font family defined by the `fontID` field of the common style record is valid. 0 = Font family not valid 1 = Font family valid
`fUseSize`	bit 4	Indicates whether the font size defined by the `fontID` field of the common style record is valid. 0 = Font size not valid 1 = Font size valid
`fUseForeColor`	bit 3	Indicates whether the `foreColor` field of the common style record is valid. 0 = Foreground color not valid 1 = Foreground color valid
`fUseBackColor`	bit 2	Indicates whether the `backColor` field of the common style record is valid. 0 = Background color not valid 1 = Background color valid
`fUseUserData`	bit 1	Indicates whether the `userData` field of the common style record is valid. 0 = User data not valid 1 = User data valid
`fUseAttributes`	bit 0	Indicates whether the attributes defined by the `fontID` field of the common style record are valid. 0 = Font attributes not valid 1 = Font attributes 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 of 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—Format of text returned at *bufferRef*
– *bufferRef* –	Long—Reference to the output text buffer
– *bufferLength* –	Long—Length of the buffer referred to by *bufferRef*
styleDescriptor	Word—Type of reference stored in *styleRef*
– *styleRef* –	Long—Reference to buffer for `TEFormat` structure defining style
– *teH* –	Long—Handle of `TERecord` in memory; NIL 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`	The *teH* parameter 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.
Resource Manager errors		Returned unchanged.

C

```
extern pascal Long TEGetText(bufferDescriptor,
          bufferRef, bufferLength, styleDescriptor,
          styleRef, teH);

Long      bufferRef, bufferLength, styleRef,
          teH;
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 15–5	Must be set to 0.
`refFormat`	bits 4–3	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 4-byte buffer to receive a handle to the output text; `TEGetText` allocates the handle; *bufferLength* is ignored

`dataFormat`	bits 2–0	Defines the format of the output text. 000 = Pascal string (resource type of `rPString`, $8006) 001 = C string (resource type of `rCString`, $801D) 010 = Class 1 GS/OS input string (resource type of `rC1InputString`, $8005) 011 = Class 1 GS/OS output string (resource type of `rC1OutputString`, $8023); application need not set the buffer size field 100 = Formatted for input to LineEdit `LETextBox2` tool call (resource type of `rTextForLETextBox2`, $800B)—see Chapter 10, "LineEdit Tool Set," in Volume 1 of the *Toolbox Reference* for details 101 = Unformatted text block (resource type of `rText`, $8016) 110 = Invalid value 111 = Invalid value

bufferLength 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 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 that can be used to access a buffer to receive the `TEFormat` structure (resource type of `rStyleBlock`, $8012)
`refIsNewHandle`	$0003	*styleRef* contains a pointer to a 4-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 NIL, `TEGetText` returns no style information and ignores *styleDescriptor*.

teH The TextEdit record for the operation. If your program specifies a NIL value, TextEdit works with the target TextEdit record. If there is no target record, then TextEdit does nothing and returns immediately to your program.

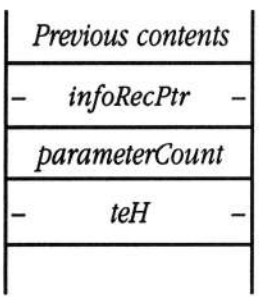
textLength The number of bytes of text in the record. Note that this value may exceed the number of bytes returned at *bufferRef*, if the referenced buffer is too small to receive all the text. In this case, `TEGetText` also returns a `teBufferOverflow` error code.

TEGetTextInfo $0D22

Returns an information record, of variable size, 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



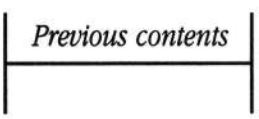
Long—Pointer to buffer for information record

Word—Number of fields to return

Long—Handle of `TERecord` in memory; NIL for active record

<—SP

Stack after call



<—SP

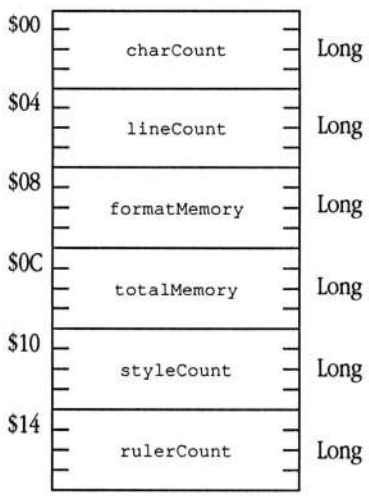
Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.
$2206	`teInvalidPCount`	Invalid parameter count value specified.

C

```
extern pascal void TEGetTextInfo(infoRecPtr,
          parameterCount, teH);

Pointer   infoRecPtr;
Long      teH;
Word      parameterCount;
```

infoRecPtr Pointer to a buffer to receive a partial or complete information record, depending on 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` The number of text characters in the record.

`lineCount` The number of lines in the record. A line is defined as all the text displayed on a single line of the screen, based on 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 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 begins with the `charCount` field and continues until the specified number of fields have been formatted.

teH

The TextEdit record for the operation. If your program specifies a NIL value, TextEdit works with the target TextEdit record. If there is no target record, TextEdit does nothing and returns immediately to your program.

TEIdle $0E22

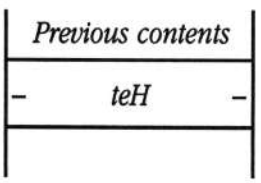
Provides processor time so that TextEdit can cause the cursor to blink and can perform other background tasks. Your program specifies the `TERecord` for the record. TextEdit then determines whether enough time has elapsed to require a cursor blink and, if so, causes the cursor to blink. In addition, 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

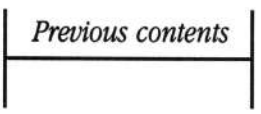
Stack before call



Long—Handle of `TERecord` in memory

<—SP

Stack after call



<—SP

Errors			
	$2202	`teNotStarted`	TextEdit has not been started.
	$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.

C

```
extern pascal void TEIdle(teH);

Long       teH;
```

teH The TextEdit record for the operation.

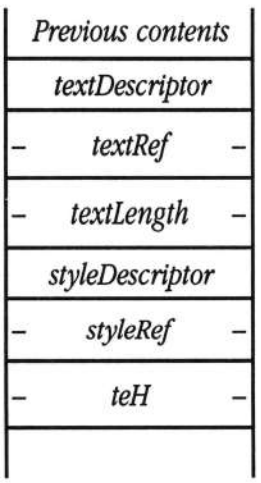
TEInsert $1A22

Inserts a block of text before the current selection in a TextEdit record and 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 Clipboard.

Parameters

Stack before call



Word—The format for text stored at *textRef*

Long—Reference to the input text buffer

Long—Length of the buffer referred to by *textRef*

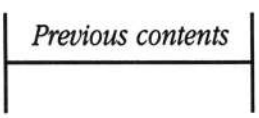
Word—The type of reference stored in *styleRef*

Long—Reference to `TEFormat` structure defining style for text

Long—Handle of `TERecord` in memory; NIL for active record

<—SP

Stack after call



<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.
$220C	`teInvalidTextBox2`	The `LETextBox2` format codes were inconsistent.
Memory Manager errors		Returned unchanged.

C

```
extern pascal void TEInsert(textDescriptor, textRef,
          textLength, styleDescriptor, styleRef,
          teH);

Long      textRef, textLength, styleRef, teH;
Word      textDescriptor, styleDescriptor;
```

textDescriptor The format of the text to be inserted, and the type of reference stored in *textRef*.

Reserved	bits 15–5	Must be set to 0.
`refFormat`	bits 4–3	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 value
`dataFormat`	bits 2–0	Defines the format of the text. 000 = Pascal string (resource type of `rPString`, $8006) 001 = C string (resource type of `rCString`, $801D) 010 = Class 1 GS/OS input string (resource type of `rC1InputString`, $8005) 011 = Class 1 GS/OS output string (resource type of `rC1OutputString`, $8023) 100 = Text formatted for input to LineEdit `LETextBox2` tool call (resource type of `rTextForLETextBox2`, $800B)—see Chapter 10, "LineEdit Tool Set," in Volume 1 of the *Toolbox Reference* for details; style data in the text overrides that specified by *styleRef* 101 = Unformatted text block (resource type of `rText`, $8016) 110 = Invalid value 111 = Invalid value

textLength Length of the buffer referenced by *textRef*. This field is valid only for reference types that do not contain length data (see *textDescriptor*). For other types of references, this field is ignored.

styleDescriptor 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 that can be used to access a buffer containing the `TEFormat` structure (resource type of `rStyleBlock`, $8012)

styleRef Reference to buffer containing style information, in `TEFormat` structure form. If this field is set to NIL, `TEInsert` uses the style of the first character in the current selection and ignores *styleDescriptor*.

teH The TextEdit record for the operation. If your program specifies a NIL value, TextEdit works with the target TextEdit record. If there is no target record, TextEdit does nothing and returns immediately to your program.

TEKey $1422

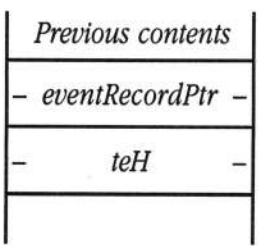
Processes a keystroke for a TextEdit record. Your program specifies the `TERecord` for the record and the event record for the keystroke; `TEKey` then processes the key. If the keystroke is a control key (one that requires special processing, as outlined in "Standard TextEdit Key Sequences" earlier in 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

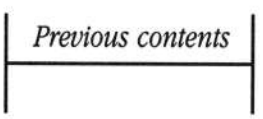


Long—Pointer to event record for the key

Long—Handle of `TERecord` in memory

<—SP

Stack after call



<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.
Memory Manager errors		Returned unchanged.

C

```
extern pascal void TEKey(eventRecordPtr, teH);

Pointer   eventRecordPtr;
Long      teH;
```

eventRecordPtr Pointer to the event record describing the keystroke. For information on the format and content of event records, see Chapter 7, "Event Manager," in Volume 1 of the *Toolbox Reference.* Note that TextEdit uses only the *message* and *modifiers* fields in the event record.

teH The TextEdit record for the operation.

TEKill $0A22

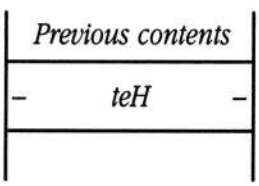
Deallocates a `TERecord` and all associated memory. Your program specifies the `TERecord` to be freed. `TEKill` then releases the record and any memory supporting it. `TEKill` does not erase or invalidate the screen, nor does it make another record the target if the target record is killed. Your program 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

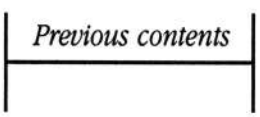
Stack before call



Long—Handle of `TERecord` in memory

<—SP

Stack after call



<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	*teH* does not refer to a valid `TERecord`.

C

```
extern pascal void TEKill(teH);

Long       teH;
```

teH The TextEdit record for the operation.

TENew $0922

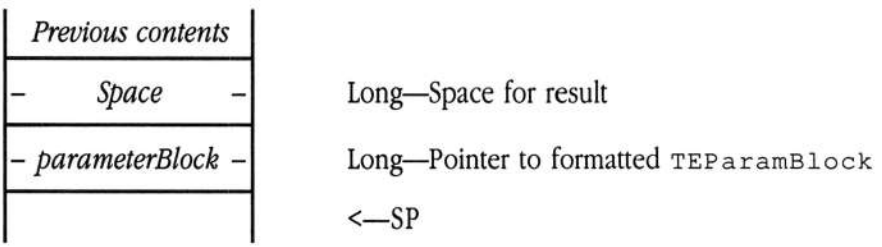
Allocates a new TextEdit record in the current port and returns the `TERecord` defining that record. Your program specifies the parameters for that record in a `TEParamBlock` structure (see "TextEdit Data Structures" earlier in this chapter for information on the format and content of the `TEParamBlock`). TextEdit then allocates and formats the `TERecord` for the record.

The boundary rectangle specified in the `TEParamBlock` must be large enough to completely enclose a single character in the largest allowable font for the record.

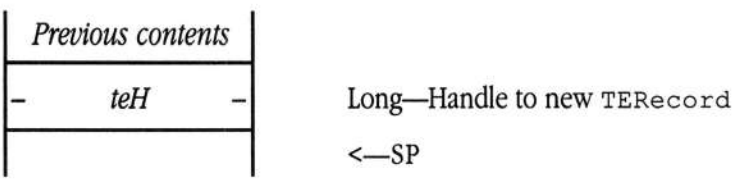
Your program should issue this call only if it is not using TextEdit controls. To create a TextEdit control, use the `NewControl2` Control Manager tool call (see Chapter 28, "Control Manager Update," in this book). Note that `NewControl2` may be used to create several controls at once.

Parameters

Stack before call



Stack after call



Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2204	`teInvalidDescriptor`	Invalid descriptor value specified.
$2205	`teInvalidFlag`	Specified flag word is invalid.
$2206	`teInvalidPCount`	Invalid parameter count value specified.
Memory Manager errors		Returned unchanged.

C

```
extern pascal Long TENew(parameterBlock);

Pointer   parameterBlock;
```

TEOffsetToPoint $2022

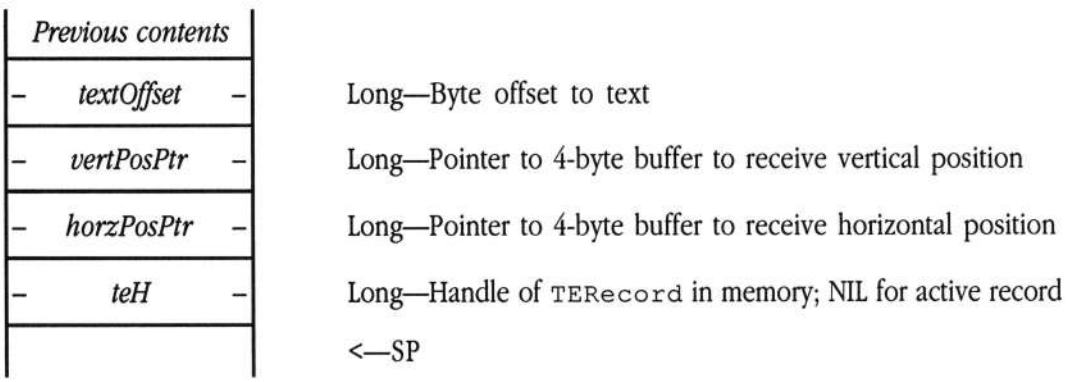
Converts a text byte offset into a pixel position expressed in the local coordinates of the GrafPort containing the TextEdit record. Your program specifies the byte offset to the character in question, the addresses of buffers to receive the pixel position information, and the `TERecord` for the record. `TEOffsetToPoint` then determines the pixel position of 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



Stack after call

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.

C

```
extern pascal void TEOffsetToPoint(textOffset,
          vertPosPtr, horzPosPtr, teH);

Long      textOffset, teH;
Pointer   vertPosPtr, horzPosPtr;
```

teH The TextEdit record for the operation. If your program specifies a NIL value, TextEdit works with the target TextEdit record. If there is no target record, 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," in Volume 1 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 (0 relative)
– *rectPtr* –	Long—Pointer to the destination rectangle in GrafPort
flags	Word—Control flags for the print operation
– *teH* –	Long—Handle of `TERecord` in memory; NIL for active record
	<—SP

Stack after call

Previous contents	
– *nextLine* –	Long—Next line number to print ($FFFFFFFF at end of file)
	<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.
$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, teH);

Pointer   grafPort, rectPtr;
Long      startingLine, teH;
Word      flags;
```

grafPort Pointer to a QuickDraw II GrafPort definition that 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 Volume 2 of the *Toolbox Reference.*

startingLine The first line to be printed. A line is defined as the text that is displayed on a single line of the screen, based on the current display options. TextEdit numbers lines starting from 0.

rectPtr Pointer 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* is filled. Refer to the description of the `PrOpenPage` tool call in Chapter 15, "Print Manager," in Volume 1 of the *Toolbox Reference* for more information on this frame rectangle.

flags Flags controlling 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 13–0	Must be set to 0.

teH The TextEdit record for the operation. If your program specifies a NIL value, TextEdit works with the target TextEdit record. If there is no target record, TextEdit does nothing and returns immediately to your program.

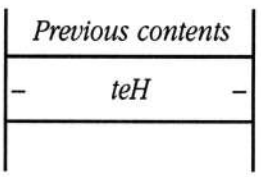
TEPaste $1822

Replaces the current selection with the contents of the Clipboard, 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 Clipboard into the record text. If the Clipboard 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

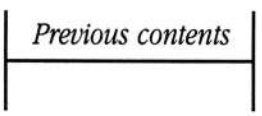
Stack before call



Long—Handle of `TERecord` in memory; NIL for active record

<—SP

Stack after call



<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.
Memory Manager errors		Returned unchanged.

C

```
extern pascal void TEPaste(teH);

Long       teH;
```

teH

The TextEdit record for the operation. If your program specifies a NIL value, TextEdit works with the target TextEdit record. If there is no target record, TextEdit does nothing and returns immediately to your program.

TEPointToOffset $2122

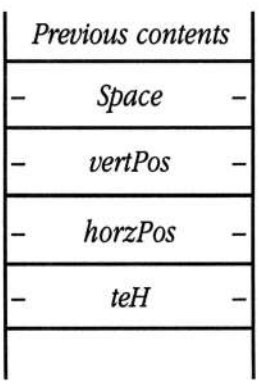
Converts a pixel position, expressed in the local coordinates of the GrafPort containing the TextEdit record, into a text byte offset to the text for the record. Your program specifies the pixel position in terms of its relative horizontal and vertical location in the GrafPort, but not as a QuickDraw 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



Long—Space for result

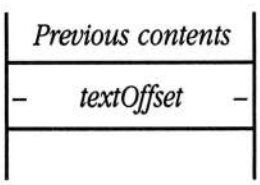
Long—Vertical position component

Long—Horizontal position component

Long—Handle of `TERecord` in memory; NIL for active record

<—SP

Stack after call



Long—Byte offset to text corresponding to pixel position

<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.

C

```
extern pascal Long TEPointToOffset(vertPos, horzPos,
          teH);

Long      vertPos, horzPos, teH;
```

teH The TextEdit record for the operation. If your program specifies a NIL value, TextEdit works with the target TextEdit record. If there is no target record, TextEdit does nothing and returns immediately to your program.

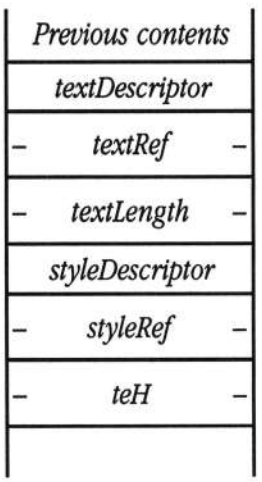
TEReplace $1B22

Replaces the current selection in a TextEdit record with a specified block of text and redraws the text screen. Your program specifies the text and style data to be replaced and the `TERecord` for the record. `TEReplace` then replaces the current selection with the new text and style data.

This call does not affect the Clipboard.

Parameters

Stack before call



Word—Format of text stored at *textRef*

Long—Reference to the input text buffer

Long—Length of the buffer referred to by *textRef*

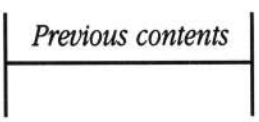
Word—Type of reference stored in *styleRef*

Long—Reference to `TEFormat` structure defining style for text

Long—Handle of `TERecord` in memory; NIL for active record

<—SP

Stack after call



<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.
$220C	`teInvalidTextBox2`	The `LETextBox2` format codes were inconsistent.
Memory Manager errors		Returned unchanged.

C

```
extern pascal void TEReplace(textDescriptor,
          textRef, textLength, styleDescriptor,
          styleRef, teH);

Long      textRef, textLength, styleRef, teH;
Word      textDescriptor, styleDescriptor;
```

textDescriptor The format of the text to be inserted and the type of reference stored in *textRef*.

Reserved	bits 15–5	Must be set to 0.
`refFormat`	bits 4–3	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 value
`dataFormat`	bits 2–0	Defines the format of the text. 000 = Pascal string (resource type of `rPString`, $8006) 001 = C string (resource type of `rCString`, $801D) 010 = Class 1 GS/OS input string (resource type of `rC1InputString`, $8005) 011 = Class 1 GS/OS output string (resource type of `rC1OutputString`, $8023) 100 = Text formatted for input to LineEdit `LETextBox2` tool call (resource type of `rTextForLETextBox2`, $800B)—see Chapter 10, "LineEdit Tool Set," in Volume 1 of the *Toolbox Reference* for details; style data in the text overrides that specified by *styleRef* 101 = Unformatted text block (resource type of `rText`, $8016) 110 = Invalid value 111 = Invalid value

textLength The length of the buffer referenced by *textRef*. This field is valid only for reference types that do not contain length data (see *textDescriptor*). For other types of references, this field is ignored.

styleDescriptor 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 that can be used to access a buffer containing the `TEFormat` structure (resource type of `rStyleBlock`, $8012)

styleRef Reference to buffer containing style information, in `TEFormat` structure form. If this field is set to NIL, `TEReplace` uses the first defined style in the current selection for the record and ignores *styleDescriptor*.

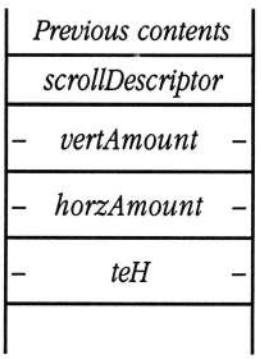
teH The TextEdit record for the operation. If your program specifies a NIL value, TextEdit works with the target TextEdit record. If there is no target record, TextEdit does nothing and returns immediately to your program.

TEScroll $2522

Causes the text in a TextEdit record to scroll. Your program specifies control information for the scroll operation and the `TERecord` for the record. `TEScroll` then updates the current position for the record accordingly.

Parameters

Stack before call



Word—Control information for the scroll operation

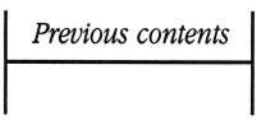
Long—Vertical amount to scroll (this is a signed value)

Long—Horizontal amount to scroll (must be set to 0)

Long—Handle of `TERecord` in memory; NIL for active record

<—SP

Stack after call



<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.

C

```
extern pascal void TEScroll(scrollDescriptor,
          vertAmount, horzAmount, teH);

Word      scrollDescriptor;
Long      vertAmount, horzAmount, teH;
```

scrollDescriptor 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 byte 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 byte 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 defining how far the top of the TextEdit window should scroll, 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.

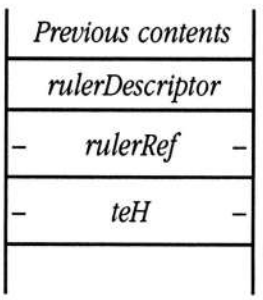
teH The TextEdit record for the operation. If your program specifies a NIL value, TextEdit works with the target TextEdit record. If there is no target record, TextEdit does nothing and returns immediately to your program.

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



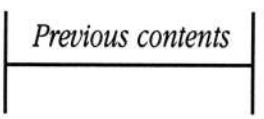
Word—Type of reference in *rulerRef*

Long—Reference to buffer containing new `TERuler` structure

Long—Handle of `TERecord` in memory; NIL for active record

<—SP

Stack after call



<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.

C

```
extern pascal void TESetRuler(rulerDescriptor,
          rulerRef, teH);

Word      rulerDescriptor;
Long      rulerRef, teH;
```

rulerDescriptor 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 that can be used to access a buffer containing the `TERuler` structure (resource type of `rTERuler`, $8025)

teH The TextEdit record for the operation. If your program specifies a NIL value, TextEdit works with the target TextEdit record. If there is no target record, TextEdit does nothing and returns immediately to your program.

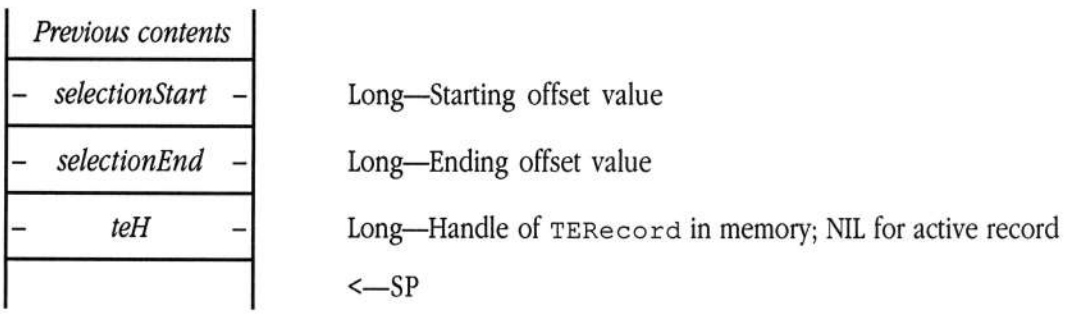
TESetSelection $1D22

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



Stack after call

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.

C

```
extern pascal void TESetSelection(selectionStart,
          selectionEnd, teH);

Pointer   selectionStart, selectionEnd;
Long      teH;
```

teH

The TextEdit record for the operation. If your program specifies a NIL value, TextEdit works with the target TextEdit record. If there is no target record, 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—Format of text stored at *textRef*
– *textRef* –	Long—Reference to the input text
– *textLength* –	Long—Length of the buffer referred to by *textRef*
styleDescriptor	Word—Type of reference stored in *styleRef*
– *styleRef* –	Long—Reference to `TEFormat` structure defining style
– *teH* –	Long—Handle of `TERecord` in memory; NIL for active record
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter 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, teH);

Long      textRef, textLength, styleRef, teH;
Word      textDescriptor, styleDescriptor;
```

textDescriptor The format of the new text for the record, and the type of reference stored in *textRef*.

Reserved	bits 15–5	Must be set to 0.
`refFormat`	bits 4–3	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 2–0	Defines the format of the text. 000 = Pascal string (resource type of `rPString`, $8006) 001 = C string (resource type of `rCString`, $801D) 010 = Class 1 GS/OS input string (resource type of `rC1InputString`, $8005) 011 = Class 1 GS/OS output string (resource type of `rC1OutputString`, $8023) 100 = Text formatted for input to LineEdit `LETextBox2` tool call (resource type of `rTextForLETextBox2`, $800B)—see Chapter 10, "LineEdit Tool Set," in Volume 1 of the *Toolbox Reference* for details; style data in the text overrides that specified by *styleRef* 101 = Unformatted text block (resource type of `rText`, $8016) 110 = Invalid value 111 = Invalid value

textLength Length of the text referenced by *textRef*. This field is valid only for reference types that do not contain length data (see *textDescriptor*). For other types of references, this field is ignored.

styleDescriptor 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 that can be used to access the `TEFormat` structure (resource type of `rStyleBlock`, $8012)

styleRef Reference to style information for the new text, in `TEFormat` structure form. If this field is set to NIL, `TESetText` uses the first style encountered in the existing text for the record.

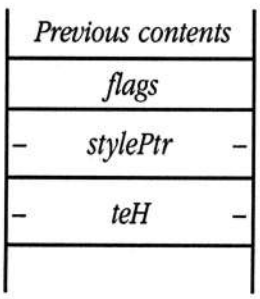
teH The TextEdit record for the operation. If your program specifies a NIL value, TextEdit works with the target TextEdit record. If there is no target record, TextEdit does nothing and returns immediately to your program.

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



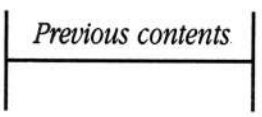
Word—Control flag for applying style data from `TEStyle`

Long—Pointer to `TEStyle` structure

Long—Handle of `TERecord` in memory; NIL for active record

<—SP

Stack after call



<—SP

Errors

$2202	`teNotStarted`	TextEdit has not been started.
$2203	`teInvalidHandle`	The *teH* parameter does not refer to a valid `TERecord`.
$2205	`teInvalidFlag`	Specified flag word is invalid.

C

```
extern pascal void TEStyleChange(flags, stylePtr,
          teH);

Word      flags;
Pointer   stylePtr;
Long      teH;
```

flags Flags indicating which portions of the `TEStyle` structure pointed to by *stylePtr* are relevant.

Reserved	bits 15–7	Must be set to 0.
`fReplaceFont`	bit 6	Controls use of font family defined by `fontID` field of `TEStyle` structure. 0 = Do not change font family 1 = Replace the font family for all styles in the current selection
`fReplaceSize`	bit 5	Controls use of font size defined by `fontID` field of `TEStyle` structure. 0 = Do not change font size 1 = Replace the font size for all styles in the current selection
`fReplaceForeColor`	bit 4	Controls use of `foreColor` field of `TEStyle` structure. 0 = Do not change the foreground color 1 = Replace the foreground color for all styles in the current selection
`fReplaceBackColor`	bit 3	Controls use of `backColor` field of `TEStyle` structure. 0 = Do not change the background color 1 = Replace the background color for all styles in the current selection
`fReplaceUserData`	bit 2	Controls the use of the `userData` field of the `TEStyle` structure. 0 = Do not change `userData` field 1 = Replace the `userData` field for all styles in the current selection with that in the supplied `TEStyle` structure
`fReplaceAttributes`	bit 1	Controls use of font attributes defined by the `fontID` field of `TEStyle` structure. 0 = Do not change font attributes 1 = Replace the font attributes for all styles in the current selection

`fSwitchAttributes`

bit 0 — Controls attribute switching.
0 = Perform no attribute switching
1 = If the entire selection contains the font attributes specified by the `TEStyle` structure `fontID` field, 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)

- *Note:* The `fReplaceAttributes` and `fSwitchAttributes` flags are mutually exclusive. If both flags are set to 1, `TEStyleChange` returns a `teInvalidFlag` error code.

stylePtr — Pointer 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.

teH — The TextEdit record for the operation. If your program specifies a NIL value, TextEdit works with the target TextEdit record. If there is no target record, TextEdit does nothing and returns immediately to your program.

TEUpdate $1222

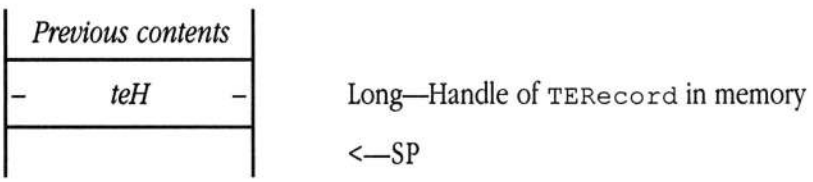
Redraws the screen for a TextEdit record. Your program specifies the `TERecord` for the record. `TEUpdate` then redraws the text for the record. Only that portion of the screen that must be redrawn is affected.

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.

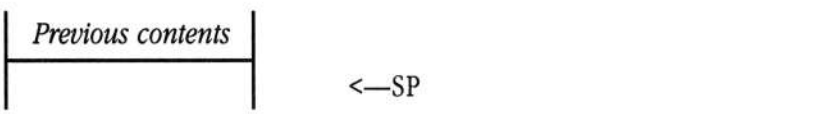
If your program uses TextEdit controls, use the ControlManager `DrawControls` tool call rather than `TEUpdate`.

Parameters

Stack before call



Stack after call



Errors	$2202	`teNotStarted`	TextEdit has not been started.
	$2203	`teInvalidHandle`	The *teH* arameter does not refer to a valid `TERecord`.

C

```
extern pascal void TEUpdate(teH);

Long       teH;
```

teH The TextEdit record for the operation.

TextEdit summary

Tables 49-1, 49-2, and 49-3 summarize the constants, data structures, and error codes (respectively) used by TextEdit.

■ **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` `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

[continued]

■ **Table 49-1** TextEdit constants [continued]

Name	Value	Description
`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
`fCtlWantEvents`	$2000	Must be set to 1
`fCtlProcRefNotPtr`	$1000	Must be set to 1
`fCtlTellAboutSize`	$0800	Record should have a size box
`fCtlIsMultiPart`	$0400	Must be set to 1
`Color table reference`	$000C	Indicates type of reference in `colorRef`
`Style reference`	$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 boundary 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 used for inside of boundary rectangle
`outlineColor`	$0002	Word	Color used for outline drawn around text
`vertColorDescriptor`	$0008	Word	Type of reference in `vertColorRef`
`vertColorRef`	$000A	Long	Reference to color table for vertical scroll bar
`horzColorDescriptor`	$000E	Word	Type of reference in `horzColorRef`
`horzColorRef`	$0010	Long	Reference to color table for horizontal scroll bar
`growColorDescriptor`	$0014	Word	Type of reference in `growColorRef`
`growColorRef`	$0016	Long	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 of format structure—value must be $0000
`rulerListLength`	$0002	Long	Size in bytes of `theRulerList` array
`theRulerList`	$0006	`TERuler`	Array of `TERuler` structures
`styleListLength`		Long	Size in bytes of `theStyleList` array
`theStyleList`		`TEStyle`	Array of `TEStyle` structures
`numberOfStyles`		Long	Number of entries in `theStyles` array
`theStyles`		`StyleItem`	Array of `StyleItem` structures

[continued]

■ **Table 49-2** TextEdit data structures [continued]

Name	Offset/Value	Type	Description
`TEParamBlock` (parameter block for creating TextEdit structures)			
`pCount`	$0000	Word	Number of parameters to follow—values range from 7 through 23
`ID`	$0002	Long	Application-assigned ID for record
`boundsRect`	$0006	Rect	Boundary rectangle for entire TextEdit record, including scroll bars and outlines
`procRef`	$000E	Long	Must be set to $85000000
`flags`	$0012	Word	Control flags
`moreFlags`	$0014	Word	More control flags
`refCon`	$0016	Long	Reserved for application use
`textFlags`	$001A	Long	TextEdit control flags
`indentRect`	$001E	Rect	Number of pixels to indent the text from each edge of the boundary rectangle
`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 NIL
`horzAmount`	$0030	Word	Reserved—must be set to $0000
`styleRef`	$0032	Long	Reference to initial style information for record
`textDescriptor`	$0036	Word	Defines format of `textRef`
`textRef`	$0038	Long	Reference to initial text for record
`textLength`	$003C	Long	Length of text referred to by `textRef`
`maxChars`	$0040	Long	Maximum number of characters allowed in record
`maxLines`	$0044	Long	Must be set to NIL
`maxCharsPerLine`	$0048	Word	Must be set to NIL
`maxHeight`	$004A	Word	Must be set to NIL
`colorRef`	$004C	Long	Reference to the `TEColorTable` for the record
`drawMode`	$0050	Word	QuickDraw II text mode
`filterProcPtr`	$0052	Pointer	Pointer to filter routine for the record

[continued]

■ **Table 49-2** TextEdit data structures [continued]

Name	Offset/Value	Type	Description
`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	Boundary 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	Long	Always set to $85000000
`ctrlAction`	$0018	Long	Reserved
`filterProc`	$001C	Pointer	Pointer to filter procedure for the record
`ctrlRefCon`	$0020	Long	Reserved for application
`colorRef`	$0024	Long	Reference to `TEColorTable` for record
`textFlags`	$0028	Long	The `textFlags` field from the `TEParamBlock` used to create the record
`textLength`	$002C	Long	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	Boundary rectangle for text on screen
`totalHeight`	$0048	Long	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	Long	Starting text offset for current selection
`selectionEnd`	$0072	Long	Ending text offset for current selection

[continued]

■ **Table 49-2** TextEdit data structures [continued]

Name	Offset/Value	Type	Description
selectionActive	$0076	Word	Indicates whether selection is active
selectionState	$0078	Word	Indicates whether selection is on screen
caretTime	$007A	Long	Tick count for insertion point blink
nullStyleActive	$007E	Word	Indicates whether null style is to be used
nullStyle	$0080	TEStyle	Style definition for null style
topTextOffset	$008C	Long	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	Long	Current vertical scroll position
vertScrollMax	$009A	Long	Maximum allowable vertical scroll from top of text
vertScrollAmount	$009E	Word	Number of pixels to scroll on each vertical arrow click
horzScrollBar	$00A0	Handle	Not supported
horzScrollPos	$00A4	Long	Not supported
horzScrollMax	$00A8	Long	Not supported
horzScrollAmount	$00AC	Word	Not supported
growBoxHandle	$00AE	Handle	Handle to the size box control
maximumChars	$00B2	Long	Maximum number of characters allowed in the text
maximumLines	$00B6	Long	Not supported
maxCharsPerLine	$00BA	Word	Not supported
maximumHeight	$00BC	Word	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

[continued]

■ **Table 49-2** TextEdit data structures [continued]

Name	Offset/Value	Type	Description
`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	Long	Text offset for cached insertion point position
`cachedSelcVPos`	$00EA	Word	Vertical component of cached insertion point position
`cachedSelcHPos`	$00EC	Word	Horizontal component of cached insertion point position
`mouseRect`	$00EE	Rect	Boundary rectangle for mouse events
`mouseTime`	$00F6	Long	Tick count value when mouse button was last released
`mouseKind`	$00FA	Word	Type of last click
`lastClick`	$00FC	Point	Location of last click
`savedHPos`	$0100	Word	Saved horizontal position for up and down scroll 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.

[continued]

■ **Table 49-2** TextEdit data structures [continued]

Name	Offset/Value	Type	Description
`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	Long	Reserved for application use
`tabType`	$0010	Word	Type of tabs used
`theTabs`	$0012	`TabItem`	Array of `TabItems`, one for each absolute tab stop
`tabTerminator`	$xxxx	Word	Either the spacing for standard tabs or a flag terminating `theTabs` array
`TEStyle` (style description structure)			
`fontID`	$0000	Long	Font ID for text using this style
`foreColor`	$0004	Word	Foreground color for the style
`backColor`	$0006	Word	Background color for the style
`userData`	$0008	Long	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," in Volume 1 of the *Toolbox Reference* for information on key modifiers)
`theInputHandle`	$0004	Handle	Handle to character to insert
`cursorOffset`	$0008	Long	New cursor location
`theOpCode`	$000C	Word	Operation code for key filter routine

[continued]

■ **Table 49-2** TextEdit data structures [continued]

Name	Offset/Value	Type	Description
`StyleItem` (style reference structure)			
`length`	$0000	Long	Number of text characters using the style
`offset`	$0004	Long	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	Long	Number of bytes of text for this `SuperBlock`
	$000C	Long	Reserved
`theItems`	$0010	`SuperItem`	Array of `SuperItems` for this block
`SuperHandle`			
`cachedHandle`	$0000	Handle	Handle to the current `SuperBlock`
`cachedOffset`	$0004	Long	Text offset of 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	Long	Number of bytes of text for this `SuperItem`
`data`	$0004	Long	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

[continued]

■ **Table 49-2** TextEdit data structures [continued]

Name	Offset/Value	Type	Description
`TextBlock`			
`nextHandle`	$0000	Handle	Handle to next `TextBlock` in list
`prevHandle`	$0004	Handle	Handle to previous `TextBlock` in list
`textLength`	$0008	Long	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	Long	Text offset of 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 *teH* 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	Reserved	Reserved.
$220B	`teInvalidParameter`	A passed parameter was invalid.
$220C	`teInvalidTextBox2`	The `LETextBox2` format codes were inconsistent.
$220D	`teNeedsTools`	The Font Manager was not started.

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 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 routes 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, to maintain compatibility with existing code, all get device calls still return slot numbers in the range from 1 through 7.

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.*

Error correction

Contrary to the call descriptions in Chapter 24 of the *Toolbox Reference,* both the `MessageCenter` and `SaveTextState` tool calls can return Memory Manager errors.

Clarification

Applications that explicitly start up Apple IIGS tool sets should start the Desk Manager last.

New features of 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 loads tools from disk only if it cannot find a tool in ROM with a version number as high as that of 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 because 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. It starts the Resource Manager.
2. It opens the resource fork for the current application in read-only mode.
3. It obtains memory for the application's direct page.
4. It starts the tools specified in the input `StartStop` record; then it updates the `StartStop` record as appropriate.
5. It returns the `StartStop` record reference to the calling program.

Your application must pass this returned `StartStop` record reference to `ShutDownTools` at tool shutdown time.

The `StartUpTools` call sets some tool set default values for you. If these values are not appropriate for your application, you should change them by issuing the appropriate tool calls after `StartUpTools` has returned:

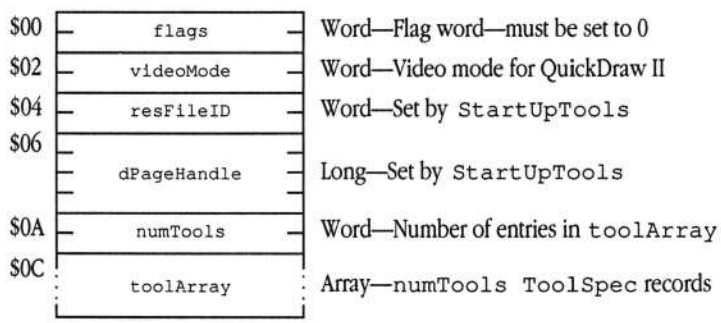
QuickDraw II	Started with the video mode from the input `StartStop` record—the `QDStartUp` *maxWidth* parameter is set to 160 bytes.
QuickDraw II Auxiliary	System calls `WaitCursor`; your application must change the cursor to an arrow before accepting user input.
Event Manager	Queue size set to 20, maximum mouse clamp set to either 320 or 640, depending on 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 use `StartInts` to enable interrupts. The Note Sequencer automatically starts the Sound Tool Set and the Note Synthesizer if you have not included them in your `StartStop` record.

The `ShutDownTools` call performs the following steps during tool set shutdown:

1. It shuts down tools specified in input `StartStop` record.
2. It disposes of the handle to the direct page.
3. It disposes of the handle to `StartStop` record (unless pointer was passed).
4. It shuts down the Resource Manager.

Both these calls require that your application format a tool `StartStop` record. That record is defined as shown in Figure 51-1.

■ **Figure 51-1** Tool set `StartStop` record

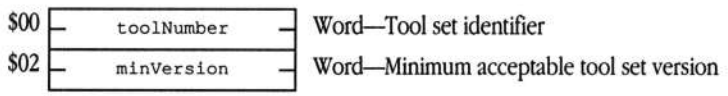


`videoMode` The video mode for QuickDraw II. See Chapter 16, "QuickDraw II," in Volume 2 of the *Toolbox Reference* for valid values.

`resFileID` The `StartUpTools` call sets this field, which `ShutDownTools` requires as input.

`dPageHandle` The `StartUpTools` call sets this field, which `ShutDownTools` requires 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:



`toolNumber` The tool set to be loaded. Valid tools set numbers are listed in Table 51-1.

`minVersion` The minimum acceptable version for the tool set. See Chapter 24, "Tool Locator," in Volume 2 of 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 (Disk Utilities)
$19 (#25)	Note Synthesizer
$1A (#26)	Note Sequencer
$1B (#27)	Font Manager
$1C (#28)	List Manager
$1D (#29)	Audio Compression and Expansion (ACE)

[continued]

■ **Table 51-1** Tool set numbers [continued]

Tool set number	Tool set name
$1E (#30) /0	Resource Manager
$20 (#32) /	MIDI Tool Set
$22 (#34) /0	TextEdit Tool Set
$21 (#33) /0	Video Overlay Tool Set
$23 (#35) /0	Midi Synth Tool Set
VF $26 (#38) / 0	Media Control Tool Set
$25-(#35) _____	Animation Tools

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 tool sets required to ensure correct system operation. Table 51-2 documents current tool set dependencies.

■ **Table 51-2** Tool set dependencies

Tool set and number	Dependencies
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) Menu Manager (#15) Control Manager (#16) LineEdit Tool Set (#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 Tool Set (#09)	Tool Locator (#01)
SANE Tool Set (#10)	Tool Locator (#01) Memory Manager (#02)
Integer Math Tool Set (#11)	Tool Locator (#01)

[continued]

■ **Table 51-2** Tool set dependencies [continued]

Tool set and number	Dependencies	
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)	
	Menu Manager (#15)	
	Control Manager (#16)	
	LineEdit Tool Set (#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)	

[continued]

■ **Table 51-2** Tool set dependencies [continued]

Tool set and number	Dependencies
◆ *Note:* QuickDraw II Auxiliary uses the Font Manager in its 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) Event Manager (#06) Window Manager (#14) Menu Manager (#15) Control Manager (#16) QuickDraw II Auxiliary (#18) LineEdit Tool Set (#20) Dialog Manager (#21) Font Manager (#27) List Manager (#28)
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) Menu Manager (#15) Control Manager (#16) QuickDraw II Auxiliary (#18) (for `Text2` items only) LineEdit Tool Set (#20) Font Manager (#27) (for `Text2` items only)

[continued]

■ **Table 51-2** Tool set dependencies [continued]

Tool set and number	Dependencies	
◆ *Note:* The LineEdit Tool Set and the Dialog Manager require the Font Manager and QuickDraw II Auxiliary if you use `LETextBox2` or `LongStatText2`, which sometimes require font styling (for example, outline, boldface, and so on).		
Scrap Manager (#22)	Tool Locator (#01)	
	Memory Manager (#02)	
Standard File Operations Tool Set (#23)		
	Tool Locator (#01)	
	Memory Manager (#02)	
	Miscellaneous Tool Set (#03)	
	QuickDraw II (#04)	
	Event Manager (#06)	
	Window Manager (#14)	
	Menu Manager (#15)	
	Control Manager (#16)	
	LineEdit Tool Set (#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 (#08) and the Note Synthesizer (#25).		
Font Manager (#27)	Tool Locator (#01)	
	Memory Manager (#02)	
	Miscellaneous Tool Set (#03)	(for `ChooseFont` only)
	QuickDraw II (#04)	
	Integer Math Tool Set (#11)	(for `ChooseFont` only)
	Window Manager (#14)	(for `ChooseFont` only)
	Menu Manager (#15)	(for `FixFontMenu` only)
	Control Manager (#16)	(for `ChooseFont` only)
	List Manager (#28)	(for `FixFontMenu` and `ChooseFont` only)
	LineEdit Tool Set (#20)	(for `ChooseFont` only)
	Dialog Manager (#21)	(for `ChooseFont` only)

[continued]

■ **Table 51-2** Tool set dependencies [continued]

Tool set and number	Dependencies	
List Manager (#28)	Tool Locator (#01)	
	Memory Manager (#02)	
	Miscellaneous Tool Set (#03)	
	QuickDraw II (#04)	
	Event Manager (#06)	
	Window Manager (#14)	
	Menu Manager (#15)	
	Control Manager (#16)	
Audio Compression and Expansion (ACE) (#29)		
	Tool Locator (#01)	
	Memory Manager (#02)	
Resource Manager (#30)	Tool Locator (#01)	
MIDI Tool Set (#32)	Tool Locator (#01)	
	Memory Manager (#02)	
	Miscellaneous Tool Set (#03)	
	Sound Tool Set (#08)	
	Note Synthesizer (#25)	(For time-stamping only)

◆ *Note:* The MIDI Tool Set requires the Note Synthesizer to support the MIDI clock feature. If you are not using the MIDI clock, the Note Synthesizer is not required.

TextEdit Tool Set (#34)	Tool Locator (#01)	Version $0300
	Miscellaneous Tool Set (#03)	Version $0300
	QuickDraw II (#04)	Version $0300
	Event Manager (#06)	Version $0300
	Window Manager (#14)	Version $0300
	Menu Manager (#15)	Version $0300
	Control Manager (#16)	Version $0300
	QuickDraw II Auxiliary (#18)	Version $0206
	Scrap Manager (#22)	Version $0104
	Font Manager (#27)	Version $0204
	Resource Manager (#30)	Version $0100

New Tool Locator calls

The `SetDefaultTPT` call has been added to the Tool Locator to facilitate permanent tool patches. The `StartUpTools` and `ShutDownTools` calls provide automatic services for bringing up or removing tool sets. The `MessageByName` tool call provides facilities allowing 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; indicates whether or not to create message
– *recordPointer* –	Long—Pointer to input record
	<—SP

Stack after call

Previous contents	
– *responseRecord* –	Long—Response record from call
	<—SP

Errors

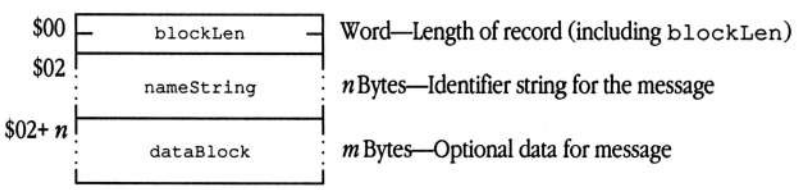
$0111	`messageNotFound`	No message found with specified name.
$0112	`messageOvfl`	No message numbers available.
$0113	`nameTooLong`	Message name too long.
Memory Manager errors		Errors from `NewHandle` returned unchanged.

C

```
extern pascal responseRecord
            MessageByName(createItFlag,
            recordPointer);

Boolean     createItFlag;
Pointer     recordPointer;
```

createItFlag Parameter determining 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 on the input record.

recordPointer Pointer to an input record defining the content and characteristics of the new message:

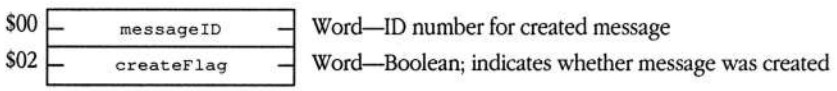


`blockLen` The length, in bytes, of the input record. Note that the value for this field includes the length of `blockLen`.

`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). To prevent message name conflict, this name string should contain the manufacturer's name, followed by the product name 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 copied into a created message. Use of this field is optional.

responseRecord The response information from the call.



`messageID` Message ID for new message, if `MessageByName` created one.

`createFlag` Flag indicating whether `MessageByName` created a message. Note that if you set *createItFlag* to TRUE on input to `MessageByName` and a message with the specified `nameString` already exists in the message center, then this flag is FALSE. In this case, `messageID` identifies the message into which your `dataBlock` was copied.

SetDefaultTPT $1601

Sets the default Tool Pointer Table (TPT) to the current TPT. Used to install a tool patch permanently.

▲ 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();
```

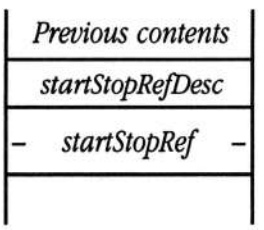
ShutDownTools $1901

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

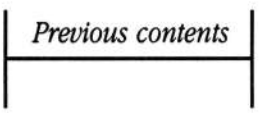


Word—Type of reference stored in *startStopRef*

Long—Reference to the `StartStop` record

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void ShutDownTools(startStopRefDesc,
                startStopRef);

Word        startStopRefDesc;
Long        startStopRef;
```

startStopRefDesc Type of reference stored in *startStopRef*.

0 Reference is by pointer
1 Reference is by handle

startStopRef Reference to the updated `StartStop` record returned by `StartUpTools`.

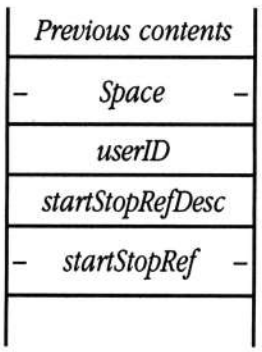
StartUpTools $1801

Starts and loads the tools specified in the input `StartStop` record. Upon successful return from `StartUpTools`, the specified tools are started, and the cursor is 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



Long—Space for result

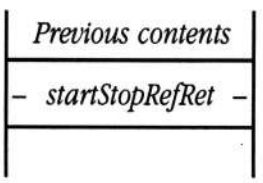
Word—Application user ID for system calls

Word—Type of reference stored in *startStopRef*

Long—Reference to the `StartStop` record

<—SP

Stack after call



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,
          startStopRefDesc, startStopRef);

Word      userID, startStopRefDesc;
Long      startStopRef;
```

startStopRefDesc Defines the type of reference stored in *startStopRef*.

0 Reference is by pointer
1 Reference is by handle
2 Reference is by resource ID

startStopRef Reference to the input `StartStop` record.

startStopRefRet 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, `StartUpTools` returns a handle.

Chapter 52 **Window Manager Update**

This chapter documents new features of the Window Manager. The complete reference to the Window Manager is in Volume 2, Chapter 25 of the *Apple IIGS Toolbox Reference.*

Error corrections

This section corrects some errors in the documentation of the Window Manager in Volume 2 of the *Toolbox Reference.*

- The description of `SetZoomRect` is incorrect. The correct description is as follows:

  Sets the `fZoomed` bit of the window's `wFrame` record to 0. The rectangle passed to `SetZoomRect` then becomes the window's zoom rectangle. The window's size and position when `SetZoomRect` is called become the window's unzoomed size and position, regardless of what the unzoomed characteristics were before `SetZoomRect` was called.

- "If `wmTaskMask` bit `tmInfo` (bit 15) = 1," on page 25-126, should read, "If `wmTaskMask` bit `tmInfo` (bit 15) = 0."

- When used with a window that does not have scroll bars, the `WindNewRes` call invokes the window's defProc to recompute window regions. A call to `SizeWindow` is not necessary under these circumstances.

- The input region for the `InvalRgn` tool call is defined in local coordinates; however, the call returns the region expressed in global coordinates.

- There are two errors in the series of equations given with the `PinRect` tool call. In the last two equations the greater-than sign (>) should be replaced with a greater-than-or-equal sign (>=).

- Note that the `CloseWindow` tool call does *not* change the GrafPort setting. Your application should ensure that a valid GrafPort is set before performing any other actions.

Clarifications

This section elaborates on topics addressed in Volume 2 of the *Toolbox Reference.*

- Window title strings should always contain leading and trailing space characters. This spacing is especially important for windows with a lined window bar because, without the spaces, the line pattern runs into the title text. Also, because window editor desk accessories may allow the user to change the title bar pattern without making the change known to your application, you should pad your window titles with spaces even if you use black title bars.
- Table 25-6 on page 25-43 of the *Toolbox Reference* contains misleading labels. Note that in this table byte 1 refers to the high-order byte of the long that defines the desktop pattern, and byte 4 refers to the low-order byte.

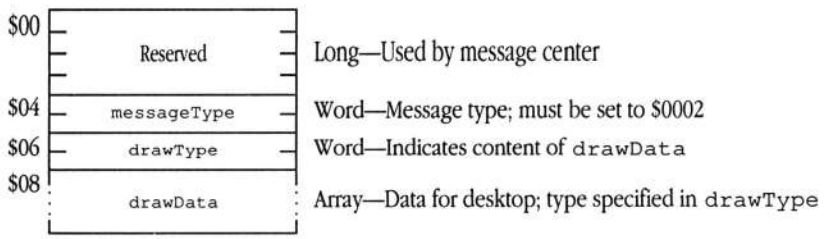
New features of the Window Manager

This section explains new features of the Window Manager and clarifies points that were not made explicit before.

- TaskMaster now brings application windows to the front after dragging is complete. TaskMaster previously brought windows to the front before dragging.
- Using the `SetOriginMask` call, a programmer can control the horizontal scrolling characteristics of windows whose scrolling is handled by TaskMaster. A common use of `SetOriginMask` is to ensure that the window origin is aligned on an even pixel so that colors do not change if the display mode is changed from 320 to 640 or vice versa. When using the call, be sure that the horizontal scroll value is a whole multiple of the mask value. Otherwise, strange behavior can occur. As an extreme example, consider an origin value of 32 and a scroll amount of 1. In this case, using the right scroll arrow causes no scrolling at all, and using the left one causes scrolling by a value of 32. The new control value for the scrolling is calculated by adding or subtracting the scroll value and the current value and applying the mask. In this case adding 1 and masking results in the original value. Subtracting 1 and masking results in a new value that is 32 less than the old value.
- The titles of standard windows can now be drawn in 16 colors regardless of mode.
- The *grid* parameter of the `DragWindow` call has been renamed *dragFlag*. Bits 0 through 7 specify the *grid* value. Bits 8 through 14 are reserved bits; they must be set to 0. Bit 15 is a selection flag; if its value is 1, then the window is brought to the top after dragging.

  It is no longer possible to specify *grid* values of 256 or 512.

- The Window Manager now uses the same default desktop drawing scheme as the Finder. When the Window Manager starts up, it looks for a `DeskMessage` call in the message center. This `DeskMessage` is formatted as follows:



`drawType` The type of data stored in `drawData`.

0	Pattern information
1	Picture information

`drawData` The pattern or picture data for the desktop image. If `drawType` is set to 0, then `drawData` contains 32 bytes of pattern data. The pattern defines 64 pixels arranged in an 8-by-8 array. In 320 mode, 4 bits are needed for each pixel; in 640 mode, the system requires 2 bits per pixel. The system uses this pattern to seed the desktop image.

If `drawType` is set to 1, then `drawData` contains 32,000 bytes of picture data; the system copies this data directly to screen memory. See Chapter 16, "QuickDraw II," in Volume 2 of the *Toolbox Reference* for details on pattern or picture images.

By loading a correctly formatted `DeskMessage` into the message center, your program can set a custom desktop image.

- The Window Manager now supports a new entry point, `TaskMasterDA`, that allows desk accessories to use TaskMaster. Previously, desk accessories could not rely on TaskMaster, because they had to work with applications that do not use TaskMaster. Desk accessories obtain the data for their task record from the Desk Manager. TaskMaster processes task records for desk accessories in the same way that it processes application task records.

- The `SizeWindow` and `ResizeWindow` tool calls now invoke the `NotifyCtls` Control Manager tool call whenever the user changes the window size. This allows applications to show a control in a constant position with respect to the lower or right border of a window. For example, now the `growControl` control definition procedure can automatically move controls in response to the dragging of the size box by the user.
- The `SetWTitle` and `GetWTitle` tool calls now allow you to store window titles in handles. Set bit 31 (the high-order bit) of the *titlePtr* parameter to the `SetWTitle` call to 1 to indicate that the parameter contains a handle to the title string. Similarly, the high-order bit in the value returned from `GetWTitle` is set to 1 if it contains a handle rather than a pointer. You must set that bit to 0 before using the handle.

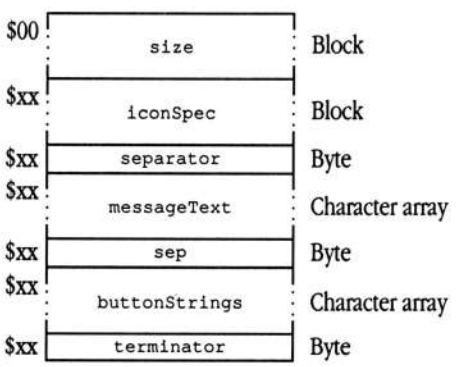
  Note that once you have called `SetWTitle`, the Window Manager owns the handle and disposes of it when you close or retitle the window. Your program must not dispose of the handle or modify the data it contains.

Alert windows

The new `AlertWindow` call (described in "New Window Manager Calls" later in this chapter) can be used to create **alert windows** that display important messages for the user. An alert window is similar to a modal dialog box. It requires the user to click a button in the window before doing anything else, and so provides a useful way to communicate vital messages such as warnings or error reports. The call does all the work of creating and displaying the window and contents of the alert window, and it returns the ID of the button that the user clicks.

The `AlertWindow` call accepts a reference to an ASCII string that contains its message, and it also accepts a reference to an array of substitution strings. The substitution strings can be any of seven standard strings (such as "OK," "Continue," and so on) or can be specified by the application and stored in the buffer to which the substitution-string pointer refers. The format of the `AlertWindow` input string is shown in Figure 52-1.

■ **Figure 52-1** `AlertWindow` input string layout



`size` A variable-length block that specifies the size of the alert window to be displayed. Valid ASCII values for the first byte lie in the range from `0` through `9` ($30 through $39) and have the following meanings:

`0` ($30)	Custom size and position, specified by rectangle definition (as shown below)
`1` ($31)	30-character display window
`2` ($32)	60-character display window
`3` ($33)	110-character display window
`4` ($34)	175-character display window
`5` ($35)	110-character display window
`6` ($36)	150-character display window
`7` ($37)	200-character display window
`8` ($38)	250-character display window
`9` ($39)	300-character display window

If the value of the first byte of `size` is not `0` ($30), then the block consists only of that byte. If `size` is set to `0` ($30), then you must specify the custom rectangle immediately after the `size` field.

`v1`	Word—y coordinate of upper-left corner
`h1`	Word—x coordinate of upper-left corner
`v2`	Word—y coordinate of lower-right corner
`h2`	Word—x coordinate of lower-right corner

Because `AlertWindow` provides a limited number of standard sizes, it is possible to create alert windows that are displayed properly whether the Apple IIGS computer is in 320 or 640 mode. It is necessary, however, to design the text and buttons carefully so that the display is correct regardless of the mode.

Table 52-1 shows the dimensions of the standard alert windows. This table gives only an approximate idea of the size of each window. Application code should not rely on the exact widths, heights, or position of standard windows.

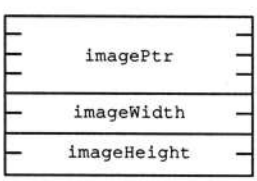
■ **Table 52-1** Standard alert window sizes

`size` value	Height 320	Width 320	Height 640	Width 640
1 ($31)	46	152	46	200
2 ($32)	62	176	54	228
3 ($33)	62	252	62	300
4 ($34)	90	252	72	352
5 ($35)	54	252	46	400
6 ($36)	62	300	54	452
7 ($37)	80	300	62	500
8 ($38)	108	300	72	552
9 ($39)	134	300	80	600

`iconSpec` A variable-length block that specifies the type of icon to be displayed in the alert window. Valid values for the first byte lie in the range from 0 through 9 ($30 through $39) and have the following meanings:

0 ($30)	No icon
1 ($31)	Custom icon; followed by an icon specification, as shown below
2 ($32)	Stop icon
3 ($33)	Note icon
4 ($34)	Caution icon
5 ($35)	Disk icon
6 ($36)	Disk swap icon
7–9 ($37–$39)	Reserved

If the first byte of `iconSpec` has a value other than 1 ($31), then the field consists only of that byte. If the first byte is set to 1 ($31), then it must be followed by an icon specification.



Long—Pointer to image data

Word—Width in bytes of the image data

Word—Height in scan lines of the image data

`separator` A character that divides substrings in the remainder of the `AlertWindow` input string. The `separator` field can contain any character, but the character cannot appear in the message text or button strings. The `separator` character differentiates the text of the message from the title of the first button, and the button titles from each other. For purposes of standardization, the slash (`/`) character is recommended, unless you will be substituting pathnames.

Do not include a separator character in any substitution strings. The Window Manager performs substitutions before scanning the alert string for separators. For example, if the separator character is a slash and a pathname containing several slashes is substituted for the string, the resulting alert window will contain several more buttons than you intended.

`messageText` The message to be displayed in the alert window. Any characters allowed by `LETextBox2` are allowed in the message text. See "Special Characters" later in this chapter for additional characteristics of `AlertWindow` message text. The total size of message text, after substitution of strings, is limited to 1000 characters.

`sep` A `separator` character.

`buttonStrings` Titles for up to three buttons to be displayed in the alert window. If there is more than one title, then the titles must be demarcated by a `separator` character. These buttons will be evenly spaced and centered at the bottom of the alert window. The width of all buttons is the same and is determined by the longest button title. The maximum length of button text after substitution of strings is 80 characters.

`terminator` The end of the alert string. Must be set to 0 ($00).

Special characters

The following *special characters* can be embedded in the message text and button strings of an `AlertWindow` input string. If a special character is to appear in the text of a button or message, you must enter it twice in the string. For example, if you want `^` to appear in an alert message, you must enter it in the message string as `^^`.

`^` Designates the default button. The default button is the button selected if the user presses the Return key on the keyboard. This button appears outlined in bold on the screen. Only one button can be the default button. Like all buttons, the default button must have a title, which in this case follows the caret. Other special characters may also appear after the caret. A single caret in the body of message text has no effect and is deleted from the message.

`#` Substitute standard string. The number sign (`#`) must be followed by an ASCII number character from `0` through `6`. Numbers `7` through `9` are reserved and should not be used. The standard substitution strings are

`#0`	OK
`#1`	Cancel
`#2`	Yes
`#3`	No
`#4`	Try again
`#5`	Quit
`#6`	Continue

`*` Substitute given string. The asterisk (`*`) character followed by an ASCII number character from `0` through `9` denotes a substitution string to be inserted at that point. The asterisk and the number following it are replaced by the corresponding string in the specified substitution array. A pointer to the substitution array is passed as a parameter to the `AlertWindow` call. The substitution array is defined as an array of pointers. Table 52-2 shows the format of a substitution string array.

- **Table 52-2** Substitution string array

LONG[0]	Pointer to string that will substitute for `*0`
LONG[1]	Pointer to string that will substitute for `*1`
LONG[2]	Pointer to string that will substitute for `*2`
LONG[3]	Pointer to string that will substitute for `*3`
LONG[4]	Pointer to string that will substitute for `*4`
LONG[5]	Pointer to string that will substitute for `*5`
LONG[6]	Pointer to string that will substitute for `*6`
LONG[7]	Pointer to string that will substitute for `*7`
LONG[8]	Pointer to string that will substitute for `*8`
LONG[9]	Pointer to string that will substitute for `*9`

Substitution strings can be C strings or Pascal strings. A parameter to the `AlertWindow` tool call allows you to specify the type of strings in the substitution array.

Alert window example

This section includes some examples of alert strings that can be passed to `AlertWindow` in 65816 assembly-language syntax.

Figure 52-2 shows a simple alert string.

- **Figure 52-2** An alert string

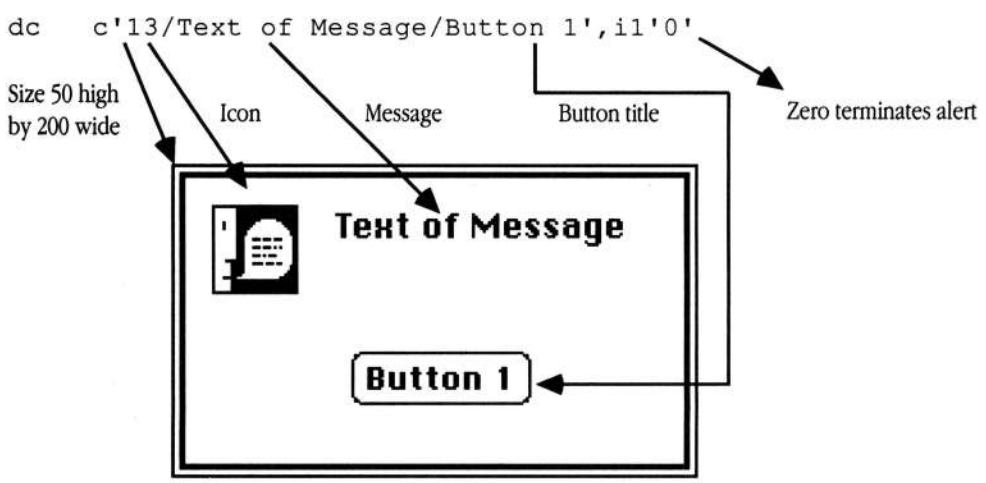


Figure 52-3 shows a more complex alert string that defines a custom rectangle.

```
dc   c'0',i2'35,100,81,500'
dc   c'1/This is the *0 of *3 alert *2*1 and standard'
dc   c'text called "#4."/'
dc   c'^#0,Really/*4/Yo!',i1'0'
```

- **Figure 52-3** An alert string defining a custom rectangle

This is the substitution array in this case:

```
      dc   i4'sub0,sub1,sub2,sub3,sub4'
sub0  dc   c'message text',i1'0'
sub1  dc   c'dow',i1'0'
sub2  dc   c'win',i1'13'
sub3  dc   c'an',i1'0'
sub4  dc   c'Door #2',i1'0'
```

TaskMaster result codes

Table 52-3 lists all the possible TaskMaster result codes.

■ **Table 52-3** TaskMaster result codes

Name	Value	Description
NULL	$0000	Successful
mouseDownEvt	$0001	Event code
mouseUpEvt	$0002	Event code
keyDownEvt	$0003	Event code
autoKeyEvt	$0005	Event code
updateEvt	$0006	Event code
activateEvt	$0008	Event code
switchEvt	$0009	Event code
deskAccEvt	$000A	Event code
driverEvt	$000B	Event code
app1Evt	$000C	Event code
app2Evt	$000D	Event code
app3Evt	$000E	Event code
app4Evt	$000F	Event code
wNoHit	$0000	Alias for no event
inNull	$0000	Alias for no event
inKey	$0003	Alias for keystroke
inButtDwn	$0001	Alias for button-down event
inUpdate	$0006	Alias for update event
wInDesk	$0010	On desktop
wInMenuBar	$0011	On system menu bar
wClickCalled	$0012	SystemClick called (returned only as action)
wInContent	$0013	In content region
wInDrag	$0014	In drag region
wInGrow	$0015	In grow region, active window only
wInGoAway	$0016	In go-away region, active window only

[continued]

■ **Table 52-3** TaskMaster result codes [continued]

Name	Value	Description
`wInZoom`	$0017	In zoom region, active window only
`wInInfo`	$0018	In information bar
`wInSpecial`	$0019	Item ID selected was 250–255
`wInDeskItem`	$001A	Item ID selected was 1–249
`wInFrame`	$001B	In frame, but not on anything else
`wInactMenu`	$001C	Inactive menu item selected
`wClosedNDA`	$001D	Desk accessory closed (returned only as action)
`wCalledSysEdit`	$001E	`SystemEdit` called (returned only as action)
`wTrackZoom`	$001F	Zoom box clicked, but not selected (action only)
`wHitFrame`	$0020	Button down on frame, made active (action only)
`wInControl`	$0021	Button or keystroke in control (can be returned as event code and as action)
`wInControlMenu`	$0022	Control-handled menu item
`wInSysWindow`	$8000	High bit set for system windows

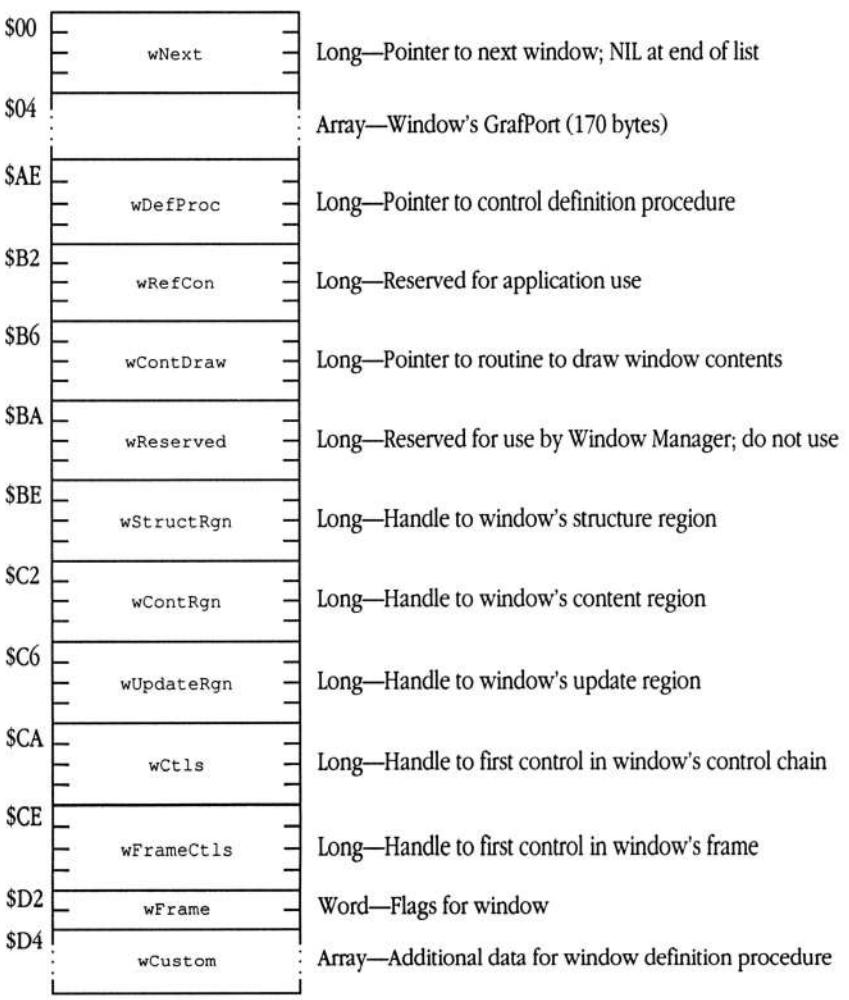
Window Manager data structures

This section discusses the format and content of changed Window Manager data structures.

Window record

The window record data structure has been redefined. Figure 52-4 illustrates the new definition.

- **Figure 52-4** Window record definition



`wReserved` A new data field reserved by Apple Computer, Inc., for future expansion.

`wFrame` A bit flag containing flags specifying the window frame. All of the bits in this flag are described in Chapter 25, "Window Manager," in Volume 2 of the *Toolbox Reference*. Some of these bits may be used by window definition procedures. The following bits may be used by window defProcs.

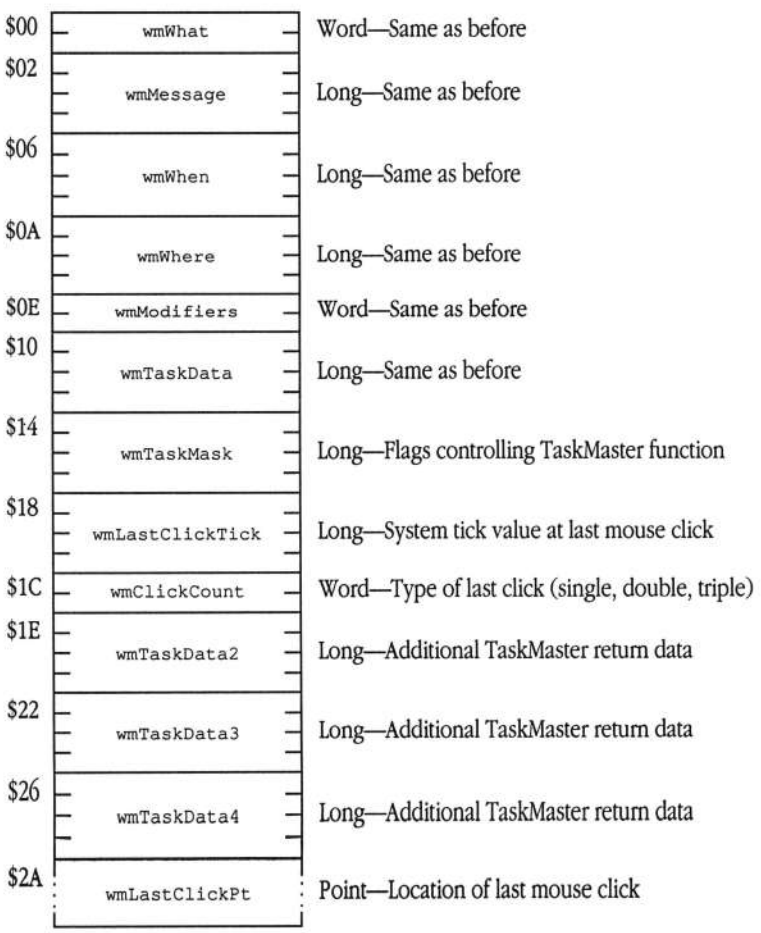
`fTitle`	bit 15
`fClose`	bit 14
`fAlert`	bit 13
`fRScroll`	bit 12
`fBScroll`	bit 11
`fGrow`	bit 10
`fFlex`	bit 9
`fZoom`	bit 8
`fMove`	bit 7
`fInfo`	bit 4
`fZoomed`	bit 1

Task record

Figure 52-5 defines the new format for the task record. This new record layout includes several new fields, each of which is set by TaskMaster every time your program calls TaskMaster. For information on the old fields, see Chapter 25, "Window Manager," in Volume 2 of the *Toolbox Reference.*

TaskMaster still accepts task records in the old format; however, if your program uses any of the new TaskMaster features (see description of `wmTaskMask` on next page), it must use the new record layout.

■ **Figure 52-5** Task record definition



`wmtAskMask` Flags controlling TaskMaster functions.

Reserved	bits 31–21	Must be set to 0.
`tmIdleEvents`	bit 20	Controls whether TaskMaster sends idle events to the target control in the active window. 0 = Do not send idle events 1 = Send idle events
`tmMultiClick`	bit 19	Controls whether TaskMaster returns multiclick information in the task record. 0 = Do not return multiclick information 1 = Return multiclick information
`tmControlMenu`	bit 18	Controls whether TaskMaster passes menu events to controls in the active window. 0 = Do not pass menu events 1 = Pass menu events
`tmControlKey`	bit 17	Controls whether TaskMaster passes key events to controls in the active window. 0 = Do not pass key events 1 = Pass key events
`tmContentControls`	bit 16	Controls whether TaskMaster calls `FindControl` and `TrackControl` when `FindWindow` returns `wInContent` and the window is already selected. 0 = Do not track the control 1 = Track the control
`tmInfo`	bit 15	Controls whether TaskMaster activates the window when the user clicks in the information bar. 0 = Activate the window 1 = Do not activate the window
`tmInactive`	bit 14	Controls whether TaskMaster returns `wInactMenu` when the user selects an inactive menu item. 0 = Never return `wInactMenu` 1 = Return `wInactMenu`
`tmCRedraw`	bit 13	Controls whether TaskMaster redraws controls whenever an activate event occurs. 0 = Do not redraw controls 1 = Redraw controls
`tmSpecial`	bit 12	Controls whether TaskMaster handles special menu items (those with IDs < 256). 0 = Do not handle special menu items 1 = Handle special menu items

`tmScroll`	bit 11	Controls whether TaskMaster enables scrolling and activates inactive windows when the user clicks on the scroll bar. 0 = Do not enable scrolling 1 = Enable scrolling
`tmGrow`	bit 10	Controls whether TaskMaster calls `GrowWindow` when the user drags the size box. 0 = Do not call `GrowWindow` 1 = Call `GrowWindow`
`tmZoom`	bit 9	Controls whether TaskMaster calls `TrackZoom` when the user clicks in the zoom box. 0 = Do not call `TrackZoom` 1 = Call `TrackZoom`
`tmClose`	bit 8	Controls whether TaskMaster calls `TrackGoAway` when the user clicks in the close box. 0 = Do not call `TrackGoAway` 1 = Call `TrackGoAway`
`tmContent`	bit 7	Controls whether TaskMaster activates the window when the user clicks in the content region. 0 = Do not activate window 1 = Activate window
`tmDragW`	bit 6	Controls whether TaskMaster calls `DragWindow` when the user drags in the drag region. 0 = Do not call `DragWindow` 1 = Call `DragWindow`
`tmSysClick`	bit 5	Controls whether TaskMaster calls `SystemClick` when the user clicks in the system window. 0 = Do not call `SystemClick` 1 = Call `SystemClick`
`tmOpenNDA`	bit 4	Controls whether TaskMaster calls `OpenNDA` when the user selects a desk accessory. 0 = Do not call `OpenNDA` 1 = Call `OpenNDA`
`tmMenuSel`	bit 3	Controls whether TaskMaster calls `MenuSelect` when the user clicks in the menu bar. 0 = Do not call `MenuSelect` 1 = Call `MenuSelect`
`tmFindW`	bit 2	Controls whether TaskMaster calls `FindWindow` for mouse-down events. 0 = Do not call `FindWindow` 1 = Call `FindWindow`

`tmUpdate`	bit 1	Controls whether TaskMaster handles update events. 0 = Do not handle update events 1 = Handle update events
`tmMenuKey`	bit 0	Controls whether TaskMaster calls `MenuKey` to handle key equivalents for menu commands. 0 = Do not call `Menukey` 1 = Call `MenuKey`

New Window Manager calls

The following tool calls have been added to the Window Manager since the publication of the first two volumes of the *Toolbox Reference.*

`AlertWindow` $590E

Creates an alert window that displays a message referred to by *alertStrRef.* The *subStrPtr* parameter points to an array of substitution strings for use with substitution characters. The substitution strings can be either C or Pascal strings, as specified by *alertFlags.* For more detailed information, see "Alert Windows" earlier in this chapter.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
alertFlags	Word—Flag word for call
– *subStrPtr* –	Long—Pointer to substitution array
– *alertStrRef* –	Long—Reference to alert string; *alertFlags* indicates type
	<—SP

Stack after call

Previous contents	
Result	Word—Button number selected (0 relative, in order created)
	<—SP

Errors None

C

```
extern pascal Word AlertWindow(alertFlags,
          subStrPtr, alertStrRef);

Word      alertFlags;
Pointer   subStrPtr;
Long      alertStrRef;
```

alertFlags Flags that indicate the type of strings referenced by *subStrRef,* as well as the type of reference contained *alertStrRef:*

Reserved	bits 15–3	Must be set to 0.
`referenceType`	bits 2–1	Indicate the type of reference stored in *alertStrRef.* 00 = *alertStrRef* is a pointer 01 = *alertStrRef* is a handle 10 = *alertStrRef* is a resource ID 11 = Invalid value
`stringType`	bit 0	Indicates type of string referred to by *subStrPtr.* 0 = C string (null-terminated) 1 = Pascal string

`CompileText` $600E

Combines source text provided by your program with either custom or standard strings to compile a result text string. For successful calls, this call allocates and correctly sizes a handle to the result text string. That result string is a simple character array. Your program must extract length information for the string from the handle. Note that your program must dispose of this handle.

Control sequences in the source text direct the system to embed either custom or standard strings into the result text string. These control sequences consist of two ASCII characters: a flag character followed by a digit. The flag character indicates whether the desired substitution string is custom or standard.

For standard strings, the flag character is `#`. The digit following the flag character designates one of the following strings:

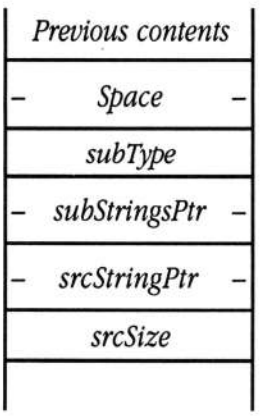
`#0`	OK
`#1`	Cancel
`#2`	Yes
`#3`	No
`#4`	Try again
`#5`	Quit
`#6`	Continue

For custom strings, the flag character is `*`. The `CompileText` call obtains custom strings from a substitution array built by your program and provided to the system in the parameters for this call. The ASCII character following the flag character specifies which string to extract. Valid values for this ASCII character lie in the range `0` through `9`. Thus, a control sequence of `*0` would cause the first string in your custom substitution array to be accessed.

To include either of the flag characters as text in your compiled text, follow the flag character with a second flag character (for example, `**` causes `*` to be displayed in the compiled text string).

Parameters

Stack before call



Long—Space for result

Word—Type of custom substitution strings

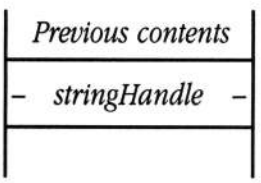
Long—Pointer to substitution array

Long—Pointer to source string

Word—Length of source string pointed to by *srcStringPtr*

<—SP

Stack after call



Long—Handle to result string

<—SP

Errors

$0E04	`compileTooLarge`	Compiled text is larger than 64 KB.

C

```
extern pascal Handle CompileText(subType,
          subStringsPtr, srcStringPtr, srcSize);

Word      subType, srcSize;
Pointer   subStringsPtr, srcStringPtr;
```

subType Indicates the type of strings stored in the substitution array pointed to by *subStringsPtr.*

0	C strings
1	Pascal strings

Note that this field is ignored if your program uses no custom substitution strings.

subStringsPtr A pointer to your custom text substitution array. This array contains from 1 to 10 long pointers to either C or Pascal strings (use *subType* to indicate which type of string you have used). Embedded control sequences in your source text direct the system to extract a specific string from this array. Note that the system does not verify string specifications against the size of this array; be careful to define the correct number of string pointers in this array.

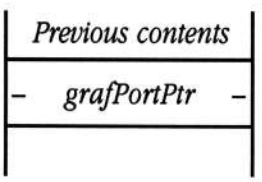
Note that this field is ignored if your program uses no custom substitution strings.

DrawInfoBar $550E

Redraws the information bar of the window specified by *grafPortPtr*. The routine that redraws the interior of the information bar is specified by the *wInfoDefProc* field of the `paramList` passed to `NewWindow` when the window is created. The Window Manager automatically clips the drawing in the information bar to the dimensions of the information bar and to the visible region of the window.

Parameters

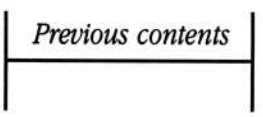
Stack before call



Long—Pointer to GrafPort for window

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void DrawInfoBar(grafPortPtr);

Pointer   grafPortPtr;
```

EndFrameDrawing $5B0E

Restores Window Manager variables after a call to `StartFrameDrawing`.

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

Errors None

C

```
extern pascal void EndFrameDrawing();
```

ErrorWindow $620E

Creates a dialog box displaying an error message for a specified error code. GS/OS error codes are listed along with standard message text in "Error Messages" later in this chapter.

Each error message is in alert string format and may require a substitution string (see "Alert Windows" earlier in this chapter for message format and text substitution information). The system retrieves the error messages from a resource file containing resources of type `rErrorString` ($8020). The resource ID for each message is formed as follows:

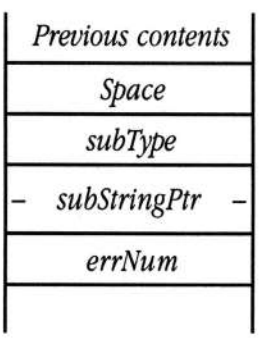
high-order word	$07FF
low-order word	error number

The default error messages are stored in the system resource file. You may assert custom error message text by defining and opening another resource file containing `rErrorString` resources with appropriate resource IDs assigned to each error message. Make sure that your resource file precedes the system resource file in the Resource Manager's search sequence. A custom error message resource file need not define substitute messages for all possible GS/OS errors; if the Resource Manager does not find a message in your file, it continues through the standard resource search sequence.

If `ErrorWindow` receives an undefined error code, it displays a dialog box with the `Unknown error` message ($72).

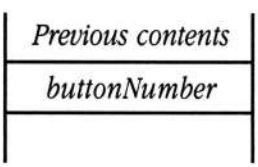
Parameters

Stack before call



Word—Space for result
Word—Type of custom substitution string
Long—Pointer to substitution string
Word—GS/OS error number
<—SP

Stack after call



Word—Number of the button clicked by the user
<—SP

Errors Resource Manager errors Returned unchanged.

C

```
extern pascal Word ErrorWindow(subType,
          subStringPtr, errNum);

Word      subType, errNum;
Pointer   subStringPtr;
```

subType The type of string pointed to by *subStringPtr.*

0	C string
1	Pascal string

Note that this field is ignored if the specified error message contains no substitution strings.

subStringPtr A pointer to your custom text substitution string. Note that this field is ignored if the specified error message contains no substitution strings.

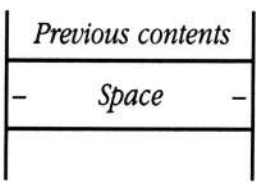
GetWindowMgrGlobals $580E

Returns a pointer to the Window Manager global data area.

▲ **Warning** An application should never make this call. ▲

Parameters

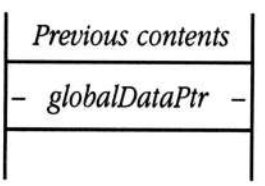
Stack before call



Long—Space for result

<—SP

Stack after call



Long—Pointer to the global data area

<—SP

Errors None

C

```
extern pascal Pointer GetWindowMgrGlobals();
```

NewWindow2 $610E

Performs the same function as NewWindow but allows you to specify the input window template as a resource (type rWindParam1, $800E, or rWindParam2, $800F). See Appendix E, "Resource Types," later in this book for complete descriptions of all resource types.

- *Note:* If you have specified the window template as a resource, then the references within that template to title, color table, and control list must also be resources (or NIL).

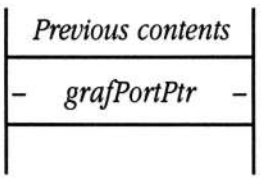
If you use NewWindow2 specifying the window template as a resource, to create an information bar you must specify a NIL infoDraw procedure in the input template and create an invisible window. After issuing the NewWindow2 call, set the infoDraw routine by calling SetInfoDraw, then use the ShowWindow tool call to make the window visible.

Parameters

Stack before call

Previous contents	
– *Space* –	Long—Space for result
– *titlePtr* –	Long—Pointer to replacement title
– *refCon* –	Long—RefCon to replace value in template
– *contentDrawPtr* –	Long—Pointer to replacement content-draw routine
– *defProcPtr* –	Long—Pointer to replacement window definition procedure
paramTableDesc	Word—Type of reference in *paramTableRef*
– *paramTableRef* –	Long—Reference to window template
resourceType	Word—Resource type of template referred to by *paramTableRef*
	<—SP

Stack after call



Long—Pointer to window GrafPort; NIL if unsuccessful

<—SP

Errors

Resource Manager errors	Returned unchanged.
Memory Manager errors	Returned unchanged.
Window Manager errors	Returned unchanged from `NewWindow`.
Control Manager errors	Returned unchanged from `NewControl2`.

C

```
extern pascal Pointer NewWindow2(titlePtr, refCon,
          contentDrawPtr, defProcPtr,
          paramTableDesc, paramTableRef,
          resourceType);

Word      paramTableDesc, resourceType;
Pointer   titlePtr, contentDrawPtr, defProcPtr;
Long      refCon, paramTableRef;
```

titlePtr, refCon, contentDrawPtr, defProcPtr

The `NewWindow2` call replaces the values supplied in the template referred to by *paramTableRef* with the contents from these fields, allowing you to use a standard template and tailor it to create different windows. To prevent `NewWindow2` from replacing the template values, supply NIL pointers in *titlePtr, contentDrawPtr,* and *defProcPtr.*

paramTableDesc The type of reference stored in *paramTableRef.*

$0000	*paramTableRef* contains a pointer to a window template
$0001	*paramTableRef* contains a handle to a window template
$0002	*paramTableRef* contains the resource ID of a window template

paramTableRef Reference to a window template. The *paramTableDesc* field defines the type of reference stored here. The *resourceType* field defines the resource type for the template. The template must comply with the format specification of resource type `rWindParam1` or `rWindParam2` (even if the template is not stored as a resource). See Appendix E, "Resource Types," in this book for information on the format and content of these resources.

resourceType The type of window template referred to by *paramTableRef*. This value should be set correctly even if *paramTableRef* does not contain a resource ID. Valid values are

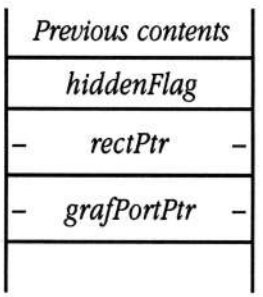
$800E	`rWindParam1`
$800F	`rWindParam2`

ResizeWindow $5C0E

Moves, resizes, and draws the window specified by *grafPortPtr*. The *rectPtr* parameter is a pointer to the window's content region. The *hiddenFlag* parameter is a Boolean value. A TRUE value specifies that those portions of the window that are covered should not be drawn. If the value is FALSE, all parts of the window, covered or not, are drawn.

Parameters

Stack before call



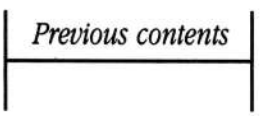
Word—Boolean; whether to hide covered area

Long—Pointer to new content rectangle

Long—Pointer to window's GrafPort

<—SP

Stack after call



<—SP

Errors None

C

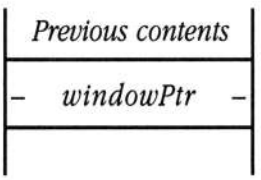
```
extern pascal void ResizeWindow(hiddenFlag, rectPtr,
          grafPortPtr);

Word      hiddenFlag;
Pointer   rectPtr, grafPortPtr;
```

StartFrameDrawing $5A0E

Sets up Window Manager data to draw a window frame. This should be called only by window definition procedures and must be balanced by a call to `EndFrameDrawing` when drawing is completed.

Parameters

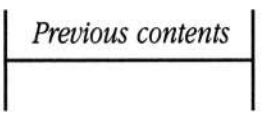
Stack before call



Long—Pointer to the window to be drawn

<—SP

Stack after call



<—SP

Errors None

C

```
extern pascal void StartFrameDrawing(windowPtr);

Pointer   windowPtr;
```

TaskMaster $1D0E

This section presents revised pseudocode for TaskMaster.

Pseudocode

```
Call SystemTask.

Call GetNextEvent using TaskMask user passed.

The wmMessage field of TaskRec is duplicated into the wmTaskData field
      of TaskRec.

If any of the reserved bits in the TaskMask field are not 0:
{
      Low word of wmTaskData = 0.
      Returns nullEvt ($0000).
      Error returned: wmTaskMaskErr ($0E03).
}

If wmWhat of TaskRec = nullEvt ($0000):
{
      If TaskMask bit tmIdleEvents (bit 20) = 1:
      {
            If there is a front window:
            {
                  Calls the BeginUpdate routine.
                  Send idle event by calling SendEventToCtl with
                        targetOnlyFlag = True.
                  If result from SendEventToCtl = True
                        (i.e., a control accepted the idle event):
                  {
                        wmTaskData2 contains handle to control that took
                              event.
                        wmTaskData3 contains the result returned from
                              defproc.
                        wmTaskData4 contains the control's ID.
                  }
                  Calls the EndUpdate routine.
            }
      Low word of wmTaskData = 0.
      Returns nullEvt ($0000).
}
```

```
If wmWhat field of TaskRec = mouseDownEvt ($0001):
{
      If TaskMask bit tmMultiClick (bit 19) = 1:
      {
            If wmClickCount field of TaskRec <> 0
                  (then not single click):
            {
                  Calculate time between mouse clicks.
                  Call GetDblTime.
                  If time between clicks is less than
                        double-click speed:
                  {
                        If mouse position of new click is
                              near last click:
                        {
                              Increment wmClickCount field of
                                    TaskRec by one.
                              Set wmLastClickTick field of
                                    TaskRec = wmWhen.
                              Set wmLastClickPt field of
                                    TaskRec = wmWhere.
                        }
                  }
            }
            Set wmClickCount field of TaskRec = 1.
            Set wmLastClickTick field of TaskRec = wmWhen.
            Set wmLastClickPt field of TaskRec = wmWhere.
      }

      If TaskMask bit tmFindW (bit 2) = 0:
      {
            wmTaskData = message field from GetNextEvent.
            Returns mouseDownEvt ($0001).
      }

      Calls FindWindow.

      If FindWindow returns wInMenuBar ($0011):
      {
            If TaskMask tmMenuSel (bit 3) = 0:
```

```
        {
              Low word of wmTaskData = 0.
              Returns wInMenuBar ($0011).
        }
        MenuSelect is called with TaskRec passed to TaskMaster.

Menu Selection:
        If low word of wmTaskData = 0, then no selection made:
        {
              If TaskMask bit tmInactive (bit 14) = 0:
              {
                    Low word of wmTaskData = wInMenuBar ($0011).
                    Returns nullEvt ($0000).
              }

              If high word of wmTaskData = nonzero:
              {
                    Low word of wmTaskData = 0.
                    High word of wmTaskData = ID of selected
                          inactive menu item.
                    Returns wInActMenu ($001C).
              }

        If low word of wmTaskData (menu item ID) > 255:
        {
              If wmTaskMask bit tmControlMenu (bit 18) =1:
              {
                    Call SendEventToCtl with TargetOnlyFlag = True.
                    If result from SendEventToCtl = nonzero:
                    {
                          wmTaskData2 = handle of control that took
                                keystroke.
                          wmTaskData3 = result passed back from
                                defproc.
                          wmTaskData4 = ID of control that took
                                keystroke.
                          Unhilite menu title for menu item that was
                                just selected.
                          Low word wmTaskData = wInControlMenu
                                ($0022).
                          Returns nullEvt ($0000).
                    }
```

```
      Low word of wmTaskData = ID of selected menu item.
      High word of wmTaskData = ID of menu from which
            selection was made.
      Returns wInMenuBar ($0011).
      }

If low word of wmTaskData (menu item ID) < 250:
{
      If TaskMask bit tmOpenNDA (bit 4) = 0:
      {
            Low word of wmTaskData = ID of selected menu
                  item.
            High word of wmTaskData = ID of menu from which
                  selection was made.
            Returns wInDeskItem ($001A).
      }
      Calls OpenNDA with item ID in low word of wmTaskData.
      Unhilite menu title for menu item that was just
            selected.
      Low word of wmTaskData = wInDeskItem ($001A).
      Returns nullEvt ($0000).
}

If TaskMask bit tmSpecial (bit 12) = 0:
{
      Low word of wmTaskData = ID of selected menu item.
      High word of wmTaskData = ID of menu from which
            selection was made.
      Returns wInSpecial ($0019).
}

If top window is an application (nonsystem) window:
{
      If TaskMask bit tmControlMenu (bit 18) = 1:
      {
            Calls SendEventToCtl with TargetOnlyFlag = True.
            If result from SendEventToCtl  = nonzero:
            {
                  wmTaskData2 = handle of control that took
                        keystroke.
                  wmTaskData3 = result passed back from
                        defproc.
```

```
                    wmTaskData4 = ID of control that took
                          keystroke.
                    Unhilite menu title for menu item that was
                          just selected.
                    Low word of wmTaskData = wInControlMenu
                          ($0022).
                    Returns nullEvt ($0000).
              }
        }
        Low word of wmTaskData = ID of selected menu item.
        High word of wmTaskData = ID of menu from which
              selection was made.
        Returns wInSpecial ($0019).
}

If low word of wmTaskData = 250, 251, 252, 253, 254
        (edit items):
{
        Calls SystemEdit with ID of special menu item.
        If SystemEdit returns False:
        {
              Low word of wmTaskData = ID of menu item
                    selected.
              High word of wmTaskData = ID of menu from which
                    selection was made.
              Returns wInSpecial ($0019).
        }
        (Top system window handled the special menu item
              selection.)
        Unhilite menu title for menu item that was just
              selected.
        Low word of wmTaskData = wCalledSysEdit ($001E).
        Returns nullEvt ($0000).
}

If low word of wmTaskData = 255 (close item):
{
        Calls CloseNDAbyWinPtr for top window (system window).
        Unhilite menu title for menu item that was selected.
        Low word of wmTaskData = wCloseNDA ($001D).
        Returns nullEvt ($0000).
} (end menu selection)
```

```
} (end FindWindow wInMenuBar)


If FindWindow returns a negative value:
{
      If TaskMask bit tmSysClick (bit 5) = 0:
      {
            wmTaskData = window pointer returned from FindWindow.
            Returns result from FindWindow.
      }
      Calls Desk Manager routine SystemClick with result from
            FindWindow.
      wmTaskData low word = wClickCalled ($0012).
      Returns nullEvt ($0000).
}

If FindWindow returns wInDrag ($0014):
{
      If TaskMask bit tmDragW (bit 6) = 0:
      {
            wmTaskData = window pointer returned from FindWindow.
            Returns wInDrag ($0014).
      }
      If bit 8 in the modifier field of TaskRec (Apple key up) and
            the window is not active:
      {
            Calls SelectWindow to make window active.
      }
      Calls DragWindow.
      wmTaskData = wInDrag ($0014).
      Returns nullEvt ($0000).
}

If FindWindow returns wInContent ($0013):
{
      Calls TaskMasterContent.
}

If FindWindow returns wInGoAway ($0016):
{
      If TaskMask bit tmClose (bit 8) = 0:
      {
```

```
            wmTaskData = window pointer returned from FindWindow.
            Returns wInGoAway ($0016).
      }
      Calls TrackGoAway.
      If TrackGoAway returns True:
      {
            wmTaskData = window pointer returned from FindWindow.
            Returns wInGoAway ($0016).
      }
      Low word of wmTaskData = wInGoAway ($0016).
      Returns nullEvt ($0000).
}

If FindWindow returns wInZoom ($0017):
{
      If TaskMask bit tmZoom (bit 9) = 0:
      {
            wmTaskData = window pointer returned from FindWindow.
            Returns wInZoom ($0017).
      }
      Calls TrackZoom.
      If TrackZoom returns True:
      {
            Calls ZoomWindow.
            Low word of wmTaskData = wInZoom ($0017).
            Returns nullEvt ($0000).
      }
      Low word of wmTaskData = wTrackZoom ($001F).
      Returns nullEvt ($0000).
}

If FindWindow returns wInGrow ($0015):
{
      If TaskMask bit tmGrow (bit 10) = 0:
      {
            wmTaskData = window pointer returned from FindWindow.
            Returns wInGrow ($0015).
      }
      Calls GrowWindow.
      Calls SizeWindow with results from GrowWindow.
      Low word of wmTaskData = wInGrow ($0015).
      Returns nullEvt ($0000).
```

```
}

If FindWindow returns wInInfo ($0018):
{
      If TaskMask bit tmInfo (bit 15) = 0:
      {
            If window not active:
            {
                  Calls SelectWindow.
                  Low word of wmTaskData = wInInfo ($0018).
                  Returns nullEvt ($0000).
            }
      }
      wmTaskData = window pointer returned from FindWindow.
      Returns wInInfo ($0018).
}

If FindWindow returns wInFrame ($001B):
{
      If TaskMask bit tmScroll (bit 11) = 0:
      {
            wmTaskData = window pointer returned from FindWindow.
            Returns wInFrame ($001B).
      }
      If window is not active:
      {
            Calls SelectWindow to make active.
            Low word of wmTaskData = wHitFrame ($0020).
            Returns nullEvt ($0000).
      }
      If button was on a window frame control (not scroll bar
            control):
      {
            Low word of wmTaskData = wHitFrame ($0020).
            Returns nullEvt ($0000).
      }
      Calls TrackControl with an action procedure within
            TaskMaster.
      The action procedure in TaskMaster performs scrolling and
            updates.
      Low word of wmTaskData = wInFrame ($001B).
      Returns nullEvt ($0000).
```

```
      }

      Else (something returned from FindWindow other than those handled
            above):
      {
            wmTaskData = returned value from FindWindow.
            Returns result from FindWindow.
      }
} (end wmWhat field of TaskRec = mouseDownEvt)


If wmWhat field of TaskRec = keyDownEvt ($0003) OR autoKeyEvt ($0005):
{
      Calls TaskMasterKey.
}


If wmWhat field of TaskRec = activateEvt ($0008):
{
      If TaskMask bit tmCRedraw (bit 13) = 1:
      {
            If wframe bit fCtlTie (bit 3) = 0:
            {
                  Invalidate the bounds rect of all normal controls in
                        the window.
                  For all extended controls in the window send the
                        defproc message ctlWinStateChange.
            }
      }
      wmTaskData = pointer to window that was activated or deactivated
            (check modifier field).
      Returns activateEvt ($0008).
}


If wmWhat field of TaskRec = updateEvt ($0006):
{
      If TaskMask bit tmUpdate (bit 1) = 0:
      {
            wmTaskData = pointer to window to be updated.
            Returns updateEvt ($0006).
      }
```

```
        If window's wContDefProc field = 0:
        {
              wmTaskData = pointer to window to be updated.
              Returns updateEvt ($0006).
        }
        Calls BeginUpdate routine.

        The window's draw routine in window's wContDefProc field is called
        (routine in application).

        Calls EndUpdate routine.

        wmTaskData low word = updateEvt ($0006).
        Returns nullEvt ($0000).
}
```

TaskMasterContent $5D0E

Internal routine that handles events inside the content region of a window. TaskMaster invokes this routine if the `tmContentControls` bit of the `taskMask` field of the task record is set to 1. Your program should never issue this call.

Pseudocode

```
If tmContentControls in wmTaskMask = 1:
      If mousedown in content region of frontmost window:
            Set wmTaskData2, wmTaskData3, and wmTaskData4 to $00000000.
            Call FindControl.
            Put resulting partCode into low-order word of wmTaskData3.
            Put controlHandle into wmTaskData2.
            If partCode <> 0:
                  Call GetCtlID.
                  Put resulting control ID into wmTaskData4.
                  Call TrackControl with actionProcPtr set to $FFFFFFFF.
                  If result <> 0 or part code corresponds to scroll bar:
                        Put resulting partCode into high-order word of
                              wmTaskData3.
                        If the control is a check box or radio button:
                              Set or clear the value, as appropriate.
                        Endif.
                        Return(wInControl).
                  Endif.
                  Set low word of wmTaskData = wInControl.
                  Return (nullEvt).
            Endif.
      Else:
            Set wmTaskData = pointer to window.
            Return(wInContent).
      Endif.
Endif.
```

`TaskMasterContent` calls `FindControl`. If the user did not press the button in a control, then the routine returns a result code of `wInContent`, indicating that the mouse is in the content region of the window.

If the user did press the mouse button in a control, `TaskMasterContent` calls `TrackControl`, directing the Control Manager to use the appropriate action procedure for the control.

When `TrackControl` returns, `TaskMasterContent` examines the part code. If the part code is set to 0, then the user decided not to use the control (released the mouse button outside the control). `TaskMasterControl` returns a result code of `nullEvt` ($0000).

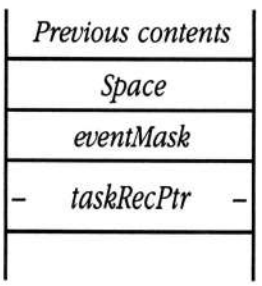
If the part code is nonzero, then the user released the mouse button within a control. `TaskMasterContent` returns a result code of `wInControl`, `wmTaskData2` contains the control handle, `wmTaskData3` (low-order word) contains the part code identifying the control in which the user pressed the mouse button, `wmTaskData3` (high-order word) contains the part code identifying the control in which the user released the mouse button, and `wmTaskData4` contains the control ID (if there is one defined).

TaskMasterDA $5F0E

This call is the TaskMaster entry point for desk accessories. Your program passes event information obtained from the Desk Manager.

Parameters

Stack before call



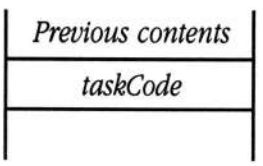
Word—Space for result

Word—Not used

Long—Pointer to extended task record

<—SP

Stack after call



Word—TaskMaster result code

<—SP

Errors None

C

```
extern pascal Word TaskMasterDA(eventMask,
          taskRecPtr);
          Pointer     taskRecPtr;

Word      eventMask;
```

TaskMasterKey $5E0E

Internal routine that handles keystroke events inside the content region of a window. Your program should never issue this call.

Pseudocode

```
If tmMenuKey in wmTaskMask = 1:
      If wmTaskData = 0:            (menu did not take keystroke):
            If tmInactive in wmTaskMask = 1:
                  If high word of wmTaskData <> 0:
                        Set low word of wmTaskData = 0.
                        Set high word of wmTaskData = ID of selected
                              inactive menu item.
                        Return (wInActMenu).
                  Endif.
                  Goto CheckControls.
            Endif.
      Else:                   (menu did take keystroke):
            If low word of wmTaskData > 255:
                  If tmControlMenu in wmTaskMask = 1:
                        Call SendEventToCtl with targetOnlyFlag = TRUE.
                        If result <> 0:
                              Set wmTaskData2 = handle of control that
                                    took keystroke.
                              Set wmTaskData3 = result code from
                                    defProc.
                              Set wmTaskData4 = ID of control that took
                                    keystroke.
                              Dim the menu title for selected menu item.
                              Set low word of wmTaskData =
                                    wInControlMenu.
                              Return (nullEvt).
                        Endif.
                        Set low word of wmTaskData = ID of selected menu
                              item.
                        Set high word of wmTaskData = ID of menu from
                              which selection was made.
                        Return (wInMenuBar).
                  Endif.
```

```
Elseif low word of wmTaskData < 250:
      If tmOpenNDA in wmTaskMask = 0:
            Set low word of wmTaskData = ID of selected menu
                  item.
            Set high word of wmTaskData = ID of menu from
                  which selection was made.
            Return (wInDeskItem).
      Endif.
      Call OpenNDA.
      Dim menu title for selected menu item.
      Set low word of wmTaskData = wInDeskItem.
      Return (nullEvt).
Elseif tmSpecial of wmTaskMask = 0:
      Set low word of wmTaskData = ID of selected menu item.
      Set high word of wmTaskData = ID of menu from which
            selection was made.
      Return (wInSpecial).
Elseif top window is an application window:
      If tmControlMenu of wmTaskMask = 1:
            Call SendEventToCtl with targetOnlyFlag = TRUE.
            If result <> 0:
                  Set wmTaskData2 = handle of control that
                        took keystroke.
                  Set wmTaskData3 = result code from
                        defProc.
                  Set wmTaskData4 = ID of control that took
                        keystroke.
                  Dim the menu title for selected menu item.
                  Set low word of wmTaskData =
                        wInControlMenu.
                  Return (nullEvt).
            Endif.
      Endif.
      Set low word of wmTaskData = ID of selected menu item.
      Set high word of wmTaskData = ID of menu from which
            item was selected.
      Return (wInSpecial).
```

```
            Elseif low word of wmTaskData = 250, 251, 252, 253, or 254:
                  Call SystemEdit.
                  If SystemEdit returns FALSE:
                        Set low word of wmTaskData = ID of selected menu
                              item.
                        Set high word of wmTaskData = ID of menu from
                              which item was selected.
                        Return (wInSpecial).
                  Endif.
                  Dim menu title for menu item that was selected.
                  Set low word of wmTaskData = wCalledSysEdit.
                  Return (nullEvt).
            Elseif low word of wmTaskData = 255:
                  Call CloseNDAbyWinPtr for top window.
                  Dim menu title for menu item that was just selected.
                  Set low word of wmTaskData = wClosedNDA.
                  Return (nullEvt).
            Endif.
      Endif.
Endif.

CheckControls:

If tmControlKey in wmTaskMask = 1:
      Set wmTaskData2, wmTaskData3, and wmTaskData4 = 0.
      If there is a front window:
            Call SendEventToCtl with targetOnlyFlag = FALSE.
            If result <> 0:
                  Set wmTaskData2 = handle of control that took the
                        keystroke.
                  Set wmTaskData3 = result from defProc.
                  Set wmTaskData4 = ID of control that took the
                        keystroke.
                  Set wmTaskData = window containing control.
                  If control is a check box or radio button:
                        Set the ctlValue for the control.
                  Endif.
                  Return (wInControl).
            Endif.
      Endif.
      Return (keyDownEvt or autoKeyEvt).
Endif.
```

`TaskMasterKey` first checks to see if menu keys are to be passed to the Menu Manager. If so, `TaskMasterKey` calls `MenuKey`. If the user entered a menu keystroke, `MenuKey` handles it, and `TaskMasterKey` returns control to the calling application.

If the user did not enter a menu key equivalent or if keystrokes are not to be passed to the Menu Manager, `TaskMasterKey` looks for a control in the active window that can receive the keystroke. If a control takes the event, `TaskMasterKey` returns `nullEvt` to the calling application. Otherwise, `TaskMasterKey` returns `keyDownEvt`, indicating that the keystroke is for the application.

`GDRPrivate` $540E

This is an internal Window Manager call; your program should never issue this call.

Error messages

Table 52-4 documents the error numbers and accompanying messages produced by the `ErrorWindow` tool call. For each error number, the following table specifies the message text displayed in the dialog box, the icon shown, and the buttons available for the user to press. Any required substitution strings are shown in the message text.

■ **Table 52-4** Error messages

Error (hex)	Message	Icon	Button
$00	`No error occurred.`	None	OK
$01	`Bad system call number.`	None	OK
$04	`Invalid parameter count.`	None	OK
$07	`GS/OS already active.`	None	OK
$10	`Device not found.`	None	OK
$11	`Invalid device number.`	None	OK
$20	`Bad request or demand.`	None	OK
$21	`Bad control or status code.`	None	OK
$22	`Bad call parameter.`	None	OK
$23	`Character device not open.`	None	OK
$24	`Character device already open.`	None	OK
$25	`Interrupt table full.`	None	OK
$26	`Resources not available.`	None	OK
$27	`I/O error.`	None	OK
$28	`Device not connected.`	None	OK
$29	`Driver is busy and not available.`	None	OK
$2B	`Device is write protected.`	None	OK
$2C	`Invalid byte count.`	None	OK
$2D	`Invalid block number.`	None	OK
$2E	`Disk has been switched.`	None	OK
$2F	`Device off-line/no media present.`	None	OK
$40	`Invalid pathname syntax.`	None	OK
$43	`Invalid reference number.`	None	OK
$44	`Subdirectory does not exist.`	None	OK
$45	`Volume not found.`	None	OK
$46	`File not found.`	None	OK

[continued]

■ **Table 52-4** Error messages [continued]

Error (hex)	Message	Icon	Button
$47	Duplicate pathname.	None	OK
$48	Volume full.	None	OK
$49	Volume directory full.	None	OK
$4A	Version error.	None	OK
$4B	Bad storage type.	None	OK
$4C	End of file encountered.	None	OK
$4D	Position out of range.	None	OK
$4E	Access not allowed.	None	OK
$4F	Buffer too small.	None	OK
$50	File is already open.	None	OK
$51	Directory error.	None	OK
$52	Unknown volume type.	None	OK
$53	Parameter out of range.	None	OK
$54	Out of memory.	None	OK
$57	Duplicate volume name.	None	OK
$58	Not a block device.	None	OK
$59	Specified level is outside legal range.	None	OK
$5A	Block number too large.	None	OK
$5B	Invalid pathnames for change_path.	None	OK
$5C	Not an executable file.	None	OK
$5D	Operating system not supported.	None	OK
$5F	Stack overflow.	None	OK
$60	Data unavailable.	None	OK
$61	End of directory has been reached.	None	OK
$62	Invalid FST call class.	None	OK
$63	File does not contain requested resource.	None	OK
$64	Specified FST is not present in system.	None	OK
$65	FST does not handle this type of call.	None	OK
$66	FST handled call, but result is weird.	None	OK

[continued]

■ **Table 52-4** Error messages [continued]

Error (hex)	Message	Icon	Button
$67	`Internal error.`	None	OK
$68	`Device list is full.`	None	OK
$69	`Supervisor list is full.`	None	OK
$70	`Cannot expand file, resource already exists.`	None	OK
$71	`Cannot add resource fork to this type of file.`	None	OK
$72	`Unknown error:` [*error string*].	None	Cancel
$80	`Error creating the new directory:` [*reason string*].	Stop	Cancel
$81	`Error saving the file:` [*reason string*].	Stop	Cancel
$82	`Insufficient access privileges to open that folder.`	Stop	OK
$83	`The selected folder cannot be opened:` [*reason string*].	Stop	Cancel
$84	`You cannot replace a folder with a file.`	Stop	Cancel
$85	`That file already exists.`	Stop	Cancel Replace
$86	`Insufficient memory to perform that operation.` `About` [*number string*]`K additional needed.`	Stop	Cancel
$87	`Initialization failed: Disk write protected.`	Stop	Cancel
$88	`The pathname is too long.`	Stop	OK
$89	`The disk is write protected.`	Caution	Cancel
$8A	`The disk is full.`	Stop	Cancel
$8B	`The disk directory is full.`	Stop	Cancel
$8C	`The file is copy-protected and can't be copied.`	Stop	Cancel
$8D	`Memory is full.`	Stop	OK

[continued]

■ **Table 52-4** Error messages [continued]

Error (hex)	Message	Icon	Button
$8E	`There isn't enough memory remaining to complete this operation. Please close some windows and try again.`	Stop	OK
$8F	`The item is locked and can't be renamed.`	Stop	Cancel
$90	`An I/O error has occurred while using the disk.`	Stop	Cancel
$91	`This disk seems to be damaged.`	Stop	Cancel
$92	`Not a ProDOS disk.`	Stop	OK
$93	`No on-line volumes can be found.`	Stop	OK
$94	`Insert the disk:` [*name string*].	Swap	Cancel

Appendix E **Resource Types**

This appendix documents the format and content of standard resources used by the Apple IIGS Toolbox. The resources are discussed in alphabetical order by resource type name. A table that lists all resources in ascending order by resource type number precedes these resource descriptions.

Resource type numbers

This appendix presents resource descriptions in order by resource type name. Often, however, you may need to determine a resource given only its resource type number. Table E-1 lists all currently defined resources in ascending order by resource type number.

■ **Table E-1** Resources listed by resource type number

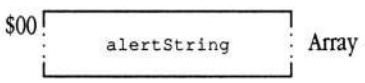
Resource type number (hex)	Resource type name	Description
$8001	rIcon	Icon specification
$8002	rPicture	QuickDraw II picture definition
$8003	rControlList	Control Manager control list
$8004	rControlTemplate	Control Manager input templates
$8005	rC1InputString	GS/OS class 1 input string
$8006	rPString	Pascal string
$8007	rStringList	Array of Pascal strings
$8008	rMenuBar	Menu bar record
$8009	rMenu	Menu template
$800A	rMenuItem	Menu item definition
$800B	rTextForLETextBox2	Data for LineEdit LETextBox2 tool call
$800D	rCtlColorTbl	Color table for control
$800E	rWindParam1	Parameters for NewWindow2
$800F	rWindParam2	Parameters for NewWindow2
$8010	rWindColor	Window Manager color table
$8011	rTextBlock	Text block
$8012	rStyleBlock	TextEdit style information
$8013	rToolStartup	Tool set startup record
$8014	rResName	Resource name
$8015	rAlertString	AlertWindow input data
$8016	rText	Unformatted text
$801A	rTwoRects	Two rectangles
$801C	rListRef	List member
$801D	rCString	C string
$8020	rErrorString	ErrorWindow input data
$8021	rKTransTable	Keystroke translation table
$8023	rC1OutputString	GS/OS class 1 output string
$8025	rTERuler	TextEdit ruler information

rAlertString $8015

Figure E-1 defines the layout of resource type `rAlertString` ($8015). Resources of this type define the data for alert windows to be displayed by the `AlertWindow` Window Manager tool call. For more complete information on alert window definitions, see Chapter 52, "Window Manager Update," earlier in this book.

`AlertWindow` accepts a reference to a string that contains its message and a reference to an array of substitution strings. The substitution strings can be any of seven standard strings (such as "OK," "Continue," and so on) or can be specified by the application and stored in the buffer to which the substitution-string pointer refers.

- **Figure E-1** Alert string, type `rAlertString` ($8015)

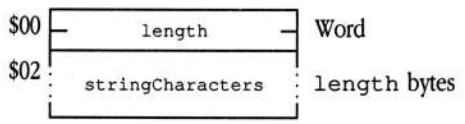


`alertString` The alert message to be displayed. Contents of this string must comply with the rules for alert window definitions documented in Chapter 52, "Window Manager Update," earlier in this book.

rC1InputString $8005

Figure E-2 defines the layout of resource type `rC1InputString` ($8005). Resources of this type contain GS/OS class 1 input strings (length word followed by data).

- **Figure E-2** GS/OS class 1 input string, type `rC1InputString` ($8005)



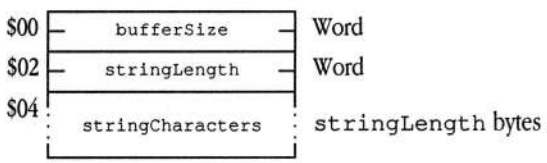
`length` The number of bytes stored at `stringCharacters`. This is an unsigned integer; valid values lie in the range from 1 to 65,535.

`stringCharacters`
Array of `length` characters.

rC1OutputString $8023

Figure E-3 defines the layout of resource type `rC1OutputString` ($8023). Resources of this type contain GS/OS class 1 output strings (buffer size word and string length word followed by data).

■ **Figure E-3** GS/OS class 1 output string, type `rC1OutputString` ($8023)



`bufferSize` The number of bytes in the entire structure, including `bufferSize`.

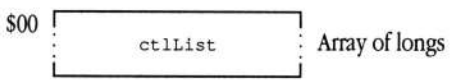
`stringLength` The number of bytes stored at `stringCharacters`. This is an unsigned integer; valid values lie in the range from 1 to 65,535. If the returned string does not fit in the buffer, this field indicates the length of the string the system wants to return. Your program should add 4 to that value (to account for `bufferSize` and `stringLength`), resize the buffer, and reissue the call.

`stringCharacters`
Array of `stringLength` characters.

rControlList $8003

Figure E-4 defines the layout of resource type `rControlList` ($8003). The Control Manager stores lists of resource IDs in resources of this type.

■ **Figure E-4** Control list, type `rControlList` ($8003)



`ctlList` List of resource IDs for control template definitions. The last entry must be set to NIL.

rControlTemplate $8004

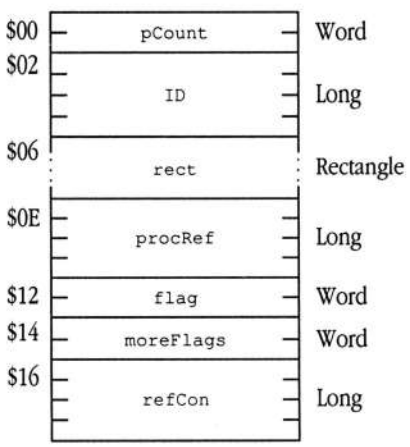
Resources of type `rControlTemplate` ($8004) define control templates, used with the Control Manager `NewControl2` tool call to create controls. You fill a type `rControlTemplate` resource according to the needs of the particular control you want to create. The system distinguishes between different control templates by examining the `procRef` field in the standard header portion that precedes each template.

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. Figure E-5 shows the format and content of the standard template header.

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 E-5** 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 on the type of control template.

`ID` Parameter that 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` Parameter that sets the `ctlRect` field of the control record for the new control. Defines the boundary rectangle for the control.

`procRef` Parameter that 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 (or a resource ID for) a control definition procedure or the ID of a standard routine. If the `fCtlProcRefNotPtr` flag in the `moreFlags` field is set to 0, then `procRef` must contain a pointer. If the flag is set to 1, then the Control Manager checks the low-order three bytes of `procRef`. If these bytes are all zero, then `procRef` must contain the ID for a standard routine; if these bytes are nonzero, `procRef` contains the resource ID for a control 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 menu
`radioControl`	$84000000	Radio button
`scrollBarControl`	$86000000	Scroll bar
`growControl`	$88000000	Size box
`statTextControl`	$81000000	Static text
`editTextControl`	$85000000	TextEdit

◆ *Note:* The `procRef` value for `iconButtonControl` is not truly a standard value but rather the resource ID of the standard control definition procedure for icon buttons.

`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`, and the low-order byte contains `ctlFlag`. The bits in `flag` are mapped as follows:

Highlight	bits 15–8	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 6–0	Values and meaning depend 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 this flag is set to 1, this control is currently the target of any typing or editing commands.
`fCtlCanBeTarget`	$4000	If this flag is set to 1, this control can be made the target control.
`fCtlWantEvents`	$2000	If this flag is set to 1, 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 the setting of this flag.)
`fCtlProcRefNotPtr`	$1000	If this flag is set to 1, then the Control Manager expects `procRef` to contain the ID or resource ID of a control procedure; if it is set to 0, then `procRef` contains a pointer to a custom control procedure.
`fCtlTellAboutSize`	$0800	If this flag is set to 1, 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 multipart controls (necessary since the Control Manager does not know about all the parts of a multipart 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 (resource type corresponds to string type)
`colorTableIsPtr`	$00	Color table reference is by pointer

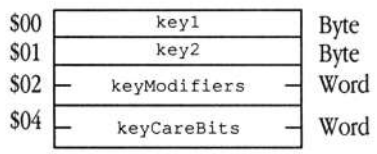
`colorTableIsHandle`	$04	Color table reference is by handle
`colorTableIsResource`	$08	Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D)

`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 E-6 shows the standard format for that keystroke information.

■ **Figure E-6** Keystroke equivalent record layout



`key1` This is the ASCII code for the uppercase or lowercase of the key equivalent.

`key2` This is the ASCII code for the uppercase or lowercase 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.

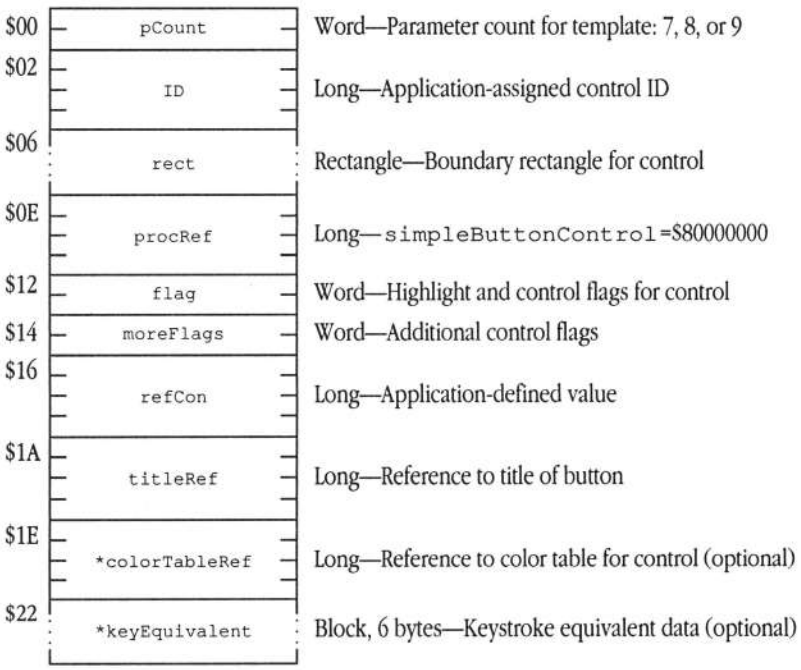
`keyModifiers` These are the modifiers that must be set to 1 for the equivalence test to pass. The format of this flag word corresponds to that defined for the event record in Chapter 7, "Event Manager," in Volume 1 of the *Toolbox Reference.* Note that only the modifiers in the high-order byte are used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

`keyCareBits` These are the modifiers that must match for the equivalence test to pass. The format of this word corresponds to that of `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 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 have the same effect, you should set the *keyPad* bit to 0.

Simple button control template

Figure E-7 shows the template that defines a simple button control.

■ **Figure E-7** Item template for simple button controls



Defined bits for `flag` are

Reserved	bits 15–8	Must be set to 0.
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–2	Must be set to 0.
Button type	bits 1–0	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

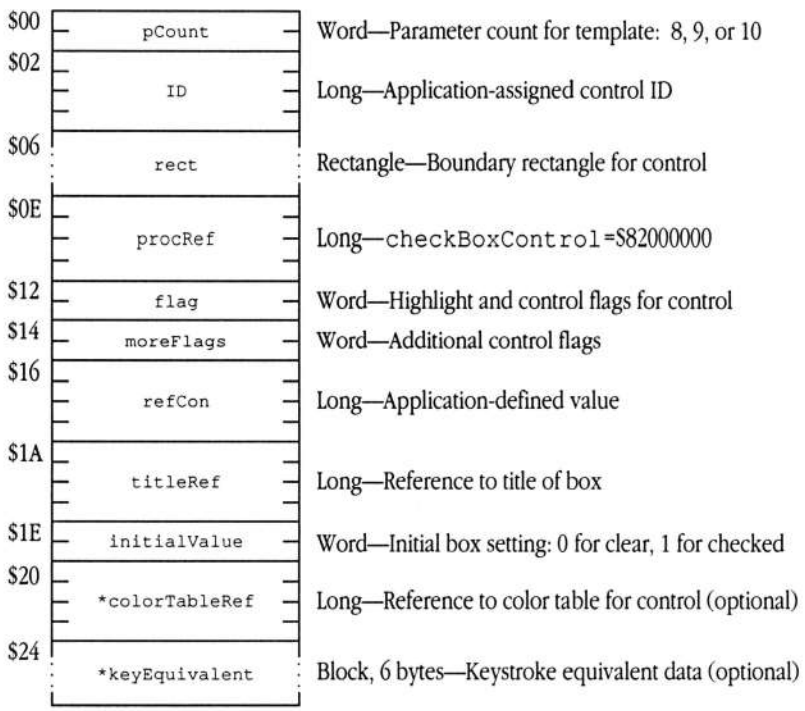
`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Set to 1 if button has keystroke equivalent.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–4	Must be set to 0.
Color table reference	bits 3–2	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 by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Title reference	bits 1–0	Defines type of title reference in `titleRef`. 00 = Title reference is by pointer 01 = Title reference is by handle 10 = Title reference is by resource ID (resource type corresponds to string type) 11 = Invalid value

`keyEquivalent` Keystroke equivalent information stored at `keyEquivalent` is formatted as shown in Figure E-6.

Check box control template

Figure E-8 shows the template that defines a check box control.

■ **Figure E-8** Control template for check box controls



Defined bits for `flag` are

Reserved	bits 15–8	Must be set to 0.
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–0	Must be set to 0.

Defined bits for `moreFlags` are

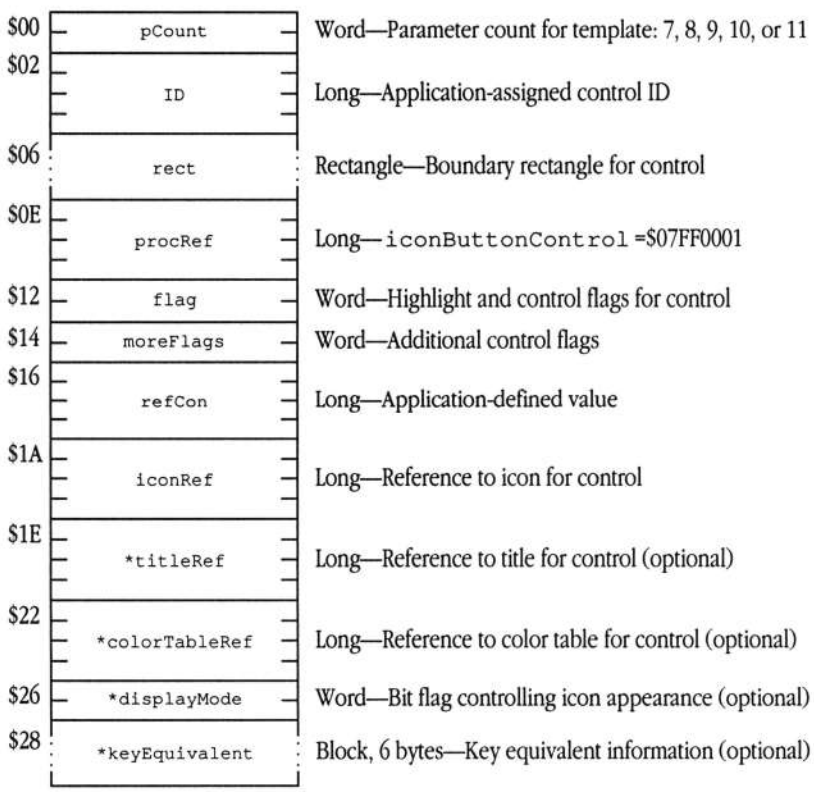
`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Set to 1 if check box has keystroke equivalent.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–4	Must be set to 0.
Color table reference	bits 3–2	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 by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Title reference	bits 1–0	Defines type of title reference in `titleRef`. 00 = Title reference is by pointer 01 = Title reference is by handle 10 = Title reference is by resource ID (resource type corresponds to string type) 11 = Invalid value

`keyEquivalent` Keystroke equivalent information stored at `keyEquivalent` is formatted as shown in Figure E-6.

Icon button control template

Figure E-9 shows the template that defines an icon button control. For more information about icon button controls, see "Icon Button Control" in Chapter 28, "Control Manager Update," in this book.

■ **Figure E-9** Control template for icon button controls



Defined bits for `flag` are

`ctlHilite`	bits 15–8	Sets the `ctlHilite` field of the control record.
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–3	Must be set to 0.
`showBorder`	bit 2	0 = Show border, 1 = No border.
`buttonType`	bits 1–0	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

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Must be set to 0.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–6	Must be set to 0.
Icon reference	bits 5–4	Defines type of icon reference in `iconRef`. 00 = Icon reference is by pointer 01 = Icon reference is by handle 10 = Icon reference is by resource ID (resource type of `rIcon`, $8001) 11 = Invalid value
Color table reference	bits 3–2	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 by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Title reference	bits 1–0	Defines type of title reference in `titleRef`. 00 = Title reference is by pointer 01 = Title reference is by handle 10 = Title reference is by resource ID (resource type of `rPString`, $8006) 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 0.

`displayMode` Passed directly to the `DrawIcon` routine, a field defining 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 15–12	Defines the background color to apply to black part of black-and-white icons.
Foreground color	bits 11–8	Defines the foreground color to apply to white part of black-and-white icons.
Reserved	bits 7–3	Must be set to 0.
`offLine`	bit 2	0 = Don't perform the AND operation on the image 1 = Perform the logical AND operation with light-gray pattern and image being copied
`openIcon`	bit 1	0 = Don't copy light-gray pattern 1 = Copy light-gray pattern instead of image
`selectedIcon`	bit 0	0 = Don't invert image 1 = Invert image before copying

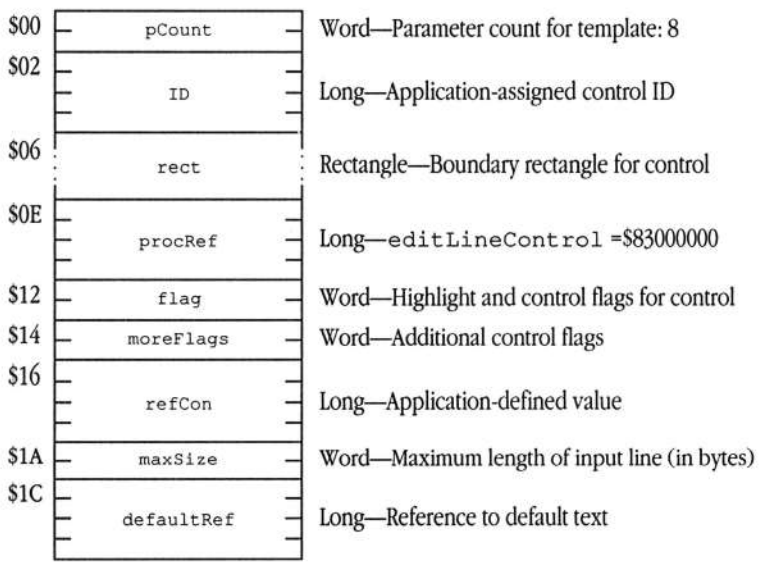
Color values (both foreground and background) are indexes into the current color table. See Chapter 16, "QuickDraw II," in Volume 2 of the *Toolbox Reference* for details about the format and content of these color tables.

`keyEquivalent` Keystroke equivalent information stored at `keyEquivalent` is formatted as shown in Figure E-6.

LineEdit control template

Figure E-10 shows the template that defines a LineEdit control. For more information about LineEdit controls, see "LineEdit Control" in Chapter 28, "Control Manager Update," in this book.

■ **Figure E-10** Control template for LineEdit controls



Defined bits for `flag` are

Reserved	bits 15–8	Must be set to 0.
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–0	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.
`fCtlWantEvents`	bit 13	Must be set to 1.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–2	Must be set to 0.
Text reference	bits 1–0	Defines type of text reference in `defaultRef`. 00 = Text reference is by pointer 01 = Text reference is by handle 10 = Text reference is by resource ID (resource type of `rPString`, $8006) 11 = Invalid value

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

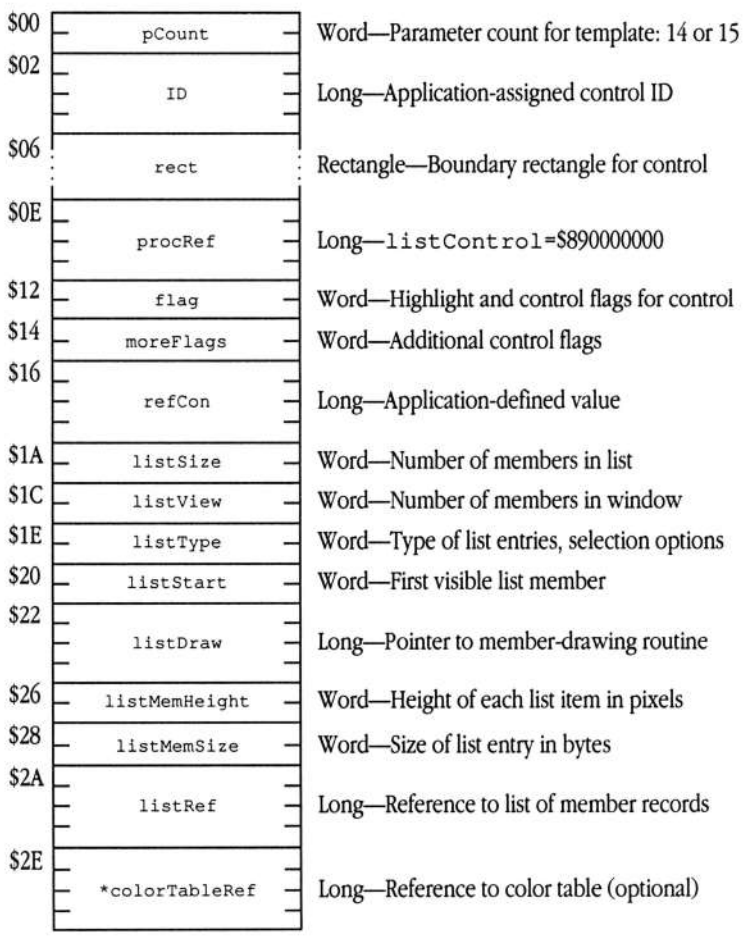
The high-order bit indicates whether the LineEdit field is a password field. Password fields protect user input by echoing asterisks rather than the actual user input. If this bit is set to 1, then the LineEdit field is a password field.

Note that LineEdit controls do not support color tables.

List control template

Figure E-11 shows the template that defines a list control. For more information about list controls, see "List Control" in Chapter 28, "Control Manager Update," in this book.

■ **Figure E-11** Control template for list controls



Defined bits for `flag` are

Reserved	bits 15–8	Must be set to 0.
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–0	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.
`fCtlWantEvents`	bit 13	Must be set to 0.
`fCtlProcRefNotPtr`	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 9–4	Must be set to 0.
Color table reference	bits 3–2	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 by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
List reference	bits 1–0	Defines type of reference in `listRef`. (The format of a list member record is described in Chapter 11, "List Manager," in Volume 1 of the *Toolbox Reference.*) 00 = List reference is by pointer 01 = List reference is by handle 10 = List reference is by resource ID (resource type of `rListRef`, $801C) 11 = Invalid value

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

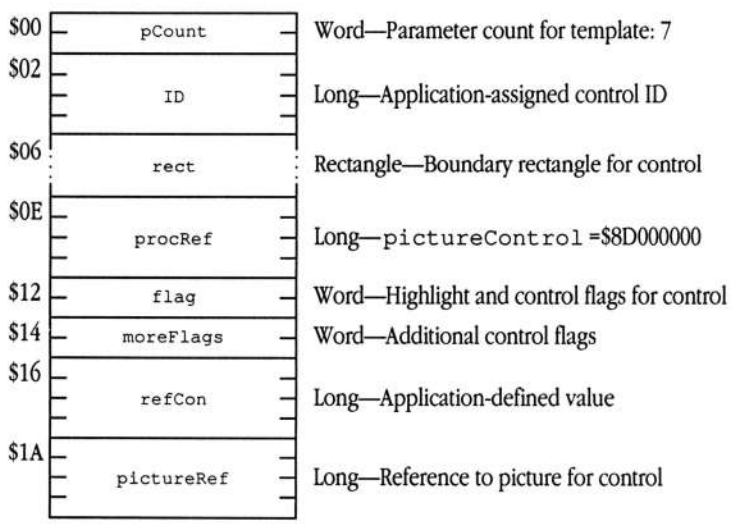
Reserved	bits 15–3	Must be set to 0.
`fListScrollBar`	bit 2	Allows you to control where the scroll bar for the list is drawn. 0 = Scroll bar drawn on outside of boundary rectangle 1 = Scroll bar drawn on inside of boundary rectangle; the List Manager calculates space needed, adjusts dimensions of boundary rectangle, and resets this flag
`fListSelect`	bit 1	Controls type of selection options available to the user. 0 = Arbitrary and range selection allowed 1 = Only single selection allowed
`fListString`	bit 0	Defines the type of strings used to define list items. 0 = Pascal strings 1 = C strings ($00-terminated)

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 E-12 shows the template that defines a picture control. For more information about picture controls, see "Picture Control" in Chapter 28, "Control Manager Update," in this book.

■ **Figure E-12** Control template for picture controls



Defined bits for `flag` are

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

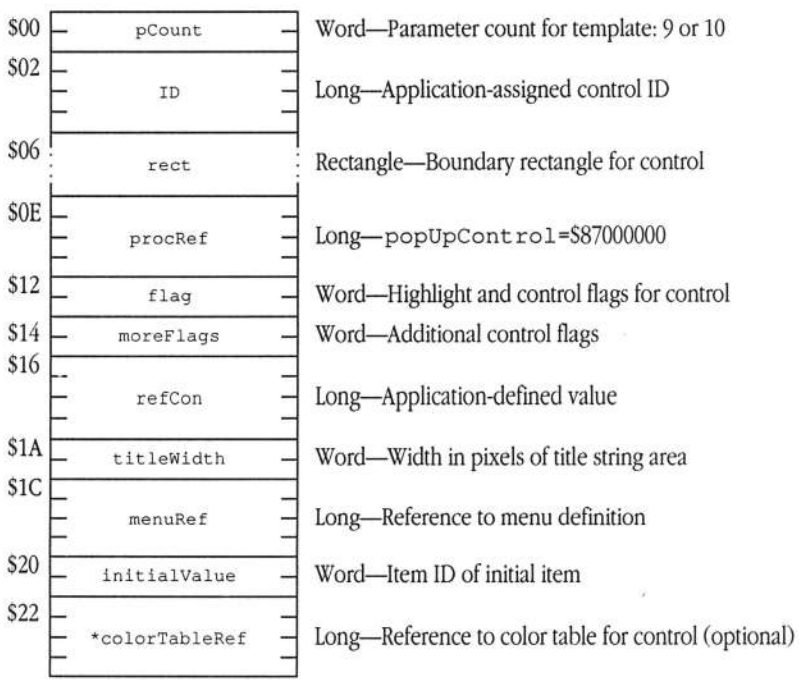
Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Must be set to 0.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–2	Must be set to 0.
Picture reference	bits 1–0	Define type of picture reference in `pictureRef`. 00 = Invalid value 01 = Reference is by handle 10 = Reference is by resource ID (resource type of `rPicture`, $8002) 11 = Invalid value

Pop-up control template

Figure E-13 shows the template that defines a pop-up control. For more information about pop-up controls, see "Pop-up Control" in Chapter 28, "Control Manager Update," in this book.

■ **Figure E-13** Control template for pop-up controls



Defined bits for `flag` are

`ctlHilite`	bits 15–8	Specifies whether the control wants to receive mouse selection events; the values for `ctlHilite` are 0 = Control is active 255 = Control is inactive
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
`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). 0 = Draw normal pop-up 1 = Draw pop-up with white space (type 2)
`fDontHiliteTitle`	bit 5	Controls highlighting of the control title. 0 = Highlight title 1 = Do not 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. 0 = Draw the title 1 = Do not draw the title
`fDontDrawResult`	bit 3	Allows you to control whether the selection is drawn in the pop-up rectangle. 0 = Draw the result 1 = Do not draw the result in the result area after a selection
`fInWindowOnly`	bit 2	Controls the extent to which the pop-up menu can be enlarged; this is particularly relevant to type 2 pop-up menus (see Chapter 37, "Menu Manager Update," for details on type 2 pop-up menus). 0 = Allow the pop-up menu to enlarge to screen size 1 = Keep the pop-up menu in the current window
`fRightJustifyTitle`	bit 1	Controls title justification. 0 = Left-justify the title 1 = Right-justify the title; note that if the title is right justified, then the control rectangle is adjusted to eliminate unneeded pixels; the value for `titleWidth` is also adjusted

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

Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Must be set to 1 if the pop-up menu has any keystroke equivalents defined.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–5	Must be set to 0.
Color table reference	bits 4–3	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 by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
`fMenuDefIsText`	bit 2	Defines type of data referred to by `menuRef`. 0 = `menuRef` is a reference to a menu template (See Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference* for details on format and content of a menu template.) 1 = `menuRef` is a pointer to a text stream in `NewMenu` format (Again, see Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference* for details.)
Menu reference	bits 1–0	Defines type of menu reference in `menuRef` (if `fMenuDefIsText` is set to 1, then these bits are ignored). 00 = Menu reference is by pointer 01 = Menu reference is by handle 10 = Menu reference is by resource ID (resource type of `rMenu`, $8009) 11 = Invalid value

`rect` The boundary rectangle for the pop-up menu and its title, before the menu is selected by the user. The Menu Manager calculates the lower-right coordinates of the rectangle for you if you specify them as (0,0).

`titleWidth` A parameter providing 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. If you are creating a series of pop-up menus and you want to align them vertically, give 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 according to the length of the menu title, and the pop-up rectangle immediately follows the title string. If the width of your title exceeds the value of `titleWidth`, results are unpredictable.

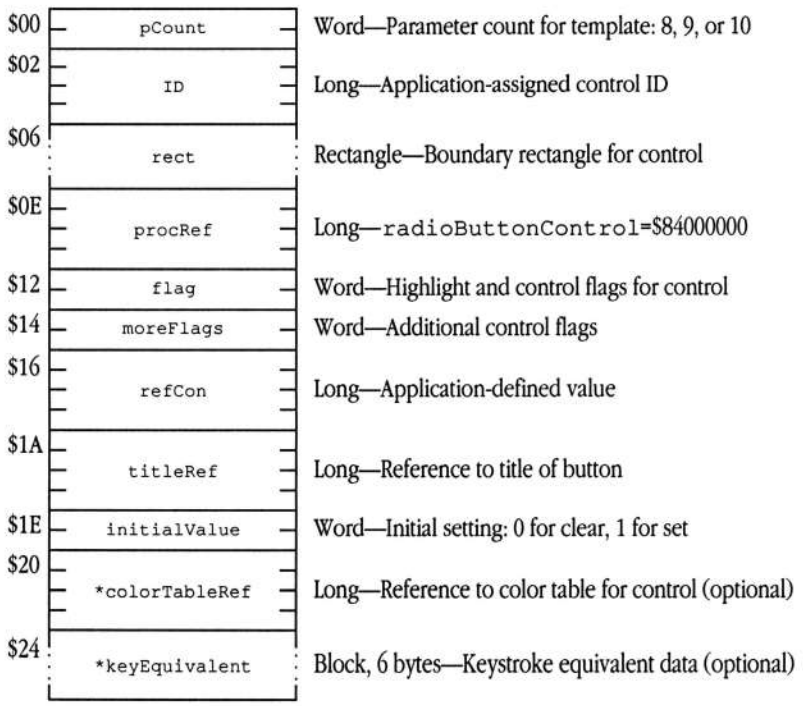
`menuRef` Reference to menu definition (see Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference* and Chapter 37, "Menu Manager Update," in this book for details on menu templates). The type of reference contained in `menuRef` is defined by the menu reference bits in `moreFlags`.

`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 unselected 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.

Radio button control template

Figure E-14 shows the template that defines a radio button control.

■ **Figure E-14** Control template for radio button controls



Defined bits for `flag` are

Reserved	bits 15–8	Must be set to 0.
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Family number	bits 6–0	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

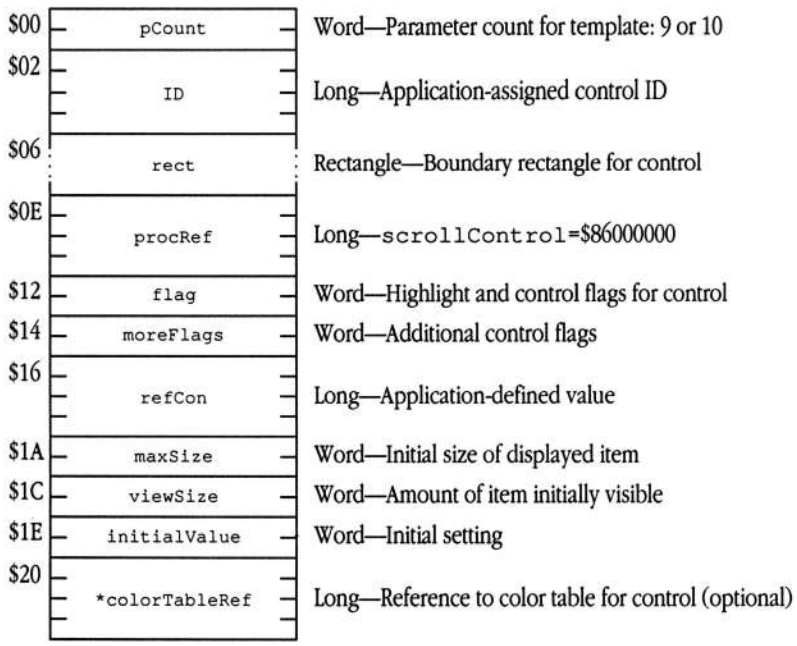
`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Set to 1 if button has keystroke equivalent.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–4	Must be set to 0.
Color table reference	bits 3–2	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 by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Title reference	bits 1–0	Defines type of title reference in `titleRef`. 00 = Title reference is by pointer 01 = Title reference is by handle 10 = Title reference is by resource ID (resource type corresponds to string type) 11 = Invalid value

`keyEquivalent` Keystroke equivalent information stored at `keyEquivalent` is formatted as shown in Figure E-6.

Scroll bar control template

Figure E-15 shows the template that defines a scroll bar control.

■ **Figure E-15** Control template for scroll bar controls



Defined bits for `flag` are

Reserved	bits15–8	Must be set to 0.
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–5	Must be set to 0.
`horScroll`	bit 4	0 = Vertical scroll bar, 1 = Horizontal scroll bar.
`rightFlag`	bit 3	0 = Bar has no right arrow, 1 = Bar has right arrow.
`leftFlag`	bit 2	0 = Bar has no left arrow, 1 = Bar has left arrow.
`downFlag`	bit 1	0 = Bar has no down arrow, 1 = Bar has down arrow.
`upFlag`	bit 0	0 = Bar has no up arrow, 1 = Bar has up arrow.

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

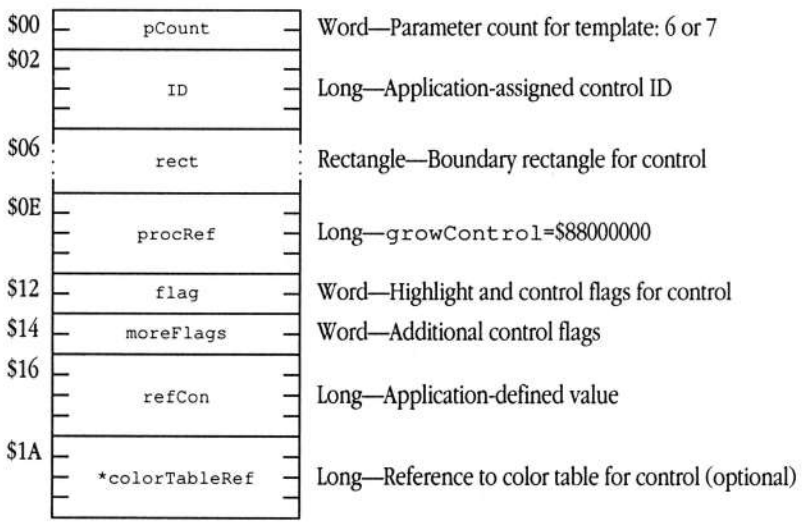
Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Must be set to 0.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–4	Must be set to 0.
Color table reference	bits 3–2	Defines type of reference in `colorTableRef`. (See Chapter 4, "Control Manager," in Volume 1 of the *Toolbox Reference* and "Clarifications" in Chapter 28, "Control Manager Update," in this book for the definition of the scroll bar color table.) 00 = Color table reference is by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Reserved	bits 1–0	Must be set to 0.

Size box control template

Figure E-16 shows the template that defines a size box control.

■ **Figure E-16** Control template for size box controls



Defined bits for `flag` are

Reserved	bits 15–8	Must be set to 0.
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–1	Must be set to 0.
`fCallWindowMgr`	bit 0	0 = Just highlight control, 1 = Call `GrowWindow` and `SizeWindow` to track this control.

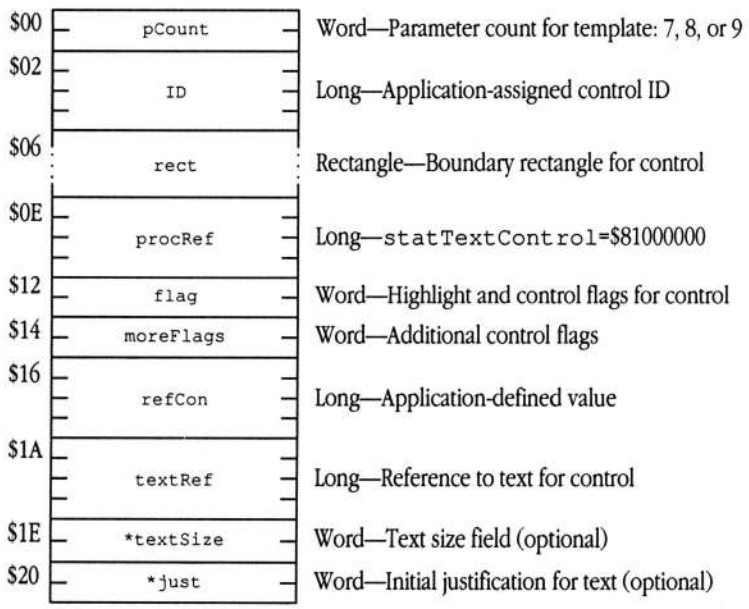
Defined bits for `moreFlags` are

`fCtlTarget`	bit 15	Must be set to 0.
`fCtlCanBeTarget`	bit 14	Must be set to 0.
`fCtlWantEvents`	bit 13	Must be set to 0.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–4	Must be set to 0.
Color table reference	bits 3–2	Defines type of reference in `colorTableRef`. (See "Error Corrections" in Chapter 28, "Control Manager Update," in this book for the definition of the size box color table.) 00 = Color table reference is by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Reserved	bits 1–0	Must be set to 0.

Static text control template

Figure E-17 shows the template that defines a static text control. For more information about static text controls, see "Static Text Control" in Chapter 28, "Control Manager Update," in this book.

■ **Figure E-17** Control template for static text controls



Defined bits for `flag` are

Reserved	bits 15–8	Must be set to 0.
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–2	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.
`fCtlWantEvents`	bit 13	Must be set to 0.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	Must be set to 0.
Reserved	bits 10–2	Must be set to 0.
Text reference	bits 1–0	Defines type of text reference in `textRef`. 00 = Text reference is by pointer 01 = Text reference is by handle 10 = Text reference is by resource ID (resource type of `rTextForLETextBox2`, $800B) 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 passed to `LETextBox2` (see Chapter 10, "LineEdit Tool Set," in Volume 1 of the *Toolbox Reference* for details on the `LETextBox2` tool call) and used to set the initial justification for the text being drawn. Valid values for `just` are

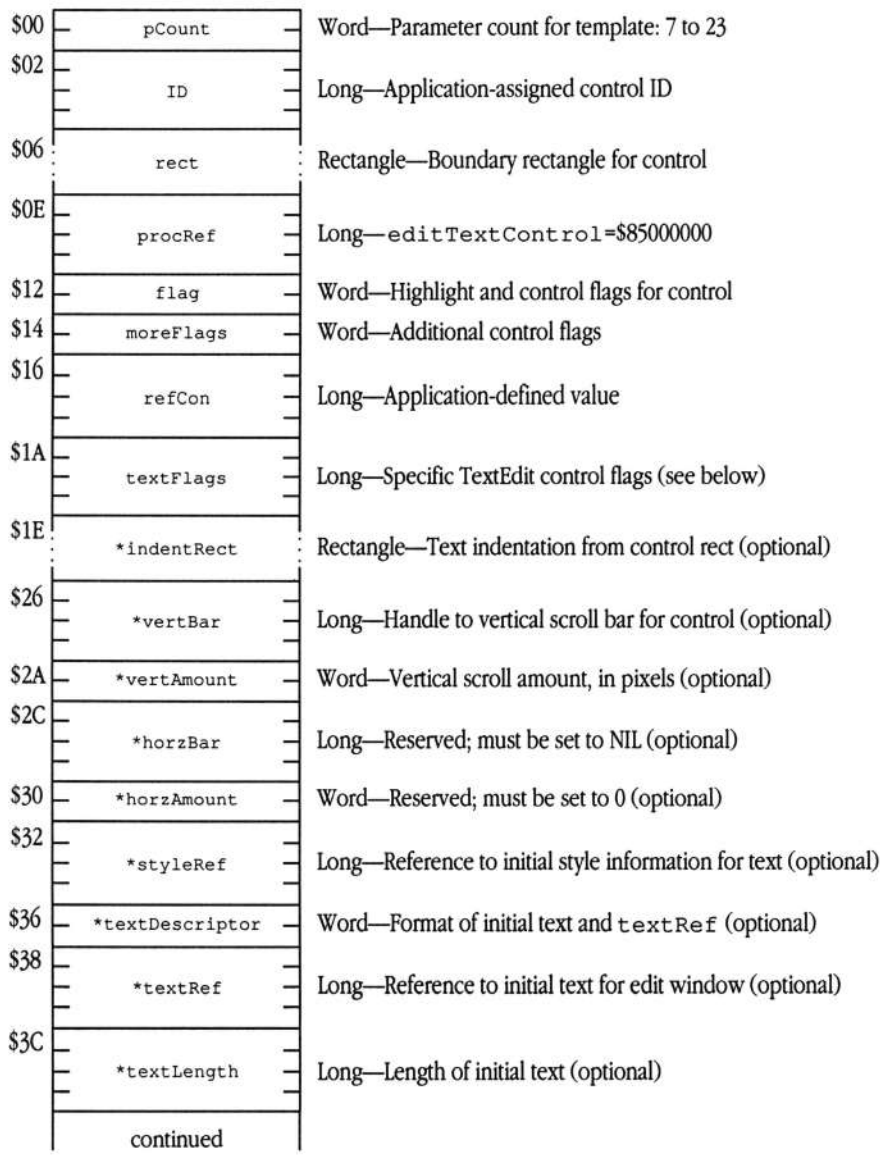
`leftJustify`	0	Text is left justified in the display window
`centerJustify`	1	Text is centered in the display window
`rightJustify`	–1	Text is right justified in the display window
`fullJustify`	2	Text is fully justified (both left and right) in the display window

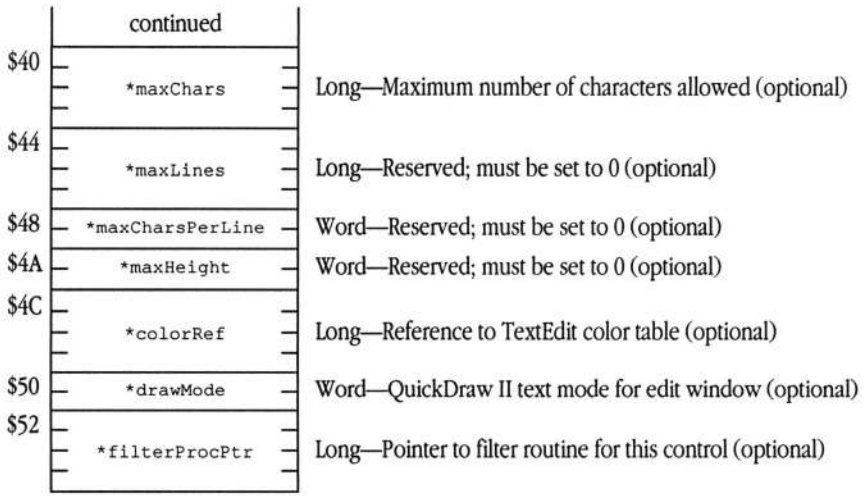
Static text controls do not support color tables. To display text of different color, you must embed the appropriate commands into the text string you are displaying. See the discussion of `LETextBox2` in Chapter 10, "LineEdit Tool Set," in Volume 1 of the *Toolbox Reference* for details on command format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

TextEdit control template

Figure E-18 shows the template that defines a TextEdit control. For more information about TextEdit controls, see "TextEdit Control" in Chapter 28, "Control Manager Update," in this book.

■ **Figure E-18** Control template for TextEdit controls





Defined bits for `flag` are

Reserved	bits 15–8	Must be set to 0.
`ctlInvis`	bit 7	0 = Visible, 1 = Invisible.
Reserved	bits 6–0	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.
`fCtlWantEvents`	bit 13	Must be set to 1.
`fCtlProcRefNotPtr`	bit 12	Must be set to 1.
`fCtlTellAboutSize`	bit 11	If this bit is set to 1, a size box is created in the lower-right corner of the window. Whenever the control window is resized, the edit text is resized and redrawn.
`fCtlIsMultiPart`	bit 10	Must be set to 1.
Reserved	bits 9–4	Must be set to 0.

Color table reference	bits 3–2	Defines type of reference in `colorRef`. (The color table for a TextEdit control [`TEColorTable`] is described in Chapter 49, "TextEdit Tool Set," in this book.) 00 = Color table reference is by pointer 01 = Color table reference is by handle 10 = Color table reference is by resource ID (resource type of `rCtlColorTbl`, $800D) 11 = Invalid value
Style reference	bits 1–0	Defines type of style reference in `styleRef`; the format for a TextEdit style descriptor is described in Chapter 49, "TextEdit Tool Set," in this book. 00 = Style reference is by pointer 01 = Style reference is by handle 10 = Style reference is by resource ID 11 = Invalid value

△ **Important** Do not set `fCtlTellAboutSize` to 1 unless the TextEdit record also has a vertical scroll bar. This flag works only for TextEdit records that are controls. △

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. 0 = Do not restrict the number of styles in the text 1 = Allow only one style in the text
`fNoWordWrap`	bit 28	Allows you to control TextEdit word wrap behavior. 0 = Perform word wrap to fit the ruler 1 = Do not word wrap the text; break lines only on return ($0D) characters
`fNoScroll`	bit 27	Controls user access to scrolling. 0 = Allow scrolling 1 = Do not allow either manual or automatic scrolling
`fReadOnly`	bit 26	Restricts the text in the window to read-only operations (copying from the window is still allowed). 0 = Allow editing 1 = Do not allow editing

`fSmartCutPaste`	bit 25	Controls TextEdit support for smart cut and paste. (See Chapter 49, "TextEdit Tool Set," for details on smart cut and paste support.) 0 = Do not use smart cut and paste 1 = Use smart cut and paste
`fTabSwitch`	bit 24	Defines behavior of the Tab key. (See Chapter 49, "TextEdit Tool Set," for details.) 0 = Tab inserted in TextEdit document 1 = Tab to next control in the window
`fDrawBounds`	bit 23	Tells TextEdit whether to draw a box around the edit window, just inside `rect`; the pen for this box is 2 pixels wide and 1 pixel high. 0 = Do not draw rectangle 1 = Draw rectangle
`fColorHilight`	bit 22	Must be set to 0.
`fGrowRuler`	bit 21	Tells TextEdit whether to resize the ruler in response to the resizing of the edit window by the user. If this bit is set to 1, TextEdit automatically adjusts the right margin value for the ruler. 0 = Do not resize the ruler 1 = Resize the ruler
`fDisableSelection`	bit 20	Controls whether user can select text. 0 = User can select text 1 = User cannot select text
`fDrawInactiveSelection`	bit 19	Controls how inactive selected text is displayed. 0 = TextEdit does not display inactive selections 1 = TextEdit draws a box around inactive selections
Reserved	bits 18–0	Must be set to 0.

`indentRect` A rectangle whose coordinates specify the amount, in pixels, of white space to leave between the boundary rectangle for the control and the text itself. 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. 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, just inside the right edge of the boundary rectangle for the control, then set this field to $FFFFFFFF.

`vertAmount` The number of pixels to scroll whenever the user presses the up or down arrow on the vertical scroll bar. 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 Tool Set," 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 NIL.

`textDescriptor` Input text descriptor that defines the reference type for the initial text (which is defined in the `textRef` field) and the format of that text. See Chapter 49, "TextEdit Tool Set," 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 NIL.

`textLength` The length of the initial text. 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 NIL.

`maxLines` Must be set to 0.

`maxCharsPerLine` Must be set to NIL.

`maxHeight` Must be set to 0.

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

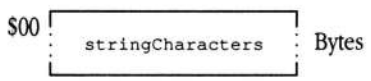
`drawMode` The text mode used by QuickDraw II for drawing text. See Chapter 16, "QuickDraw II," in Volume 2 of the *Toolbox Reference* for details on valid text modes.

`filterProcPtr` Pointer to a filter routine for the control. See Chapter 49, "TextEdit Tool Set," 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.

rCString $801D

Figure E-19 defines the layout of resource type `rCString` ($801D). Resources of this type contain C strings (null-terminated character arrays).

■ **Figure E-19** C string, type `rCString` ($801D)



`stringCharacters`
Array of characters; last character must be a null terminator ($00). The string may contain up to 65,535 characters, including the null terminator.

rCtlColorTbl $800D

Resources of this type store color tables for various tool sets. These resources do not have a consistent internal layout; you must construct these resources according to the needs of the tool set that is to use the color table.

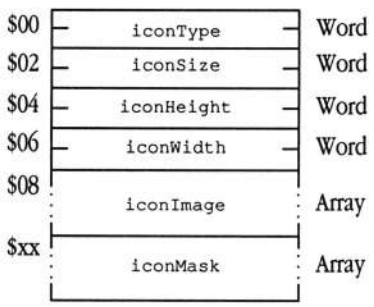
rErrorString $8020

Resources of this type define the data that appears in error windows displayed by the `ErrorWindow` Window Manager tool call. The layout of `rErrorString` resources is the same as that of `rAlertString` resources, which in turn correspond to the strings that define alert windows. For more complete information on alert string definitions, see Chapter 52, "Window Manager Update," in this book.

rIcon $8001

Figure E-20 defines the layout of resource type `rIcon` ($8001).

■ **Figure E-20** Icon, type `rIcon` ($8001)

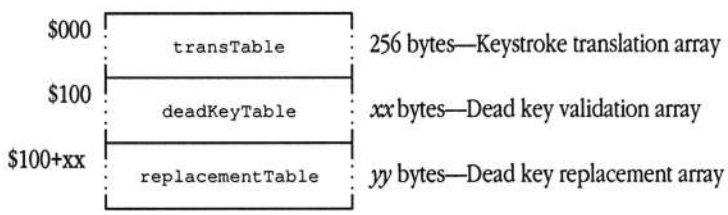


`iconType`	Flags defining the type of icon stored in the icon record.	
Color indicator	bit 15	Indicates whether the icon contains a color or black-and-white image. 0 = Icon is black and white 1 = Icon is color
`iconSize`	The size, in bytes, of the icon image stored at `iconImage`.	
`iconHeight`	The height, in pixels, of the icon.	
`iconWidth`	The width, in pixels, of the icon.	
`iconImage`	`iconSize` bytes of icon image data.	
`iconMask`	`iconSize` bytes of mask data to be applied to the image located at `iconImage`.	

rKTransTable $8021

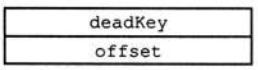
Figure E-21 defines the layout of resource type `rKTransTable` ($8021). Resources of this type define keystroke translation tables for use by the Event Manager (see Chapter 31, "Event Manager Update," in this book for complete information on the format and content of resources of this type).

■ **Figure E-21** Keystroke translation table, type `rKTransTable` ($8021)



`transTable` 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 corresponds directly 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 carriage return (CR). Cells 128 to 255 ($80 to $FF) of the `transTable` contain values for Option-key sequences (such as Option-S).

`deadKeyTable` Table containing entries used to validate dead keys. Dead keys are keystrokes used to introduce multikey sequences that produce single characters. For example, pressing Option-U followed by e yields ë. 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:

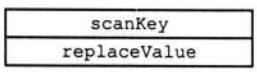


Byte—Character code for dead key
Byte—Offset from `deadKeyTable` into `replacementTable`

`deadKey` 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 the 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`
Table containing 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 subarray must be set to $0000. Each entry in the `replacementTable` must be formatted as follows:



Byte—Character code for dead key combination
Byte—Result character code for dead key combination

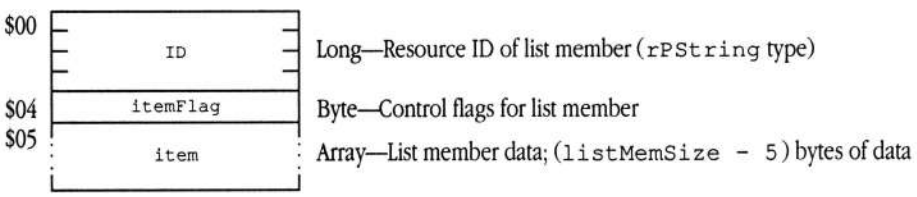
`scanKey` A valid character code for dead key replacement. The system uses this field to determine whether the user entered a valid dead key combination. The system compares this value with the second user keystroke.

`replaceValue` 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.

rListRef $801C

Figure E-22 defines the layout of the array element that composes resource type `rListRef` ($801C). Resources of this type define members of list controls (see Chapter 28, "Control Manager Update," in this book for more information on list controls). A single `rListRef` resource may contain more than one of these elements; you concatenate the elements to form the resource.

■ **Figure E-22** List member reference array element, type `rListRef` ($801C)



`ID` Resource ID of the list member (resource type of `rPString`, $8006).

`itemFlag` Control flags for the member.

`memSelect` bits 7–6 Indicates whether the item is selected.
00 = Item is enabled but not selected
01 = Item is disabled (cannot be selected)
10 = Item is selected
11 = Invalid value

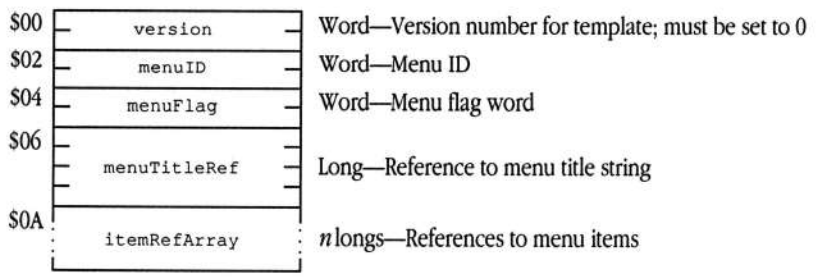
Reserved bits 5–0 Must be set to 0.

`item` Application-specific data for the list member. The `listMemSize` field of the list control template specifies the size of this field, plus 5. For example, to assign a 2-byte tag to each list member, you would set `listMemSize` to 7 (2+5) and place the tag value at `item` in each list member.

rMenu $8009

Figure E-23 defines the layout of resource type `rMenu` ($8009). Resources of this type define parameters to some new Menu Manager tool calls. See Chapter 37, "Menu Manager Update," in this book for more information.

■ **Figure E-23** Menu template, type `rMenu` ($8009)



`version` 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 Volume 1 of the *Toolbox Reference* for information on valid values for `menuID`.

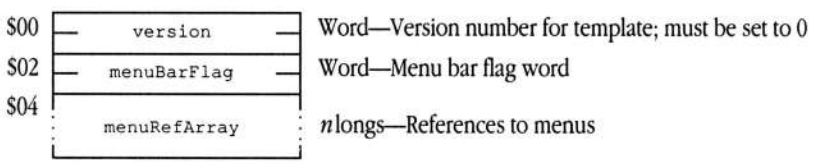
`menuFlag`		Bit flags controlling the display and processing attributes of the menu. Valid values for `menuFlag` are
`titleRefType`	bits 15–14	Defines the type of reference in `menuTitleRef`. 00 = Reference is by pointer 01 = Reference is by handle 10 = Reference is by resource ID 11 = Invalid value
`itemRefType`	bits 13–12	Defines the type of reference in each entry of `itemRefArray` (all array entries must be of the same type). 00 = Reference is by pointer 01 = Reference is by handle 10 = Reference is by resource ID 11 = Invalid value
Reserved	bits 11–9	Must be set to 0.
`alwaysCallmChoose`	bit 8	Causes the Menu Manager to call a custom menu defProc `mChoose` routine even when the pointer 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 item 1 = Use XOR to highlight item
`custom`	bit 4	Indicates whether menu is custom or standard. 0 = Standard menu 1 = Custom menu
`allowCache`	bit 3	Controls menu caching. 0 = No menu caching allowed 1 = Menu caching allowed
Reserved	bits 2–0	Must be set to 0.
`menuTitleRef`		Reference to title string of 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 items in 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 be of the same reference type. The last entry in the array must be set to $00000000.

rMenuBar $8008

Figure E-24 defines the layout of resource type `rMenuBar` ($8008). Resources of this type define the characteristics of a menu bar for new Menu Manager tool calls. For more information, see Chapter 37, "Menu Manager Update," in this book.

■ **Figure E-24** Menu bar record, type `rMenuBar` ($8008)



`version` 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 15–14 Defines the type of reference in each entry of `menuRefArray` (all array entries must be of the same type).
00 = Reference is by pointer
01 = Reference is by handle
10 = Reference is by resource ID
11 = Invalid value

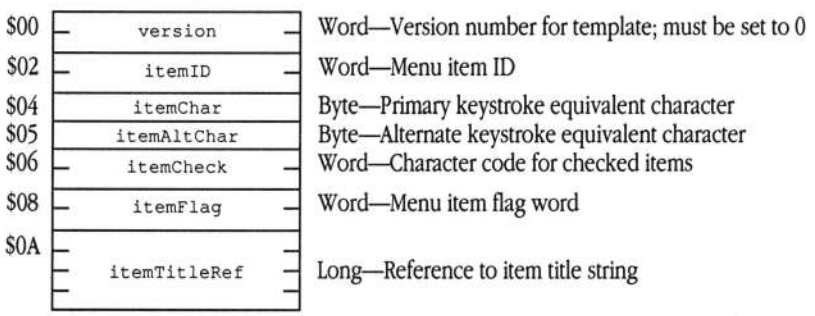
Reserved bits 13–0 Must be set to 0.

`menuRefArray` Array of references to the menus in 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 be of the same reference type. The last entry in the array must be set to $00000000.

rMenuItem $800A

Figure E-25 defines the layout of resource type `rMenuItem` ($800A). Resources of this type define menu items to some new Menu Manager tool calls. See Chapter 37, "Menu Manager Update," in this book for more information.

■ **Figure E-25** Menu item template, type `rMenuItem` ($800A)



`version` 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 Volume 1 of the *Toolbox Reference* for information on valid values for `itemID`.

`itemChar`, `itemAltChar`
Fields defining 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 uppercase and lowercase ASCII codes for a particular character. If you have only a single key equivalence, set both fields to that value.

`itemCheck` The character to be displayed next to the item when it is checked.

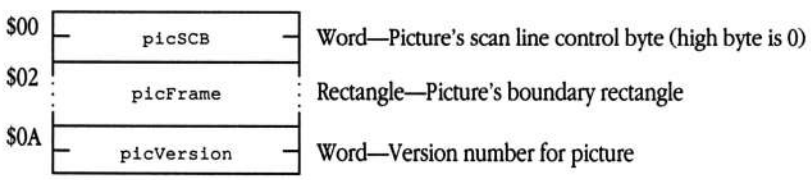
`itemFlag`	Bit flags controlling the display attributes of the menu item. Valid values for `itemFlag` are	
`titleRefType`	bits 15–14	Defines the type of reference in `itemTitleRef`. 00 = Reference is by pointer 01 = Reference is by handle 10 = Reference is by resource ID 11 = Invalid value
Reserved	bit 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 10–8	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 of a divider bar below item. 0 = No divider bar 1 = Divider bar
`XOR`	bit 5	Controls how highlighting is performed. 0 = Do not use XOR to highlight item 1 = Use XOR to highlight item
Reserved	bits 4–3	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 in boldface. 0 = Not bold 1 = Bold

`itemTitleRef` Reference to title string of 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.

rPicture $8002

Resources of this type store QuickDraw picture definitions. QuickDraw pictures are described by a series of QuickDraw operation codes specifying the commands that created the picture. When these pictures are stored as data structures, the actual picture data (the operation codes) is preceded by control information, some of which may be of interest to Apple IIGS developers. Figure E-26 shows some of this control information. Note that the layout of this control information is subject to change.

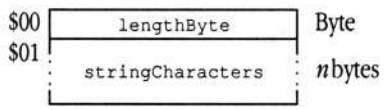
■ **Figure E-26** QuickDraw picture, type `rPicture` ($8002)



`rPString` $8006

Figure E-27 defines the layout of resource type `rPString` ($8006). Resources of this type contain Pascal strings.

■ **Figure E-27** Pascal string, type `rPString` ($8006)



`lengthByte` Number of bytes of data stored in `stringCharacters` array.

`stringCharacters`
Array of `lengthByte` characters.

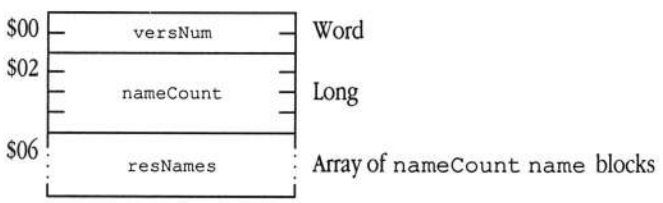
rResName $8014

Figure E-28 defines the layout of resource type `rResName` ($8014). Resources of this type define name strings for resources of a given type and ID. The resource ID value assigned to an `rResName` resource must be of the form

$0001*xxxx*

where *xxxx* corresponds to the resource type of resources whose names are defined in this resource. Within the `rResName` resource you define name strings corresponding to resources with specified resource IDs. Names are stored in Pascal strings, are not case-sensitive, and must be unique within the appropriate resource type. Resource names are not required, so you may specify names for only a few resources within a given type.

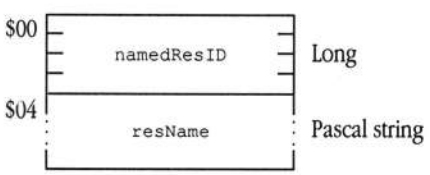
■ **Figure E-28** Resource name array, type `rResName` ($8014)



`versNum` The resource template version. Must be set to 1.

`nameCount` Count of entries in the `resNames` name-definition array.

`resNames` Array of name strings. Each entry must be formatted as follows:



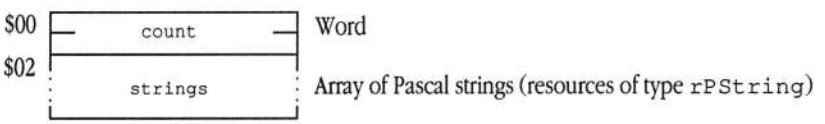
`namedResID` ID of the resource for this name.

`resName` Name string of the resource.

rStringList $8007

Figure E-29 defines the layout of resource type `rStringList` ($8007). Resources of this type contain an array of Pascal strings.

■ **Figure E-29** Pascal string array, type `rStringList` ($8007)



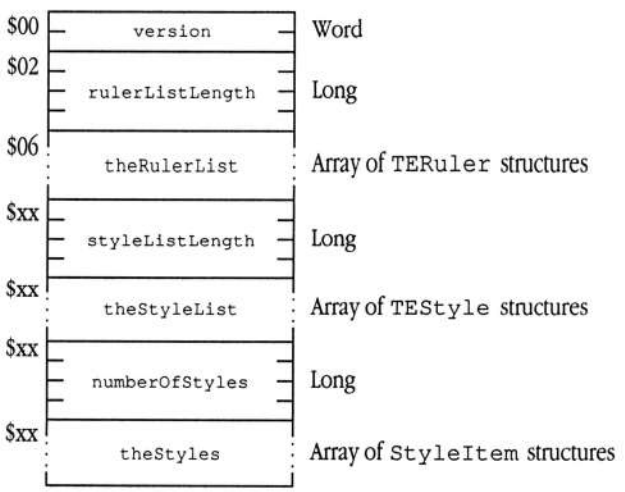
`count` The number of Pascal strings stored at `strings`.

`strings` An array of `count` Pascal strings.

rStyleBlock $8012

Figure E-30 defines the layout of resource type `rStyleBlock` ($8012). Resources of this type contain TextEdit `TEFormat` structures, which store TextEdit style information.

■ **Figure E-30** TextEdit style information, type `rStyleBlock` ($8012)



`version` Version number corresponding to the layout of this `TEFormat` structure. The number of this version of the structure is $0000.

`rulerListLength`
The length, in bytes, of `theRulerList`.

`theRulerList` Ruler data for the text record. The `TERuler` structure is embedded in the `TEFormat` structure at this location.

`styleListLength`
The length, in bytes, of `theStyleList`.

`theStyleList` List of all unique styles for the text record. The `TEStyle` structures are embedded in the `TEFormat` structure at this location. Each `TEStyle` structure must define a unique style—there must be no duplicate style entries. 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.

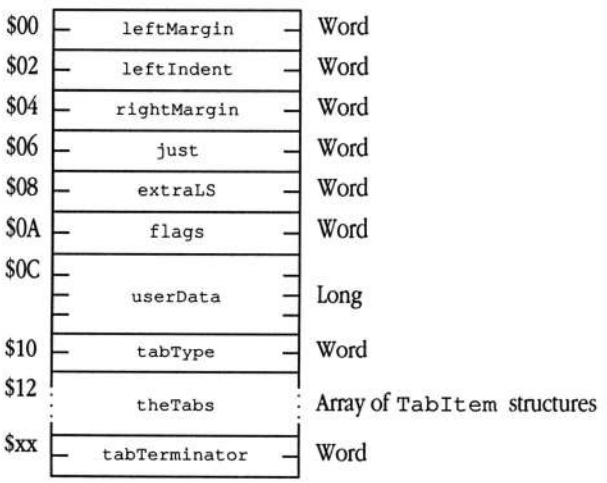
`numberOfStyles`
: The number of `StyleItems` contained in `theStyles`.

`theStyles`
: Array of `StyleItems` specifying which actual styles (stored in `theStyleList`) apply to which text within the TextEdit record.

rTERuler $8025

Figure E-31 defines the layout of resource type `rTERuler` ($8025). Resources of this type contain TextEdit `TERuler` structures, which store TextEdit ruler information.

■ **Figure E-31** TextEdit ruler information, type `rTERuler` ($8025)



`leftMargin` 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` The number of pixels to indent from the left edge of the text rectangle for text lines that start paragraphs.

`rightMargin` Maximum line length, expressed as the number of pixels from the left edge of the text rectangle.

`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 justify the text fully

`extraLS` Line spacing, expressed as the number of pixels to add between lines of text. Negative values result in text overlap.

`flags` Reserved.

`userData` Application-specific data.

`tabType` The type of tab data, specified as 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`

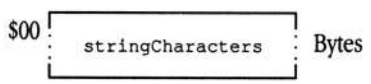
`theTabs` If `tabType` is set to 2, this is an array of `TabItem` structures defining the absolute pixel positions for the various tab stops. The `tabTerminator` field, with a value of $FFFF, marks the end of this array. For other values of `tabType`, this field is omitted from the structure.

`tabTerminator` If `tabType` is set to 0, 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, `tabTerminator` is set to $FFFF and marks the end of `theTabs` array.

rText $8016

Figure E-32 defines the layout of resource type `rText` ($8016). Resources of this type contain text blocks (data arrays with no embedded length information; block length must be indicated in other fields).

■ **Figure E-32** Text block, type `rText` ($8016)

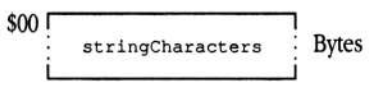


`stringCharacters`
Array of up to 65,535 characters. Any length information is contained in a separately maintained field.

rTextBlock $8011

Figure E-33 defines the layout of resource type `rTextBlock` ($8011). Resources of this type contain text blocks (data arrays with no embedded length information; block length must be indicated in other fields).

■ **Figure E-33** Text block, type `rTextBlock` ($8011)

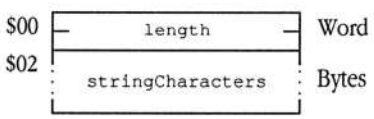


`stringCharacters`
Array of up to 65,535 characters. Any length information is contained in a separately maintained field.

rTextForLETextBox2 $800B

Figure E-34 defines the layout of resource type `rTextForLETextBox2` ($800B). Resources of this type contain data formatted as input to the `LETextBox2` LineEdit tool call (see Chapter 10, "LineEdit Tool Set," in Volume 1 of the *Toolbox Reference* for details).

■ **Figure E-34** `LETextBox2` input text, type `rTextForLETextBox2` ($800B)



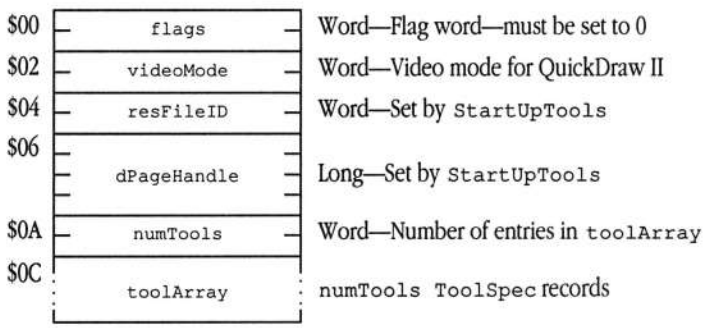
`length` The number of bytes stored at `stringCharacters`. Valid values lie in the range from 1 to 32,767.

`stringCharacters`
Array of up to 32,767 characters. Formatting information is embedded in the character array and is included in the value of `length`. See Chapter 10, "LineEdit Tool Set," in Volume 1 of the *Toolbox Reference* for complete information on the syntax of this embedded information.

rToolStartup $8013

Figure E-35 defines the layout of resource type `rToolStartup` ($8013). Resources of this type define tool set startup records for use with the Tool Locator `StartUpTools` and `ShutDownTools` tool calls (see Chapter 51, "Tool Locator Update," in this book for more information).

■ **Figure E-35** Tool set start-stop record, type `rToolStartup` ($8013)

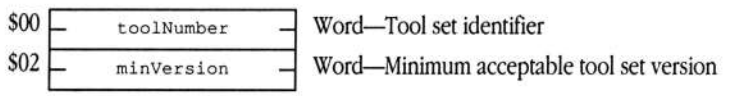


`videoMode` Defines the *masterSCB* for QuickDraw II. See Chapter 43, "QuickDraw II Update," in this book for valid values.

`resFileID` The `StartUpTools` call sets this field, which `ShutDownTools` requires as input.

`dPageHandle` The `StartUpTools` call sets this field, which `ShutDownTools` requires 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:



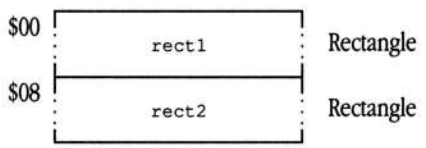
`toolNumber` The tool set to be loaded. Valid tool set numbers are discussed in Chapter 51, "Tool Locator Update," in this book.

`minVersion` The minimum acceptable version for the tool set. See Chapter 24, "Tool Locator," in Volume 2 of the *Toolbox Reference* for the format of this field.

`rTwoRects` $801A

Figure E-36 defines the layout of resource type `rTwoRects` ($801A).

■ **Figure E-36** Two rectangles, type `rTwoRects` ($801A)



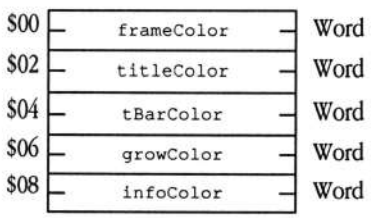
`rect1` First rectangle.

`rect2` Second rectangle.

rWindColor $8010

Figure E-37 defines the layout of resource type `rWindColor` ($8010). Resources of this type define window color tables for the Window Manager.

■ **Figure E-37** Window color table, type `rWindColor` ($8010)



`frameColor` Color of the window frame and the alert frame.

Reserved	bits 15–8	Must be set to 0.
`windowFrame`	bits 7–4	Color of window frame—value is an index into the active color table.
Reserved	bits 3–0	Must be set to 0.

`titleColor` Colors of inactive title bar, inactive title, and active title.

Reserved	bits 15–12	Must be set to 0.
`inactiveTitleBar`	bits 11–8	Color of inactive title bars—value is an index into the active color table.
`inactiveTitle`	bits 7–4	Color of inactive titles—value is an index into the active color table.
`activeTitle`	bits 3–0	Color of active titles, close box, and zoom box—value is an index into the active color table.

`tBarColor` Color and pattern information for active title bar.

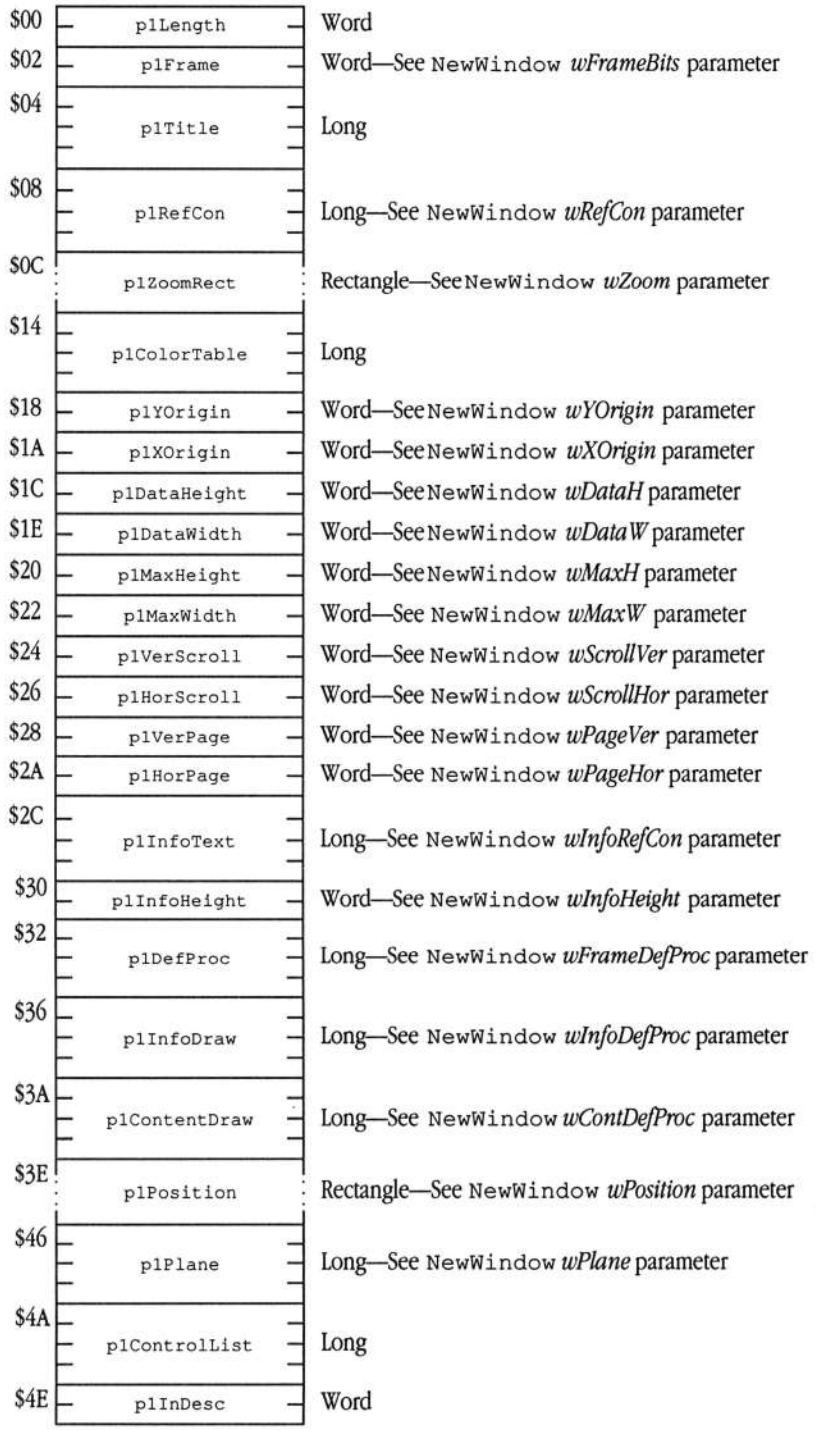
`pattern`	bits 15–8	Defines pattern of title bar. 00 = Solid 01 = Dithered 02 = Lined
`patternColor`	bits 7–4	Color of pattern—value is an index into the active color table.
`backColor`	bits 3–0	Background color—value is an index into the active color table.

`growColor`	Color of size box and middle outline of alert frame.	
`alertMidFrame`	bits 15–12	Color of middle outline of alert frame—value is an index into the active color table.
Reserved	bits 11–8	Must be set to 0.
`sizeUnselected`	bits 7–4	Color of unselected size box—value is an index into the active color table.
`sizeSelected`	bits 3–0	Color of selected size box—value is an index into the active color table.
`infoColor`	Color of information bar and inside outline of alert frame.	
`alertMidFrame`	bits 15–12	Color of inside outline of alert frame—value is an index into the active color table.
Reserved	bits 11–8	Must be set to 0.
`infoBar`	bits 7–4	Color of information bar—value is an index into the active color table.
Reserved	bits 3–0	Must be set to 0.

rWindParam1 $800E

Figure E-38 defines the layout of resource type `rWindParam1` ($800E). This resource defines a template used to create windows with the `NewWindow2` Window Manager tool call (see Chapter 52, "Window Manager Update," in this book). Most of these fields correspond to fields in the `NewWindow` parameter list (defined in Chapter 25, "Window Manager," in Volume 2 of the *Toolbox Reference*).

■ **Figure E-38** Window template, type `rWindParam1` ($800E)



`p1Length` The number of bytes in the template, including the length of `p1Length`. Must be set to $50.

`p1Title` Reference to title string for the window. The contents of `p1InDesc` specify the type of reference stored here. The title must be stored in a Pascal string containing both a leading and a trailing space.

If `p1Title` is set to NIL, the Window Manager creates a window without a title bar. If your program is creating a window with a title bar, you must specify a title of some sort. To create a window without a title, make `p1Title` (or *titlePtr* on the `NewWindow2` call) refer to a null string.

Note that the Window Manager creates a copy of the title string, allowing your program to free the memory occupied by this string after the `NewWindow2` call is issued.

If you specify a non-NIL value for *titlePtr* on the `NewWindow2` call, this field is ignored.

`p1ColorTable` Reference to the color table for the window. The contents of `p1InDesc` specify the type of reference stored here. If `p1ColorTable` is set to NIL, the Window Manager assumes that there is no color table for the window.

The format of the color table is defined in Chapter 25, "Window Manager," in Volume 2 of the *Toolbox Reference*. If `p1ColorTable` refers to a resource, then the color table must be defined in a resource of type `rWindColor`.

`p1ControlList` Reference to the template or templates defining controls for the window. The Window Manager passes this value to the `NewControl2` Control Manager tool call as the *reference* parameter. Note that `p1InDesc` contains the data for the `NewControl2` *referenceDesc* parameter. Refer to Chapter 28, "Control Manager Update," in this book for more information about `NewControl2`.

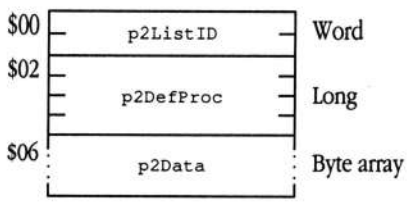
If this field is set to NIL, then the Window Manager assumes that there is no control list for the window and does not call `NewControl2`.

`p1InDesc`		The type of reference stored in `p1ColorTable` and `p1Title`. This field also contains the *referenceDesc* value for `NewControl2` that defines the contents of `p1ControlList`.
Reserved	bits 15–12	Must be set to 0.
`colorTableRef`	bits 11–10	Defines the type of reference stored in `p1ColorTable`. 00 = Reference is by pointer to color table 01 = Reference is by handle to color table 10 = Reference is by resource ID of `rWindColor` resource 11 = Invalid value
`titleRef`	bits 9–8	Defines the type of reference stored in `p1Title`. 00 = Reference is by pointer to Pascal string 01 = Reference is by handle to Pascal string 10 = Reference is by resource ID of `rPString` resource 11 = Invalid value
`controlRef`	bits 7–0	Defines the type of reference stored in `p1ControlList`; passed directly to the `NewControl2` Control Manager tool call as the *referenceDesc* parameter. (For valid values, see the description of the `NewControl2` tool call in Chapter 28, "Control Manager Update," earlier in this book.)

rWindParam2 $800F

Figure E-39 defines the layout of resource type `rWindParam2` ($800F). This resource defines a template used to create windows with the `NewWindow2` Window Manager tool call (see Chapter 52, "Window Manager Update," in this book). Use this template for custom windows.

■ **Figure E-39** Window template, type `rWindParam2` ($800F)



`p2ListID` The resource template version. Must be set to NIL.

`p2DefProc` Pointer to the definition procedure for the window. When using the `rWindParam2` window template, you must pass a pointer to a valid definition procedure, either in the template or with the *defProcPtr* parameter to the `NewWindow2` Window Manager tool call. On disk, this field does not contain a valid value.

`p2Data` Window definition data required by the routine pointed to by `p2DefProc`. The format and content of this field are determined by the window definition procedure.

Appendix F **Delta Guide**

This appendix collects all information that corrects errors or clarifies ambiguities in Volumes 1 and 2 of the *Apple IIGS Toolbox Reference.* This information was derived from the "Error Corrections" and "Clarifications" sections in the chapters of this book. This appendix contains a separate major section for each tool set to be addressed; the sections are presented alphabetically, by tool set name.

Apple Desktop Bus

The following sections correct errors or omissions in Chapter 3, "Apple Desktop Bus Tool Set," in Volume 1 of the *Toolbox Reference.*

Error corrections

The parameter table for the `ReadKeyMicroData` tool call ($0A09) in Volume 1 of the *Toolbox Reference* incorrectly describes the format of 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 list correctly describes `ReadConfigRec`, the configuration record for the `ReadKeyMicroData` tool call.

Name	Offset	Type	Definition
`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.

If you call `AsyncADBReceive` to poll a device using register 2, it returns 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

Audio Compression and Expansion Tool Set

The following section discusses an error in a previous version of this book.

Error correction

An error existed in the *Apple IIGS Toolbox Reference Update* (distributed by APDA). The description of the `ACEExpand` tool call included an incorrect parameter block. This book contains a corrected description.

Control Manager

The following sections correct errors or omissions in Chapter 4, "Control Manager," in Volume 1 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	Outline color	
		Bits 15–8	= zero
		Bits 7–4	= outline color
		Bits 3–0	= zero
`growNorBack`	word	Color of interior when not highlighted	
		Bits 15–8	= zero
		Bits 7–4	= background color
		Bits 3–0	= icon color
`growSelBack`	**word**	**Color of interior when highlighted**	
		Bits 15–8	**= zero**
		Bits 7–4	**= background color**
		Bits 3–0	**= icon color**

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

Clarifications

The following items provide additional information about features previously described in Volume 1 of the *Toolbox Reference.*

- The `barArrowBack` entry in the scroll bar color table was never implemented as first intended and is 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 boundary rectangles automatically. The computed size is based on the size of the title string of the button or box.
- To ensure predictable color behavior, you should always align controls based on color tables 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.

Dialog Manager

The following section corrects errors or omissions in Chapter 6, "Dialog Manager," in Volume 1 of the *Toolbox Reference.*

Error corrections

This section documents errors in Chapter 6, "Dialog Manager," in Volume 1 of the *Toolbox Reference.*

- A statement about `SetDItemType` on page 6-82 of Volume 1 of the *Toolbox Reference* is in error. This call is *not* 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.
- An entry in Table 6-3 on page 6-12 of Volume 1 of the *Toolbox Reference* is incorrect. The Dialog Manager does *not* support dialog item type values of `picItem` or `iconItem`.

Event Manager

The following section corrects an error in Chapter 7, "Event Manager," in Volume 1 of the *Toolbox Reference.*

Error correction

- The description of the `EMShutDown` tool call incorrectly states that the call returns no errors. This call can return any valid Event Manager error code.

Font Manager

The following section corrects an error in Chapter 8, "Font Manager," in Volume 1 of the *Toolbox Reference.*

Error corrections

- On page 8-4 of Volume 1 of the *Toolbox Reference,* the font family number for the Shaston font is given as 65,524. This is incorrect. The correct decimal value is 65,534 ($FFFE).
- Page 8-24, Volume 1 of the *Toolbox Reference* incorrectly describes the *newSpecs* parameter, indicating that it contains a word of `FontSpecBits`. Actually, this parameter contains `FontStatBits` for the new font.
- Contrary to the call description in the *Toolbox Reference,* the `FMSetSysFont` tool call does *not* load or install the indicated font.

Integer Math Tool Set

The following section describes a bug that has been fixed in the Integer Math Tool Set.

Clarification

This section presents new information about the `Long2Dec` Integer Math tool call.

- The `Long2Dec` Integer Math tool call now correctly handles input long values whose low-order three bytes are set to zero.

List Manager

The following sections correct errors or omissions in Chapter 11, "List Manager," in Volume 1 of the *Toolbox Reference.*

Clarifications

The following items provide additional information about features previously described in Volume 1 of the *Toolbox Reference.*

- The *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 a user can, for instance, select a disabled command as part of a help dialog box. To make an item unselectable, make it inactive (see "List Manager Definitions" later in this appendix).
- Any List Manager tool call that draws will change fields in the GrafPort record. 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.
- Item 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 understand this relationship, 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 automatically adjusts the `listRect.v2` field and sets 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 of a list item:

inactive	Inactive items appear dimmed and cannot be highlighted or selected. Bit 5 of the list item's `memFlag` field is set to 1.
disabled	Disabled items appear dimmed and cannot be highlighted. Bit 6 of the list item's `memFlag` field is set to 1.
enabled	Enabled items are not dimmed and can be highlighted. Bit 6 of the list item's `memFlag` field is set to 0.
selected	This bit is set when a user clicks the list item or when the item is in a range of selected items. A selected item appears highlighted only if it is also enabled. Bit 7 of the list item's `memFlag` field is set to 1.
highlighted	An item in a list appears highlighted only when it is both selected and enabled. A highlighted item is drawn in the highlight colors. Bit 7 of the `memFlag` field is set to 1 and bit 6 is set to 0.

Memory Manager

The following sections correct errors or omissions in Chapter 12, "Memory Manager," in Volume 1 of the *Toolbox Reference.*

Error correction

Figure 12-7 on page 12-10 of Volume 1 of the *Toolbox Reference* shows the low-order bit of the user ID as 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) includes the statement, "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 compacts memory or purges blocks 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."

Menu Manager

The following sections correct errors or omissions in Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference.*

Error corrections

This section documents errors in Chapter 13, "Menu Manager," in Volume 1 of the *Toolbox Reference.*

- Part of the description of the `SetSysBar` tool call (pages 13-86 and 13-3) in Volume 1 of the *Toolbox Reference* is incorrect. It includes the mistaken statement that, after an application issues this call, the new system menu bar becomes the current menu bar. In reality, 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 and 13-18), Volume 1 of the *Toolbox Reference* shows that bits 0–5 of the `ctlFlag` field are used to indicate the starting position of the first title in the menu bar. This is incorrect. The `ctlHilite` field defines the starting position of 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.
- The call descriptions for the `MenuKey` and `MenuSelect` tool calls are incorrect. The calls do not return selection status information in the `when` field of the event record. Rather, these calls both return selection status information in the `TaskData` field of the task record.

Clarifications

The following items provide additional information about features previously described in Volume 1 of the *Toolbox Reference.*

- 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 so that it points 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 to display the modified menu bar,your application must call `FixMenuBar` before calling `DrawMenuBar`.
- The description of the `InitPalette` tool call in the *Toolbox Reference* should also note that this 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 the size of a menu. These parameters may contain the width and height of the menu or may contain the value $0000 or $FFFF. A value of $0000 tells `CalcMenuSize` to calculate the parameter automatically. A value of $FFFF tells it to calculate the parameter only if the current setting is 0.

  These are the effects of all three uses:

  - **Pass the new value.** The value passed determines the size of the resulting menu. Use this method when you need a menu of a specific size.
  - **Pass $0000.** The size value is automatically computed. This option is useful if commands are added or deleted, resulting in an incorrect size. The height and width of the menu can be automatically adjusted by calling `CalcMenuSize` with *newWidth* and *newHeight* equal to $0000.
  - **Pass $FFFF.** The width and height of a menu are 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.
- To provide the user a consistent visual interface, you should always pad your menu titles with leading and trailing space characters. The Apple IIGS Finder uses two spaces.

Miscellaneous Tool Set

The following section corrects errors or omissions in Chapter 14, "Miscellaneous Tool Set," in Volume 1 of the *Toolbox Reference.*

Error corrections

This section documents errors in Chapter 14, "Miscellaneous Tool Set," in Volume 1 of the *Toolbox Reference.*

- On page 14-58 of Volume 1 of the *Toolbox Reference,* Figure 14-3 shows the low-order bit of the user ID as 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 two errors. In the code to clear the 1-second IRQ source, the second instruction reads

  ```
  TSB    $C032
  ```

  This instruction should read

  ```
  TRB    $C032
  ```

  In addition, preceding this instruction the following code should be inserted

  ```
  PEA    $0000
  PLB
  PLB
  ```

  These three instructions allow the code to reliably access the appropriate location in bank zero memory. These same three instructions should also be inserted in the code shown on page 14-29, immediately preceding the `STA` instruction.
- 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 a pointer to a pointer. Creating *startHandle* as a handle will cause unpredictable system behavior.
- Throughout Chapter 14 of the *Toolbox Reference* the value of the signature word for Miscellaneous Tool Set data structures is given as $5AA5 and $A55A. Signature words are *always* $A55A, *never* $5AA5.

Clarification

Note that the `ClrHeartBeat` tool call removes all tasks from the heartbeat interrupt task queue, including those installed by system software. Consequently, only system software should issue the `ClrHeartBeat` tool call.

Print Manager

The following sections correct errors or omissions in Chapter 15, "Print Manager," in Volume 1 of the *Toolbox Reference.*

Error corrections

This section documents errors in Volume 1 of the *Toolbox Reference.*

- The diagram of the job subrecord, Figure 15-10 on page 15-14 of Volume 1 of the *Toolbox Reference,* shows that the `fFromUsr` field is a word. This is incorrect. The `fFromUsr` field is actually a byte. Note that as a result the offsets for all fields following this one are incorrect. This error is also reflected in the tool set summary at the end of the chapter.
- The description of the `PrJobDialog` tool call includes this incorrect statement: "The initial settings displayed in the dialog box are taken from the printer driver." The sentence should be "The initial settings displayed in the dialog box are taken from the print record."

Clarifications

The following items provide additional information about features previously described in Volume 1 of the *Toolbox Reference.*

- 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 Volume 2 of the *Toolbox Reference* for the layout of the `locInfo` record).
- The port driver auxiliary file type of an AppleTalk driver is $0003. Its file type remains $BB.

QuickDraw II

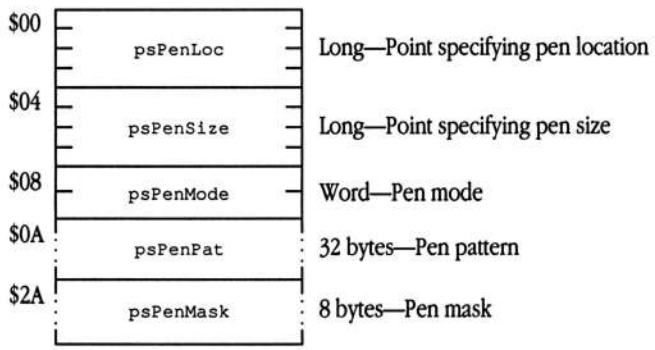
The following section corrects errors or omissions in Chapter 16, "QuickDraw II," in Volume 2 of the *Toolbox Reference.*

Error corrections

The following items provide corrections to the documentation for QuickDraw II in Volume 2 of the *Toolbox Reference:*

- The explanation of 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.
- Chapter 16 states that the coordinates passed to the `LineTo` and `MoveTo` calls should be expressed as global coordinates. In fact, the coordinates must be local and must refer to the GrafPort in which the drawing or moving takes place.
- The pen state record shown in Figure 16-38 on page 16-238 is incorrect. The correct record layout is shown in Figure F-1.

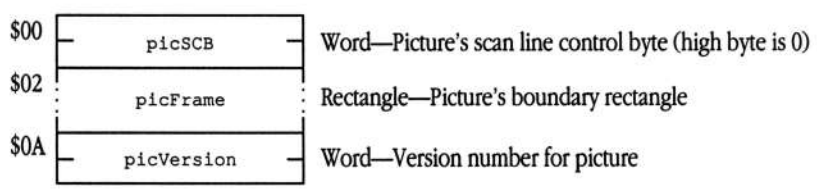
■ **Figure F-1** Pen state record



Clarification

QuickDraw pictures are described by a series of QuickDraw operation codes specifying the commands by which the picture was created. When these pictures are stored as data structures, the actual picture data (the operation codes) is preceded by control information, some of which may be of interest to Apple IIGS developers. Figure F-2 shows some of this control information. Note that the layout of this control information is subject to change.

■ **Figure F-2** QuickDraw picture header



Sound Tool Set

The following sections correct errors or omissions in Chapter 21, "Sound Tool Set," in Volume 2 of the *Toolbox Reference.*

Error corrections

This section contains corrections to the documentation of the Sound Tool Set in Volume 2 of the *Toolbox Reference.*

- The documentation of the `FFSoundDoneStatus` call contains an error. You will note that the paragraph that describes the call does not agree with the diagram describing the stack after the call. 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.
- The description of the `GetSoundVolume` tool call is misleading with respect to the number of significant bits in the returned volume setting. The text accompanying the stack diagram is correct—only the high nibble of the low-order byte contains valid volume data.
- The `FFGeneratorStatus` tool call can return error code $0813, indicating that the *genNumber* parameter contains an invalid generator number.

Clarification

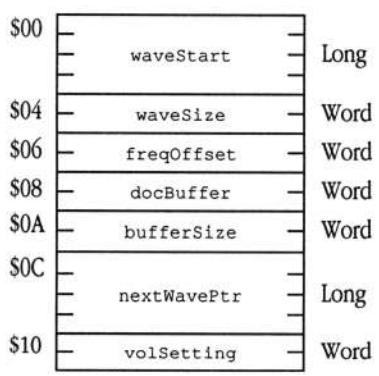
This section presents more complete information about the `FFStartSound` tool call, including further explanation of its parameters, a new error code, an example procedure for moving a sound from the Macintosh computer to the Apple IIGS computer, and some sample code demonstrating the use of the call. The original documentation for this call is in Chapter 21, "Sound Tool Set," in Volume 2 of the *Toolbox Reference.*

FFStartSound

The free-form synthesizer is designed to play back long waveforms. 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 4 KB buffers, then those buffers must be aligned on 4 KB memory boundaries.

Parameter block



`waveStart` The starting address of the wave to be played, not in Digital Oscillator Chip (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.
`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 3 bits set the values for the table-size and resolution portions of the DOC Bank-Select/Table-Size/Resolution register.
`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 computer to the Apple IIGS computer

To move a digitized sound from the Macintosh computer to the Apple IIGS computer and play the sound, you perform the following steps:

1. Save the sound as a pure data file on the Macintosh computer.
2. Transfer the file to the Apple IIGS computer (using Apple File Exchange, for example).
3. Filter all the 0 sample bytes out of the file by replacing them with bytes set to $01. This is very important, because the Apple IIGS computer interprets 0 bytes as the end of a sample.
4. Load the sound into memory with GS/OS calls.
5. Issue the `FFStartSound` tool call to play the sound. Set the `freqOffset` parameter to $01B7 to match the tempo at which the sound is played on the Macintosh computer, assuming that you recorded the original sound at the standard Macintosh sampling rate of 22 kHz.

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 param block
            _FFStartSound               ; Start free-form 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
```

Tool Locator

The following sections correct errors or omissions in Chapter 24, "Tool Locator," in Volume 2 of the *Toolbox Reference.*

Error correction

Contrary to the call descriptions in Chapter 24 of the *Toolbox Reference,* both the `MessageCenter` and `SaveTextState` tool calls can return Memory Manager errors.

Clarification

Applications that explicitly start up Apple IIGS tool sets should start the Desk Manager last.

Window Manager

The following section corrects errors or omissions in Chapter 25, "Window Manager," in Volume 2 of the *Toolbox Reference.*

Error corrections

This section corrects some errors in the documentation of the Window Manager in Volume 2 of the *Toolbox Reference.*

- The description of `SetZoomRect` is incorrect. The correct description is as follows:

  Sets the `fZoomed` bit of the window's `wFrame` record to 0. The rectangle passed to `SetZoomRect` then becomes the window's zoom rectangle. The window's size and position when `SetZoomRect` is called become the window's unzoomed size and position, regardless of what the unzoomed characteristics were before `SetZoomRect` was called.

- "If `wmTaskMask` bit `tmInfo` (bit 15) = 1," on page 25-126, should read, "If `wmTaskMask` bit `tmInfo` (bit 15) = 0."

- When used with a window that does not have scroll bars, the `WindNewRes` call invokes the window's defProc to recompute window regions. A call to `SizeWindow` is not necessary under these circumstances.

- The input region for the `InvalRgn` tool call is defined in local coordinates; however, the call returns the region expressed in global coordinates.

- There are two errors in the series of equations given with the `PinRect` tool call. In the last two equations the greater-than sign (>) should be replaced with a greater-than-or-equal sign (>=).

- Note that the `CloseWindow` tool call does *not* change the GrafPort setting. Your application should ensure that a valid GrafPort is set before performing any other actions.

Clarifications

This section elaborates on topics addressed in Volume 2 of the *Toolbox Reference.*

- Window title strings should always contain leading and trailing space characters. This spacing is especially important for windows with a lined window bar because, without the spaces, the line pattern runs into the title text. Also, because window editor desk accessories may allow the user to change the title bar pattern without making the change known to your application, you should pad your window titles with spaces even if you use black title bars.
- Table 25-6 on page 25-43 of the *Toolbox Reference* contains misleading labels. Note that in this table byte 1 refers to the high-order byte of the long that defines the desktop pattern, and byte 4 refers to the low-order byte.

Appendix G **Toolbox Code Example**

This appendix contains a sample program, BusyBox, that demonstrates the use of many of the new features of the Apple IIGS Toolbox.

The `Busy.p` module

This section contains the source listing for the main module of the BusyBox program.

```
{*************************************************************************
{*
{* BusyBox (Main Program)
{*
{* Copyright (c)
{* Apple Computer, Inc. 1986-1990
{* All Rights Reserved.
{*
{* This file contains the BusyBox program.
{*
{*************************************************************************}
{$R-}

program BusyBox;

USES
      types,
      gsos,
      Quickdraw,
      fonts,
      memory,
      IntMath,
      events,
      prodos,
      locator,
      controls,
      windows,
      lists,
      scrap,
      lineedit,
      dialogs,
      menus,
      desk,
      STDFile,
      QDAUX,
      print,
      miscTool,
      resources,
```

```
      uGlobals,          { HodgePodge Code Units }
      uUtils,
      uWindow,
      uMenu,
      uEvent;

var
      InitRef : ref;     { This holds the reference to the startstop
                           record }

BEGIN                    { of MAIN program BusyBox }

                         { Init our globals }
      InitGlobals;

      MyMemoryID := MMStartup;
                         { Start up & get ID from the Memory Manager }
      TLStartUp;         { Start up the tool locator }
                         { Startup the tools using the new toolbox call }

      InitRef := StartupTools(MyMemoryID,RefIsResource,ref(1));
      if _toolErr = 0 then     { note: usage of _toolErr may vary between
                                 compilers }

            begin
                  SetUpMenus; { Set up menus }
                  SetupWindows;
                  InitCursor; { Make cursor show ready }
                  MainEvent;  { Use application }
            end;

      { Let the toolbox shut down the tools }
      ShutDownTools(RefIsHandle,InitRef);
      TLShutDown; { Shut down the tool locator }

END.  { of MAIN program BusyBox }
```

The `busybox.r` module

This section contains the MPW source statements for the Apple IIGS resource editor that create the resource file for the BusyBox program.

```
/*-------------------------------------------------------------*/

/* For APW, the following should read '#include "types.rez"'.     */
#include "typesiigs.r"

/*-------------------- Values used throughout --------------------*/

#define MainWindow                  $2000
#define ButtonWindow                $2001
#define StatTextWindow              $2002
#define LineEditWindow              $2003
#define PictureWindow               $2004
#define PopUpWindow                 $2005
#define TextEditWindow              $2006
#define ListWindow                  $2007
#define Prog1Window                 $2008
#define Prog2Window                 $2009
#define Prog3Window                 $200A
#define Prog4Window                 $200B
#define Prog5Window                 $200C
#define Prog6Window                 $200D

#define ButButtons                  $0001
#define ButStatText                 $0002
#define ButLineEdit                 $0003
#define ButPictures                 $0004
#define ButPopUps                   $0005
#define ButTextEdit                 $0006
#define ButLists                    $0007
#define ButProg1                    $0008
#define ButProg2                    $0009
#define ButProg3                    $000A
#define ButProg4                    $000B
#define ButProg5                    $000C
#define ButProg6                    $000D
#define MainText                    $000E
```

```
#define AboutBusyAlert        1
#define BusyBoxStartup        1


/*----------------------- About Box -----------------------------*/
resource rAlertString (AboutBusyAlert) {
      "0\$19\$00\$A0\$00\$AA\$00\$E0\$01"
      "0/"
      TBCenterJust
      TBStyleOutline
      "BusyBox"
      TBEndOfLine
      TBStylePlain
      "A sample program to demonstrate the new features of the "
      "Apple IIGS toolbox."
      TBEndOfLine
      TBEndOfLine
      "by"
      TBEndOfLine
      "Steven E. Glass"
      TBEndOfLine
      TBEndOfLine
      "Copyright Apple Computer, Inc."
      TBEndOfLine
      "All Rights Reserved"
      TBEndOfLine
      "Version 1.1/^#6\$00"
};
```

```
/*--------------------------Startup Record --------------------------*/
resource rToolStartup (BusyBoxStartup) {
      mode640,                                  /* master SCB */
 {
            3,$0100,                            /* Misc Tool */
            4,$0100,                            /* QuickDraw */
            5,$0100,                            /* Desk Manager */
            6,$0100,                            /* Event Manager */
/*          7,$0100,                            /* Scheduler */
/*          8,$0100,                            /* sound tools */
/*          9,$0100,                            /* ADB tools */
/*          10,$0100,                           /* SANE */
            11,$0100,                           /* Int Math */
            14,$0300,                           /* Window Manager */
            15,$0300,                           /* Menu Manager */
            16,$0300,                           /* Control Manager */
            18,$0200,                           /* QD AUX */
            19,$0100,                           /* Print Manager */
            20,$0100,                           /* LineEdit Tool Set */
            21,$0100,                           /* Dialog Manager */
            22,$0100,                           /* Scrap Manager */
            23,$0100,                           /* Standard File */
            27,$0100,                           /* Font Manager */
            28,$0100,                           /* List Manager */
            34,$0100,                           /* TextEdit */
/*          29,$0100,                           /* ACE */
/*          32,$0100,                           /* MIDI Tools */
/*          25,$0100,                           /* Note Synth */
/*          26,$0100                            /* Note Seq */
      }
};
```

```
/*-------------------------------------------------------------------*/
/*
/* Main Window
/*
/* This is the template for the main window with all the buttons that
/* lead to other buttons.
/*
/*-------------------------------------------------------------------*/
resource rWindParam1 (MainWindow) {
      fTitle+fVis,                          /* frame bits              */
       MainWindow,                          /* title id                */
      0,                                    /* ref con                 */
      {0,0,0,0},                            /* zoom rect               */
      0,                                    /* color table id          */
      {0,0},                                /* origin                  */
      {0,0},                                /* data size               */
      {0,0},                                /* max height-width        */
      {0,0},                                /* scroll amount, hor,ver  */
      {0,0},                                /* page amount             */
      0,                                    /* wInfo ref con           */
      0,                                    /* wInfo height            */
      {40,90,180,550},                      /* window position         */
      infront,                              /* wPlane                  */
      MainWindow,                           /* control ref             */
      refIsResource*0x0100+resourceToResource
                                            /* descriptor              */
};

/*-------------------------------------------------------------------*/
/* This is the title of the main window.
/*-------------------------------------------------------------------*/
resource rPString (MainWindow) {
      "BusyBox"
 };
```

```
/*-------------------------------------------------------------------*/
/* The following define the controls for the main window.
/* First I start with some constants.
/*-------------------------------------------------------------------*/
#define ButWidth              140
#define ButHeight             12
#define ButSep                8
#define ButVSep               5

#define TopOfRow1             50
#define BottomOfRow1          TopOfRow1+ButHeight
#define TopOfRow2             BottomOfRow1+ButVSep
#define BottomOfRow2          TopOfRow2+ButHeight
#define TopOfRow3             BottomOfRow2+ButVSep
#define BottomOfRow3          TopOfRow3+ButHeight
#define TopOfRow4             BottomOfRow3+ButVSep+ButVSep
#define BottomOfRow4          TopOfRow4+ButHeight
#define TopOfRow5             BottomOfRow4+ButVSep
#define BottomOfRow5          TopOfRow5+ButHeight

#define LeftEdge1             ButSep
#define RightEdge1            LeftEdge1+ButWidth
#define LeftEdge2             RightEdge1+ButSep
#define RightEdge2            LeftEdge2+ButWidth
#define LeftEdge3             RightEdge2+ButSep
#define RightEdge3            LeftEdge3+ButWidth
```

```
/* List of all controls in main window */

resource rControlList (MainWindow) {
      {
            ButButtons,
            ButStatText,
            ButLineEdit,
            ButPictures,
            ButPopUps,
            ButTextEdit,
            ButLists,
            ButProg1,
            ButProg2,
            ButProg3,
            ButProg4,
            ButProg5,
            ButProg6,
            MainText
      };
};
```

```
resource rControlTemplate (MainText) {
      MainText,                  /* control id */
      {2,4,42,456},              /* control rectangle */
      EditTextControl{{          /* control type */
            0x0000,              /* flag */
            fCtlCanBeTarget+fCtlWantEvents+fctlProcRefNotPtr+
                  fCtlIsMultiPart,
                                 /* more flags */
            0,                   /* ref con */
            fReadOnly+fDrawBounds,
                                 /* text flags */
            {0xFFFF,0xFFFF,0xFFFF,0xFFFF},
                                 /* indent rect */
            0xFFFFFFFF,          /* vert bar */
            0,                   /* vert amount */
            0,                   /* hor bar */
            0,                   /* hor amount */
            0,                   /* style ref */
            dataIsTextBlock+RefIsResource*8,
                                 /* text descriptor */
            MainText,            /* text ref */
            0                    /* text size (not used) */


      }};
};
```

```
/* The static text for main window */
resource rText (MainText) {
      "The new toolbox makes it much easier to write programs for the "
      "Apple IIGS."
      TBEndOfLine
      TBEndOfLine
      "This program is incredibly simple. "
      TBEndOfLine
      TBEndOfLine
      "Press one of the round buttons to find out about the new kinds "
      "of controls that are supported. "
      TBEndOfLine
      TBEndOfLine
      "Press one of the square "
      "buttons to see the code for this program."
};

/* The definition of the Buttons button */
resource rControlTemplate (ButButtons) {
      ButButtons,             /* control id */
      {TopOfRow1,LeftEdge1,BottomOfRow1,RightEdge1},
                              /* control rect */
      SimpleButtonControl{{   /* control type */
            NormalButton,     /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                              /* more flags */
            0,                /* ref con */
            ButButtons        /* title ref */
      }};
};

resource rpString (ButButtons) {
 "Buttons..."
};
```

```
/* The Static Text button */
resource rControlTemplate (ButStatText) {
      ButStatText,            /* control id */
      {TopOfRow1,LeftEdge2,BottomOfRow1,RightEdge2},
                              /* control rect */
      SimpleButtonControl{{   /* control type */
            NormalButton,     /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                              /* more flags */
            0,                /* ref con */
            ButStatText       /* title ref */
      }};
};

resource rpString (ButStatText) {
 "Static Text..."
};

/* The Line Edit button */
resource rControlTemplate (ButLineEdit) {
      ButLineEdit,            /* control id */
      {TopOfRow1,LeftEdge3,BottomOfRow1,RightEdge3},
                              /* control rect */
      SimpleButtonControl{{   /* control type */
            NormalButton,     /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                              /* more flags */
            0,                /* ref con */
            ButLineEdit       /* title ref */
      }};
};

resource rpString (ButLineEdit) {
 "Line Edit..."
};
```

```
/* The Pictures button */
resource rControlTemplate (ButPictures) {
      ButPictures,              /* control id */
      {TopOfRow2,LeftEdge1,BottomOfRow2,RightEdge1},
                                /* control rect */
      SimpleButtonControl{{     /* button type */
            NormalButton,       /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                                /* more flags */
            0,                  /* ref con */
            ButPictures         /* title ref */
      }};
};

resource rpString (ButPictures) {
 "Pictures..."
};


/* The Pop-ups button */
resource rControlTemplate (ButPopUps) {
      ButPopUps,                /* control id */
      {TopOfRow2,LeftEdge2,BottomOfRow2,RightEdge2},
                                /* control rect */
      SimpleButtonControl{{     /* control type */
            NormalButton,       /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                                /* more flags */
            0,                  /* ref con */
            ButPopUps           /* title ref */
      }};
};

resource rpString (ButPopUps) {
 "Pop-up Menus..."
};
```

```
/* The TextEdit button */
resource rControlTemplate (ButTextEdit) {
      ButTextEdit,            /* control id */
      {TopOfRow2,LeftEdge3,BottomOfRow2,RightEdge3},
                              /* control rect */
      SimpleButtonControl{{   /* control type */
            NormalButton,     /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                              /* more flags */
            0,                /* ref con */
            ButTextEdit       /* title ref */
      }};
};

resource rpString (ButTextEdit) {
 "Text Edit..."
};


/* The Lists button */
resource rControlTemplate (ButLists) {
      ButLists,               /* control id */
      {TopOfRow3,LeftEdge2,BottomOfRow3,RightEdge2},
                              /* control rect */
      SimpleButtonControl{{   /* control type */
            NormalButton,     /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                              /* more flags */
            0,                /* ref con */
            ButLists          /* title ref */
      }};
};

resource rpString (ButLists) {
 "Lists..."
};
```

```
/* The Main Program button */
resource rControlTemplate (ButProg1) {
      ButProg1,                  /* control id */
      {TopOfRow4,LeftEdge1,BottomOfRow4,RightEdge1},
                                 /* control rect */
      SimpleButtonControl{{      /* control type */
            SquareButton,        /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                                 /* more flags */
            0,                   /* ref con */
            ButProg1             /* title ref */
      }};
};

resource rpString (ButProg1) {
 "Main Program..."
};

/* The Main Program button */
resource rControlTemplate (ButProg2) {
      ButProg2,                  /* control id */
      {TopOfRow4,LeftEdge2,BottomOfRow4,RightEdge2},
                                 /* control rect */
      SimpleButtonControl{{      /* control type */
            SquareButton,        /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                                 /* more flags */
            0,                   /* ref con */
            ButProg2             /* title ref */
      }};
};

resource rpString (ButProg2) {
 "Events..."
};
```

```
/* The Main Program button */
resource rControlTemplate (ButProg3) {
      ButProg3,                  /* control id */
      {TopOfRow4,LeftEdge3,BottomOfRow4,RightEdge3},
                                 /* control rect */
      SimpleButtonControl{{      /* control type */
            SquareButton,        /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                                 /* more flags */
            0,                   /* ref con */
            ButProg3             /* title ref */
      }};
};

resource rpString (ButProg3) {
 "Menus..."
};

/* The Main Program button */
resource rControlTemplate (ButProg4) {
      ButProg4,                  /* control id */
      {TopOfRow5,LeftEdge1,BottomOfRow5,RightEdge1},
                                 /* control rect */
      SimpleButtonControl{{      /* control type */
            SquareButton,        /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                                 /* more flags */
            0,                   /* ref con */
            ButProg4             /* title ref */
      }};
};

resource rpString (ButProg4) {
 "Windows..."
};
```

```
/* The Main Program button */
resource rControlTemplate (ButProg5) {
      ButProg5,                  /* control id */
      {TopOfRow5,LeftEdge2,BottomOfRow5,RightEdge2},
                                 /* control rect */
       SimpleButtonControl{{     /* control type */
            SquareButton,        /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                                 /* more flags */
            0,                   /* ref con */
            ButProg5             /* title ref */
      }};
};

resource rpString (ButProg5) {
 "Utilities..."
};

/* The Main Program button */
resource rControlTemplate (ButProg6) {
      ButProg6,                  /* control id */
      {TopOfRow5,LeftEdge3,BottomOfRow5,RightEdge3},
                                 /* control rect */
      SimpleButtonControl{{      /* control type */
            SquareButton,        /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                                 /* more flags */
            0,                   /* ref con */
            ButProg6             /* title ref */
      }};
};

resource rpString (ButProg6) {
 "Globals..."
};
```

```
/*--------------------------------------------------------------------*/
/*
/* Buttons...
/*
/* The List window uses IDs in the $3000 range.
/*
/*--------------------------------------------------------------------*/

#define ButtonTextID                $3001
#define But1                        $3101
#define But2                        $3102
#define But3                        $3103
#define But4                        $3104
#define Check1                      $3105
#define Check2                      $3106
#define Check3                      $3107
#define Check4                      $3108
#define Radio1                      $3109
#define Radio2                      $310A
#define Radio3                      $310B
#define Radio4                      $310C
#define Icon1                       $310D
#define Icon2                       $310E

#define BLine1              50
#define BLine2              BLine1+18
#define BLine3              BLine2+18
#define BLine4              BLine3+18
```

```
resource rWindParam1 (ButtonWindow) {
      fTitle+fMove+fZoom+fGrow+fBScroll+fRScroll+fClose,
                              /* frame bits */
      ButtonWindow,           /* title id */
      0,                      /* ref con */
      {0,0,0,0},              /* zoom rect */
      0,                      /* color table id */
      {0,0},                  /* origin */
      {400,640},              /* data size */
      {200,640},              /* max height-width */
      {1,1},                  /* scroll amount, hor,ver */
      {0,0},                  /* page amount */
      0,                      /* wInfo ref con */
      0,                      /* wInfo height */
      {50,50,120,260},        /* window position */
      infront,                /* wPlane */
      ButtonWindow,           /* control ref */
      refIsResource*0x0100+resourceToResource
                              /* descriptor */
};


resource rpString (ButtonWindow) {
 "Buttons Window"
};
```

```
resource rControlList (ButtonWindow) {
      {
            ButtonTextID,
            But1,
            But2,
            But3,
            But4,
            Check1,
            Check2,
            Check3,
            Check4,
            Radio1,
            Radio2,
            Radio3,
            Radio4,
            Icon1,
            Icon2
       };
};

/* Template for static text in main window */
resource rControlTemplate (ButtonTextID) {
      ButtonTextID,           /* control id */
      {2,4,48,460},           /* control rectangle */
      StatTextControl{{       /* control type */
            ctlInactive,      /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                              /* more flags */
            0,                /* ref con */
            ButtonTextID      /* title ref */
      }};
};

/* The static text for List window */
resource rTextForLETextBox2 (ButtonTextID) {
      "There are four types of buttons: simple buttons, check boxes, "
      "radio buttons, and Icon Buttons. Each button can have its own "
      "keyboard equivalent. All tracking and hiliting is handled by "
      "TaskMaster."
};
```

```
resource rControlTemplate (But1) {
      But1,                    /* control id */
      {BLine1,LeftEdge1,0,0}, /* control rect */
      SimpleButtonControl{{    /* control type */
            NormalButton,      /* flag */
            fctlProcRefNotPtr+fCtlWantEvents+RefIsResource,
                               /* more flags */
            0,                 /* ref con */
            But1,              /* title ref */
            0,                 /* color table not used */
            {"A","a",0,0}      /* key equiv */
      }};
};

resource rpString (But1) {
 "Normal Button (A)"
};


resource rControlTemplate (But2) {
      But2,                    /* control id */
      {BLine2,LeftEdge1,0,0}, /* control rect */
      SimpleButtonControl{{    /* control type */
            DefaultButton,     /* flag */
            fctlProcRefNotPtr+fCtlWantEvents+RefIsResource,
                               /* more flags */
            0,                 /* ref con */
            But2,              /* title ref */
            0,                 /* color table not used */
            {"B","b",0,0}      /* key equiv */
      }};
};

resource rpString (But2) {
      "Default Button (B)"
};
```

```
resource rControlTemplate (But3) {
      But3,                    /* control id */
      {BLine3,LeftEdge1,0,0}, /* control rect */
      SimpleButtonControl{{    /* control type */
            SquareButton,      /* flag */
            fctlProcRefNotPtr+fCtlWantEvents+RefIsResource,
                               /* more flags */
            0,                 /* ref con */
            But3,              /* title ref */
            0,                 /* color table not used */
            {"C","c",0,0}      /* key equiv */
      }};
};

resource rpString (But3) {
      "Square Button (C)"
};


resource rControlTemplate (But4) {
      But4,                    /* control id */
      {BLine4,LeftEdge1,0,0}, /* control rect */
      SimpleButtonControl{{    /* control type */
            SquareShadowButton,
                               /* flag */
            fctlProcRefNotPtr+fCtlWantEvents+RefIsResource,
                               /* more flags */
            0,                 /* ref con */
            But4,              /* title ref */
            0,                 /* color table not used */
            {"D","d",0,0}      /* key equiv */
      }};
};

resource rpString (But4) {
      "Square Shadow Button (D)"
};
```

```
resource rControlTemplate (Check1) {
      Check1,                 /* control id */
      {BLine1,LeftEdge3,0,0}, /* control rect */
      CheckControl{{          /* control type */
            0,                /* flag */
            fctlProcRefNotPtr+fCtlWantEvents+RefIsResource,
                              /* more flags */
            0,                /* ref con */
            Check1,           /* title ref */
            1,                /* initial value */
            0,                /* color table not used */
            {"e","E",0,0}     /* key equiv */
      }};
};

resource rpString (Check1) {
      "Check One (E)"
};

resource rControlTemplate (Check2) {
      Check2,                 /* control id */
      {BLine1+10,LeftEdge3,0,0},
                              /* control rect */
      CheckControl{{          /* control type */
            0,                /* flag */
            fctlProcRefNotPtr+fCtlWantEvents+RefIsResource,
                              /* more flags */
            0,                /* ref con */
            Check2,           /* title ref */
            1,                /* initial value */
            0,                /* color table not used */
            {"f","F",0,0}     /* key equiv */
      }};
};

resource rpString (Check2) {
      "Check Two (F)"
};
```

```
resource rControlTemplate (Check3) {
      Check3,                    /* control id */
      {BLine1+20,LeftEdge3,0,0},
                                 /* control rect */
      CheckControl{{             /* control type */
            0,                   /* flag */
            fctlProcRefNotPtr+fCtlWantEvents+RefIsResource,
                                 /* more flags */
            0,                   /* ref con */
            Check3,              /* title ref */
            0,                   /* initial value */
            0,                   /* color table not used */
            {"G","g",0,0}        /* key equiv */
      }};
};

resource rpString (Check3) {
      "Check Three (G)"
};

resource rControlTemplate (Check4) {
      Check4,                    /* control id */
      {BLine1+30,LeftEdge3,0,0},
                                 /* control rect */
      CheckControl{{             /* control type */
            0,                   /* flag */
            fctlProcRefNotPtr+fCtlWantEvents+RefIsResource,
                                 /* more flags */
            0,                   /* ref con */
            Check4,              /* title ref */
            1,                   /* initial value */
            0,                   /* color table not used */
            {"H","h",0,0}        /* key equiv */
      }};
};

resource rpString (Check4) {
      "Check Four (H)"
};
```

```
resource rControlTemplate (Radio1) {
      Radio1,                    /* control id */
      {BLine4,LeftEdge3,0,0},    /* control rect */
      RadioControl{{             /* control type */
            0,                   /* flag */
            fctlProcRefNotPtr+fCtlWantEvents+RefIsResource,
                                 /* more flags */
            0,                   /* ref con */
            Radio1,              /* title ref */
            0,                   /* initial value */
            0,                   /* color table not used */
            {"i","I",0,0}        /* key equiv */
      }};
};

resource rpString (Radio1) {
      "Radio One (I)"
};

resource rControlTemplate (Radio2) {
      Radio2,                    /* control id */
      {BLine4+10,LeftEdge3,0,0},     /* control rect */
      RadioControl{{             /* control type */
            0,                   /* flag */
            fctlProcRefNotPtr+fCtlWantEvents+RefIsResource,
                                 /* more flags */
            0,                   /* ref con */
            Radio2,              /* title ref */
            1,                   /* initial value */
            0,                   /* color table not used */
            {"J","j",0,0}        /* key equiv */
      }};
};

resource rpString (Radio2) {
      "Radio Two (J)"
};
```

```
resource rControlTemplate (Radio3) {
      Radio3,                    /* control id */
      {BLine4+20,LeftEdge3,0,0},
                                 /* control rect */
      RadioControl{{             /* control type */
            0,                   /* flag */
            fctlProcRefNotPtr+fCtlWantEvents+RefIsResource,
                                 /* more flags */
            0,                   /* ref con */
            Radio3,              /* title ref */
            0,                   /* initial value */
            0,                   /* color table not used */
            {"K","k",0,0}        /* key equiv */
      }};
};

resource rpString (Radio3) {
      "Radio Three (K)"
};

resource rControlTemplate (Radio4) {
      Radio4,                    /* control id */
      {BLine4+30,LeftEdge3,0,0},
                                 /* control rect */
      RadioControl{{             /* control type */
            0,                   /* flag */
            fctlProcRefNotPtr+fCtlWantEvents+RefIsResource,
                                 /* more flags */
            0,                   /* ref con */
            Radio4,              /* title ref */
            0,                   /* initial value */
            0,                   /* color table not used */
            {"L","l",0,0}        /* key equiv */
      }};
};

resource rpString (Radio4) {
      "Radio Four (L)"
};
```

```
resource rControlTemplate (Icon1) {
      Icon1,                  /* control id */
      {BLine4+20,LeftEdge1,BLine4+20+40,LeftEdge1+100},
                              /* control rect */
      IconButtonControl{{     /* control type */
            SquareButton,     /* flag */
    fctlProcRefNotPtr+fCtlWantEvents+RefIsResource+RefIsResource*$0010,
                              /* more flags */
            0,                /* ref con */
            Icon1,            /* icon ref */
            Icon1,            /* title ref */
            0,                /* color table not used */
            0,                /* display mode */
            {"M","m",0,0}     /* key equiv */
      }};
};

resource rpString (Icon1) {
      "Icon One (M)"
};

resource rIcon (Icon1) {
      0x8000,                 /* kind */
      20,                     /* height */
      28,                     /* width */
```

```
        $"FFFFFFFFFFF00000FFFFFFFFFFF"
        $"FFFFFFFF000ddddd000FFFFFFFF"
        $"FFFFFF0088888ddddddd00FFFFFF"
        $"FFFFF0d888888d888dd8d0FFFFFF"
        $"FFFF08888888dd888dd8880FFFFF"
        $"FFFF08888888dd88dd88880FFFFF"
        $"FFF08888888ddddddddd88880FFFF"
        $"FFF08888888ddddddddddd8d0FFFF"
        $"FF0d8d88dd8ddddddd8888880FFF"
        $"FF0d8d88dd8dddddd88888880FFF"
        $"FF0dddd8ddddddddd88888880FFF"
        $"FF0dd88888ddddddd88888880FFF"
        $"FFF08888888ddddddd8888880FFF"
        $"FFF08888888ddddddd888880FFFF"
        $"FFFF0888888ddddddddd8880FFFFF"
        $"FFFF088888ddddddddd8880FFFFF"
        $"FFFFF08888ddddddddd880FFFFFF"
        $"FFFFFF008ddddddddddd00FFFFFFF"
        $"FFFFFFFF000ddddd000FFFFFFFFF"
        $"FFFFFFFFFFF00000FFFFFFFFFFF",

        $"000000000000FFFFF000000000000"
        $"00000000FFFFFFFFFFF000000000"
        $"000000FFFFFFFFFFFFFFF0000000"
        $"00000FFFFFFFFFFFFFFFFF000000"
        $"0000FFFFFFFFFFFFFFFFFFF00000"
        $"0000FFFFFFFFFFFFFFFFFFF00000"
        $"000FFFFFFFFFFFFFFFFFFFFF0000"
        $"000FFFFFFFFFFFFFFFFFFFFF0000"
        $"00FFFFFFFFFFFFFFFFFFFFFFF000"
        $"00FFFFFFFFFFFFFFFFFFFFFFF000"
        $"00FFFFFFFFFFFFFFFFFFFFFFF000"
        $"00FFFFFFFFFFFFFFFFFFFFFFF000"
        $"000FFFFFFFFFFFFFFFFFFFFFF0000"
        $"000FFFFFFFFFFFFFFFFFFFFFF0000"
        $"0000FFFFFFFFFFFFFFFFFFF00000"
        $"0000FFFFFFFFFFFFFFFFFFF00000"
        $"00000FFFFFFFFFFFFFFFFF000000"
        $"000000FFFFFFFFFFFFFFF0000000"
        $"00000000FFFFFFFFFFF000000000"
        $"000000000000FFFFF000000000000";
};
```

```
resource rControlTemplate (Icon2) {
      Icon2,                    /* control id */
      {BLine4+20,LeftEdge2,BLine4+20+40,LeftEdge2+100},
                                /* control rect */
      IconButtonControl{{       /* control type */
            SquareButton,       /* flag */
     fctlProcRefNotPtr+fCtlWantEvents+RefIsResource+RefIsResource*$0010,
                                /* more flags */
            0,                  /* ref con */
            Icon2,              /* icon ref */
            Icon2,              /* title ref */
            0,                  /* color table not used */
            0,                  /* display mode */
            {"N","n",0,0}       /* key equiv */
      }};
};

resource rpString (Icon2) {
      "Icon Two (N)"
};

resource rIcon (Icon2) {
      0x8000,           /* kind */
      20,               /* height */
      28,               /* width */
```

```
$"FFFFFFFFFFFFFFFFFFFFFFFFFFFF"
$"FFFFFFFFFFFFFFFFFFFFFFFFFFFF"
$"FFFFFFFFFFFFFFFFFFFFFFFFFFFF"
$"FFFFFFFFFFFFFFFFFFFFFFFFFFFF"
$"FFFFFFFFFF0FFFFFFFFFDFFFFFFF"
$"FFFFFFFFF00FFFFFFFEFFDFFFFFF"
$"FFFFFFFF0F0FFFFFAFFEFFDFFFFF"
$"FFFFFFF0FF0FF7FFAFFEFFDFFFFF"
$"FF00000FFF0FFF7FFAFFEFFDFFFF"
$"F0FFFF0FFF0FFF7FFAFFEFFDFFFF"
$"F0FFFF0FFF0FFF7FFAFFEFFDFFFF"
$"F0FFFF0FFF0FFF7FFAFFEFFDFFFF"
$"FF00000FFF0FFF7FFAFFEFFDFFFF"
$"FFFFFFF0FF0FF7FFAFFEFFDFFFFF"
$"FFFFFFFF0F0FFFFFAFFEFFDFFFFF"
$"FFFFFFFFF00FFFFFFFEFFDFFFFFF"
$"FFFFFFFFFF0FFFFFFFFFDFFFFFFF"
$"FFFFFFFFFFFFFFFFFFFFFFFFFFFF"
$"FFFFFFFFFFFFFFFFFFFFFFFFFFFF"
$"FFFFFFFFFFFFFFFFFFFFFFFFFFFF",
```

```
        $"00000000000000000000000000000000"
        $"00000000000000000000000000000000"
        $"00000000000000000000000000000000"
        $"00000000000000000000000000000000"
        $"0000000000F000000000F0000000"
        $"000000000FF0000000F00F000000"
        $"00000000FFF00000F00F00F00000"
        $"0000000FFFF00F00F00F00F00000"
        $"00FFFFFFFFF000F00F00F00F0000"
        $"0FFFFFFFFFF000F00F00F00F0000"
        $"0FFFFFFFFFF000F00F00F00F0000"
        $"0FFFFFFFFFF000F00F00F00F0000"
        $"00FFFFFFFFF000F00F00F00F0000"
        $"0000000FFFF00F00F00F00F00000"
        $"00000000FFF00000F00F00F00000"
        $"000000000FF0000000F00F000000"
        $"0000000000F000000000F0000000"
        $"00000000000000000000000000000000"
        $"00000000000000000000000000000000"
        $"00000000000000000000000000000000";

};
```

```
/*------------------------------------------------------------------*/
/*
/* StatText...
/*
/* The StatText window uses IDs in the $4000 range.
/*
/*------------------------------------------------------------------*/

#define StatTextTextID $4001

resource rWindParam1 (StatTextWindow) {
      fTitle+fMove+fZoom+fGrow+fBScroll+fRScroll+fClose,
                              /* frame bits */
      StatTextWindow,         /* title id */
      0,                      /* ref con */
      {0,0,0,0},              /* zoom rect */
      0,                      /* color table id */
      {0,0},                  /* origin */
      {400,640},              /* data size */
      {200,640},              /* max height-width */
      {1,1},                  /* scroll amount, hor,ver */
      {0,0},                  /* page amount */
      0,                      /* wInfo ref con */
      0,                      /* wInfo height */
      {50,50,120,260},        /* window position */
      infront,                /* wPlane */
      StatTextWindow,         /* control ref */
      refIsResource*0x0100+resourceToResource
                              /* descriptor */
};



resource rpString (StatTextWindow) {
      "Static Text Window"
};
```

```
resource rControlList (StatTextWindow) {
      {
            StatTextTextID,
            0
      };
};

/* Template for static text in main window */
resource rControlTemplate (StatTextTextID) {
      StatTextTextID,          /* control id */
      {2,4,200,560},           /* control rectangle */
      StatTextControl{{        /* control type */
            ctlInactive+fSubstituteText,
                               /* flag */
            fctlProcRefNotPtr+RefIsResource,
                               /* more flags */
            0,                 /* ref con */
            StatTextTextID     /* title ref */
      }};
};
```

```
/* The static text for List window */
resource rTextForLETextBox2 (StatTextTextID) {
      "Static text is a simple but powerful control that lets you put "
      "predefined text in a window. The text is drawn with LETextBox2 "
      "so you can format the text any way you want: using special "
            TBStyleOutline
            "styles"
            TBStylePlain
            ", "
            TBFont
            TBVenice
            "\$00\$0E"
            "fonts"
            TBFont
            TBShaston
            "\$00\$08"
            ", "
            TBForeColor
            TBColor5
            "colors"
            TBForeColor
            TBColor0
            ","
            TBEndOfLine
            TBRightJust
            "indenting or justification."
            TBEndOfLine
            TBLeftJust
            TBEndOfLine
      "An additional feature of static text is substitutions. You may "
      "substitute up to ten strings into your ""static"" text, making "
      "it not so static. The ## and ** symbols are used to indicate "
      "substitutions."
      "You use ##n to indicate a built-in string. You use **n to "
      "indicate a particular string of your own. The SetCtlParamPtr "
      "call lets you set up the substitution array that should be "
      "used."
```

```
            TBEndOfLine
            TBEndOfLine
            "The built-in strings are "
            TBEndOfLine
            TBEndOfLine
            TBLeftMargin
            "\$20\$00"
            "##0 is #0"
            TBEndOfLine
            "##1 is ""#1"""
            TBEndOfLine
            "##2 is ""#2"""
            TBEndOfLine
            "##3 is ""#3"""
            TBEndOfLine
            "##4 is ""#4"""
            TBEndOfLine
            "##5 is ""#5"""
            TBEndOfLine
            "##6 is ""#6"""
            TBEndOfLine
};

/*--------------------------------------------------------------------*/
/*
/* LineEdit...
/*
/* The List window uses IDs in the $5000 range.
/*
/*--------------------------------------------------------------------*/

#define LineEditTextID              $5001
#define LineEdit1                   $5002
#define LineEdit2                   $5003
#define LineEdit3                   $5004
#define LineEdit4                   $5005
#define LineEdit5                   $5006
#define LineEdit6                   $5007
```

```
#define LELine1                         80
#define LELine2                         100
#define LELine3                         120
#define LELeft1                         10
#define LEWidth                         200
#define LEHeight                        13
#define LELeft2                         220

resource rWindParam1 (LineEditWindow) {
      fTitle+fMove+fZoom+fGrow+fBScroll+fRScroll+fClose,
                                /* frame bits */
      LineEditWindow,           /* title id */
      0,                        /* ref con */
      {0,0,0,0},                /* zoom rect */
      0,                        /* color table id */
      {0,0},                    /* origin */
      {400,640},                /* data size */
      {200,640},                /* max height-width */
      {1,1},                    /* scroll amount, hor,ver */
      {0,0},                    /* page amount */
      0,                        /* wInfo ref con */
      0,                        /* wInfo height */
      {50,50,120,260},          /* window position */
      infront,                  /* wPlane */
      LineEditWindow,           /* control ref */
      refIsResource*0x0100+resourceToResource
                                /* descriptor */
};


resource rpString (LineEditWindow) {
      "Line Edit Window"
};
```

```
resource rControlList (LineEditWindow) {
 {
            LineEditTextID,
            LineEdit6,
            LineEdit5,
            LineEdit4,
            LineEdit3,
            LineEdit2,
            LineEdit1
      };
};


resource rControlTemplate (LineEditTextID) {
      LineEditTextID,          /* control id */
      {2,4,52,460},            /* control rectangle */
      EditTextControl{{        /* control type */
            0x0000,            /* flag */
            fCtlCanBeTarget+fCtlWantEvents+fctlProcRefNotPtr,
                               /* more flags */
            0,                 /* ref con */
            fReadOnly+fDrawBounds+fTabSwitch,
                               /* text flags */
            {0xFFFF,0xFFFF,0xFFFF,0xFFFF},
                               /* indent rect */
            0xFFFFFFFF,        /* vert bar */
            0,                 /* vert amount */
            0,                 /* hor bar */
            0,                 /* hor amount */
            0,                 /* style ref */
            dataIsTextBlock+RefIsResource*8,
                               /* text descriptor */
            LineEditTextID,    /* text ref */
            0                  /* text length */

      }};
};
```

```
/* The static text for List window */
resource rText (LineEditTextID) {
      "The following six line edit fields are all defined in "
      "resources. "
      "All the typing, mouse tracking, and tabbing are handled by the "
      "Toolbox. The application does not have to do anything until it "
      "wants to read what is in the fields. Note that the fifth item "
      "is set up to work as a password item. The characters you type "
      "are not echoed, but they are collected correctly. "

};


resource rControlTemplate (LineEdit1) {
      0,                          /* control id */
      {LELine1,LELeft1,LELine1+LEHeight,LELeft1+LEWidth},
                                  /* control rectangle */
      EditLineControl{{           /* control type */
            0,                    /* flag */
            fctlProcRefNotPtr+RefIsResource,
                                  /* more flags */
            0,                    /* ref con */
            40,                   /* max length */
            LineEdit1             /* initial value ref */
      }};
};

resource rPString (LineEdit1) {
      "First Line Edit Item"
};
```

```
resource rControlTemplate (LineEdit2) {
      0,                        /* control id */
      {LELine1,LELeft2,LELine1+LEHeight,LELeft2+LEWidth},
                                /* control rectangle */
      EditLineControl{{         /* control type */
            0,                  /* flag */
            fctlProcRefNotPtr+RefIsResource,
                                /* more flags */
            0,                  /* ref con */
            40,                 /* max length */
            LineEdit2           /* initial value ref */
      }};
};


resource rPString (LineEdit2) {
      "Second Line Edit Item"
 };


resource rControlTemplate (LineEdit3) {
      0,                        /* control id */
      {LELine2,LELeft1,LELine2+LEHeight,LELeft1+LEWidth },
                                /* control rectangle */
      EditLineControl{{         /* control type */
            0,                  /* flag */
            fctlProcRefNotPtr+RefIsResource,
                                /* more flags */
            0,                  /* ref con */
            40,                 /* max length */
            LineEdit3           /* initial value ref */
      }};
};

resource rPString (LineEdit3) {
      "Third Line Edit Item"
 };
```

```
resource rControlTemplate (LineEdit4) {
      0,                        /* control id */
      {LELine2,LELeft2,LELine2+LEHeight,LELeft2+LEWidth},
                                /* control rectangle */
      EditLineControl{{         /* control type */
            0,                  /* flag */
            fctlProcRefNotPtr+RefIsResource,
                                /* more flags */
            0,                  /* ref con */
            40,                 /* max length */
            LineEdit4           /* initial value ref */
      }};
};

resource rPString (LineEdit4) {
      "Fourth Line Edit Item"
 };


resource rControlTemplate (LineEdit5) {
      0,                        /* control id */
      {LELine3,LELeft1,LELine3+LEHeight,LELeft1+LEWidth},
                                /* control rectangle */
      EditLineControl{{         /* control type */
            0,                  /* flag */
            fctlProcRefNotPtr+RefIsResource,
                                /* more flags */
            0,                  /* ref con */
            40+$8000,           /* max length (password field) */
            LineEdit5           /* initial value ref */
      }};
};


resource rPString (LineEdit5) {
      "Fifth Line Edit Item"
 };
```

```
resource rControlTemplate (LineEdit6) {
      0,                        /* control id */
 {LELine3,LELeft2,LELine3+LEHeight,LELeft2+LEWidth},
                                /* control rectangle */
      EditLineControl{{         /* control type */
            0,                  /* flag */
            fctlProcRefNotPtr+RefIsResource,
                                /* more flags */
            0,                  /* ref con */
            40,                 /* max length */
            LineEdit6           /* initial value ref */
      }};
};

resource rPString (LineEdit6) {
      "Sixth Line Edit Item"
 };

/*-----------------------------------------------------------------------*/
/*
/* Pictures...
/*
/* The List window uses IDs in the $6000 range.
/*
/*-----------------------------------------------------------------------*/

#define PictureTextID   $6001
#define Pic1            $6002
```

```
resource rWindParam1 (PictureWindow) {
      fTitle+fMove+fZoom+fGrow+fBScroll+fRScroll+fClose,
                              /* frame bits */
      PictureWindow,          /* title id */
      0,                      /* ref con */
      {0,0,0,0},              /* zoom rect */
      0,                      /* color table id */
      {0,0},                  /* origin */
      {400,640},              /* data size */
      {200,640},              /* max height-width */
      {1,1},                  /* scroll amount, hor,ver */
      {0,0},                  /* page amount */
      0,                      /* wInfo ref con */
      0,                      /* wInfo height */
      {50,50,120,260},        /* window position */
      infront,                /* wPlane */
      PictureWindow,          /* control ref */
      refIsResource*0x0100+resourceToResource
                              /* descriptor */
};


resource rpString (PictureWindow) {
      "Pictures Window"
};


resource rControlList (PictureWindow) {
      {
            PictureTextID,
            Pic1
      };
};
```

```
/* Template for static text in main window */
resource rControlTemplate (PictureTextID) {
      PictureTextID,          /* control id */
      {2,4,48,460},           /* control rectangle */
      StatTextControl{{       /* control type */
            ctlInactive,      /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                              /* more flags */
            0,                /* ref con */
            PictureTextID     /* title ref */
      }};
};

/* The static text for List window */
resource rTextForLETextBox2 (PictureTextID) {
      "You can also make picture controls. Pictures are collections of "
      "QuickDraw commands that are all drawn at once. They can contain "
      "most any drawing command including text, color, and special "
      "fonts."
};


resource rControlTemplate (Pic1) {
      Pic1,                   /* control id */
      {50,2,150,202},         /* control rectangle */
      PictureControl{{        /* control type */
            ctlInactive,      /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                              /* more flags */
            0,                /* ref con */
            Pic1              /* title ref */
      }};
};
```

```
data rPicture (Pic1) {
$"80 00 00 00 00 00 8F 00 38 01 11 82 01 00 0A 00" /* A.....e.8..C.... */
$"01 C0 01 C0 FF 3F FF 3F 51 00 05 00 0A 00 8A 00" /* .?.?.?.?Q.....a. */
$"2E 01 53 00 0A 00 14 00 85 00 24 01 53 00 0F 00" /* ..S.....O.$.S... */
$"1E 00 80 00 1A 01 53 00 14 00 28 00 7B 00 10 01" /* ..A...S...(.{... */
$"53 00 19 00 32 00 76 00 06 01 53 00 1E 00 3C 00" /* S...2.v...S...<. */
$"71 00 FC 00 53 00 23 00 46 00 6C 00 F2 00 53 00" /* q...S.#.F.l...S. */
$"28 00 50 00 67 00 E8 00 53 00 2D 00 5A 00 62 00" /* (.P.g...S.-.Z.b. */
$"DE 00"                                           /* .. */
};


/*--------------------------------------------------------------------*/
/*
/* PopUps...
/*
/* The List window uses IDs in the $7000 range.
/*
/*--------------------------------------------------------------------*/

#define PopUpTextID                 $7001
#define PopUp1                      $7100
#define PopUp2                      $7200
#define PopUp1Item1                 $7101
#define PopUp1Item2                 $7102
#define PopUp1Item3                 $7103
#define PopUp2Item1                 $7201
#define PopUp2Item2                 $7202
#define PopUp2Item3                 $7203
#define PopUp2Item4                 $7204
#define PopUp2Item5                 $7205
#define PopUp2Item6                 $7206
#define PopUp2Item7                 $7207
#define PopUp2Item8                 $7208
#define PopUp2Item9                 $7209
```

```
resource rWindParam1 (PopUpWindow) {
      fTitle+fMove+fZoom+fGrow+fBScroll+fRScroll+fClose,
                              /* frame bits */
      PopUpWindow,            /* title id */
      0,                      /* ref con */
      {0,0,0,0},              /* zoom rect */
      0,                      /* color table id */
      {0,0},                  /* origin */
      {400,640},              /* data size */
      {200,640},              /* max height-width */
      {1,1},                  /* scroll amount, hor,ver */
      {0,0},                  /* page amount */
      0,                      /* wInfo ref con */
      0,                      /* wInfo height */
      {50,50,120,260},        /* window position */
      infront,                /* wPlane */
      PopUpWindow,            /* control ref */
      refIsResource*0x0100+resourceToResource
                              /* descriptor */
};


resource rpString (PopUpWindow) {
      "PopUps Window"
};


resource rControlList (PopUpWindow) {
      {
            PopUpTextID,
            PopUp1,
            PopUp2
      };
};
```

```
/* Template for static text in main window */
resource rControlTemplate (PopUpTextID) {
      PopUpTextID,            /* control id */
      {2,4,48,460},           /* control rectangle */
      StatTextControl{{       /* control type */
            ctlInactive,      /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                              /* more flags */
            0,                /* ref con */
            PopUpTextID       /* title ref */
      }};
};

/* The static text for List window */
resource rTextForLETextBox2 (PopUpTextID) {
      "This window contains two pop-up menus. The first menu has three "
      "items and is constrained to pop up inside the window. The "
      "second has nine items and can pop up outside the window. The "
      "first pop-up is a type 1 pop-up, and the second is a type 2. "
};



resource rControlTemplate (PopUp1) {
      PopUpTextID,            /* control id */
      {50,50,0,0},            /* control rectangle */
      PopUpControl{{          /* control type */
            fInWindowOnly,    /* flags */
            fctlProcRefNotPtr+RefIsResource,
                              /* more flags */
            0,                /* ref con */
            0,                /* title width */
            PopUp1,           /* menu ref */
            PopUp1Item1       /* initial value */
      }}
};
```

```
resource rMenu (PopUp1) {
      PopUp1,                    /*id of menu */
 RefIsResource*MenuTitleRefShift+RefIsResource*ItemRefShift+fAllowCache,
                                 /* menu flags */
      PopUp1,                    /* id of title string */
      { PopUp1Item1,PopUp1Item2,PopUp1Item3 };
                                 /* id's of items */
};

resource rPString (PopUp1,noCrossBank) {
      "Pop-up One "
};


resource rMenuItem (PopUp1Item1) {
      PopUp1Item1,
      "","",
      0,
      RefIsResource*ItemTitleRefShift+fXOR,
      PopUp1Item1
};
resource rPString (PopUp1Item1,noCrossBank) {
      "Pop-up One: Item 1"
};


resource rMenuItem (PopUp1Item2) {
      PopUp1Item2,
      "","",
      0,
      RefIsResource*ItemTitleRefShift+fXOR,
      PopUp1Item2
};
resource rPString (PopUp1Item2,noCrossBank) {
      "Pop-up One: Item 2"
};
```

```
resource rMenuItem (PopUp1Item3) {
      PopUp1Item3,
      "","",
      0,
      RefIsResource*ItemTitleRefShift+fXOR,
      PopUp1Item3
};
resource rPString (PopUp1Item3,noCrossBank) {
      "Pop-up One: Item 3"
};



resource rControlTemplate (PopUp2) {
      PopUp2,                 /* control id */
      {80,50,0,0},            /* control rectangle */
      PopUpControl{{          /* control type */
            fType2PopUp,      /* flags */
            fctlProcRefNotPtr+RefIsResource,
                              /* more flags */
            0,                /* ref con */
            0,                /* title width */
            PopUp2,           /* menu ref */
            PopUp2Item1       /* initial value */
      }}
};
```

```
resource rMenu (PopUp2) {
      PopUp2,                    /* id of menu */
 RefIsResource*MenuTitleRefShift+RefIsResource*ItemRefShift+fAllowCache,
                                 /* menu flags */
      PopUp2,                    /* id of title string */
      {     PopUp2Item1,
            PopUp2Item2,
            PopUp2Item3,
            PopUp2Item4,
            Popup2Item5,
            Popup2Item6,
            Popup2Item7,
            Popup2Item8,
            Popup2Item9
      };                         /* id's of items */
};
resource rPString (PopUp2,noCrossBank) {
      "Pop-up Two "
};


resource rMenuItem (PopUp2Item1) {
      PopUp2Item1,
      "","",
      0,
      RefIsResource*ItemTitleRefShift+fXOR,
      PopUp2Item1
};
resource rPString (PopUp2Item1,noCrossBank) {
      "Pop-up Two: Item 1"
};
```

```
resource rMenuItem (PopUp2Item2) {
      PopUp2Item2,
      "","",
      0,
      RefIsResource*ItemTitleRefShift+fXOR,
      PopUp2Item2
};
resource rPString (PopUp2Item2,noCrossBank) {
      "Pop-up Two: Item 2"
};


resource rMenuItem (PopUp2Item3) {
      PopUp2Item3,
      "","",
      0,
      RefIsResource*ItemTitleRefShift+fXOR,
      PopUp2Item3
};
resource rPString (PopUp2Item3,noCrossBank) {
      "Pop-up Two: Item 3"
};

resource rMenuItem (PopUp2Item4) {
      PopUp2Item4,
      "","",
      0,
      RefIsResource*ItemTitleRefShift+fXOR,
      PopUp2Item4
};
resource rPString (PopUp2Item4,noCrossBank) {
      "Pop-up Two: Item 4"
};
```

```
resource rMenuItem (PopUp2Item5) {
      PopUp2Item5,
      "","",
      0,
      RefIsResource*ItemTitleRefShift+fXOR,
      PopUp2Item5
};
resource rPString (PopUp2Item5,noCrossBank) {
      "Pop-up Two: Item 5"
};


resource rMenuItem (PopUp2Item6) {
      PopUp2Item6,
      "","",
      0,
      RefIsResource*ItemTitleRefShift+fXOR,
      PopUp2Item6
};
resource rPString (PopUp2Item6,noCrossBank) {
      "Pop-up Two: Item 6"
};

resource rMenuItem (PopUp2Item7) {
      PopUp2Item7,
      "","",
      0,
      RefIsResource*ItemTitleRefShift+fXOR,
      PopUp2Item7
};
resource rPString (PopUp2Item7,noCrossBank) {
      "Pop-up Two: Item 7"
};
```

```
resource rMenuItem (PopUp2Item8) {
      PopUp2Item8,
      "","",
      0,
      RefIsResource*ItemTitleRefShift+fXOR,
      PopUp2Item8
};
resource rPString (PopUp2Item8,noCrossBank) {
      "Pop-up Two: Item 8"
};

resource rMenuItem (PopUp2Item9) {
      PopUp2Item9,
      "","",
      0,
      RefIsResource*ItemTitleRefShift+fXOR,
      PopUp2Item9
};
resource rPString (PopUp2Item9,noCrossBank) {
      "Pop-up Two: Item 9"
};


/*-----------------------------------------------------------------*/
/*
/* TextEdits...
/*
/* The TextEdit window uses IDs in the $8000 range.
/*
/*-----------------------------------------------------------------*/

#define TextEditTextID              $8001
#define TextEdit1                   $8002
#define TextEdit2                   $8003
```

```
resource rWindParam1 (TextEditWindow) {
      fTitle+fMove+fZoom+fGrow+fBScroll+fRScroll+fClose,
                              /* frame bits */
      TextEditWindow,         /* title id */
      0,                      /* ref con */
      {0,0,0,0}    ,          /* zoom rect */
      0,                      /* color table id */
      {0,0},                  /* origin */
      {400,640},              /* data size */
      {200,640},              /* max height-width */
      {1,1},                  /* scroll amount, hor,ver */
      {0,0},                  /* page amount */
      0,                      /* wInfo ref con */
      0,                      /* wInfo height */
      {50,50,120,260},        /* window position */
      infront,                /* wPlane */
      TextEditWindow,         /* control ref */
      refIsResource*0x0100+resourceToResource
                              /* descriptor */
};


resource rpString (TextEditWindow) {
      "TextEdits Window"
};


resource rControlList (TextEditWindow) {
      {
            TextEditTextID,
            TextEdit2,
            TextEdit1,
            0
      };
};
```

```
/* Template for static text in main window */
resource rControlTemplate (TextEditTextID) {
      TextEditTextID,         /* control id */
      {2,4,48,460},           /* control rectangle */
      StatTextControl{{       /* control type */
            ctlInactive,      /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                              /* more flags */
            0,                /* ref con */
            TextEditTextID    /* title ref */
      }};
};


/* The static text for List window */
resource rTextForLETextBox2 (TextEditTextID) {
      "Two text edit fields."
};
```

```
resource rControlTemplate (TextEdit1) {
      TextEdit1,              /* control id */
      {50,4,100,460},         /* control rectangle */
      EditTextControl{{       /* control type */
            0x0000,           /* flag */
            fCtlCanBeTarget+fCtlWantEvents+fctlProcRefNotPtr,
                              /* more flags */
            0,                /* ref con */
            fSmartCutPaste+fTabSwitch+fDrawBounds,
                              /* text flags */
            {0xFFFF,0xFFFF,0xFFFF,0xFFFF},
                              /* indent rect */
            0xFFFFFFFF,       /* vert bar */
            0,                /* vert amount */
            0,                /* hor bar */
            0,                /* hor amount */
            0,                /* style ref */
            dataIsPString+RefIsResource*8,
                              /* text descriptor */
            TextEdit1,        /* text ref */
            0                 /* text length */

      }};
};


resource rPString (TextEdit1) {
      "This is a PString that you can edit."
};

resource rCString (TextEdit1) {
      "This is a CString that you can edit."
};

resource rText (TextEdit1) {
      "This is a text block that you can edit."
};
```

```
resource rControlTemplate (TextEdit2) {
      TextEdit1,              /* control id */
      {110,4,150,460},        /* control rectangle */
      EditTextControl{{       /* control type */
            0x0000,           /* flag */
            fCtlCanBeTarget+fCtlWantEvents+fctlProcRefNotPtr,
                              /* more flags */
            0,                /* ref con */
            fSmartCutPaste+fTabSwitch+fDrawBounds,
                              /* text flags */
            {0xFFFF,0xFFFF,0xFFFF,0xFFFF},
                              /* indent rect */
            0xFFFFFFFF,       /* vert bar */
            0,                /* vert amount */
            0,                /* hor bar */
            0,                /* hor amount */
            0,                /* style ref */
            dataIsTextBlock+RefIsResource*8,
                              /* text descriptor */
            TextEdit2,        /* text ref */
            0                 /* text length */


      }};
};


/* The static text for List window */
resource rText (TextEdit2) {
      "More text. Will it tab?"
};
```

```
/*------------------------------------------------------------------*/
/*
/* Lists...
/*
/* The List window uses IDs in the $9000 range.
/*
/*------------------------------------------------------------------*/

resource rWindParam1 (ListWindow) {
      fTitle+fMove+fZoom+fGrow+fBScroll+fRScroll+fClose,
                              /* frame bits */
      ListWindow,             /* title id */
      0,                      /* ref con */
      {0,0,0,0},              /* zoom rect */
      0,                      /* color table id */
      {0,0},                  /* origin */
      {400,640},              /* data size */
      {200,640},              /* max height-width */
      {1,1},                  /* scroll amount, hor,ver */
      {0,0},                  /* page amount */
      0,                      /* wInfo ref con */
      0,                      /* wInfo height */
      {50,50,120,260},        /* window position */
      infront,                /* wPlane */
      ListWindow,             /* control ref */
      refIsResource*0x0100+resourceToResource
                              /* descriptor */
};


resource rpString (ListWindow) {
      "Lists Window"
};


#define ListID                        $9000
#define ListTextID                    $9001
```

```
/* List of all controls in main window */

resource rControlList (ListWindow) {
      {
            ListID,
            ListTextID,
            0
      };
};

/* Template for static text in main window */
resource rControlTemplate (ListTextID) {
      ListTextID,                /* control id */
      {2,4,48,460},              /* control rectangle */
      StatTextControl{{          /* control type */
            ctlInactive,         /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                                 /* more flags */
            0,                   /* ref con */
            ListTextID,          /* title ref */
            0,                   /* text size (not used) */
      }};
};

/* The static text for List window */
resource rTextForLETextBox2 (ListTextID) {
      "This list is defined and contained entirely in resources. "
      "The strings in the list are also resources."
};
```

```
resource rControlTemplate (ListID) {
      ListID,                    /* control id */
      {50,50,152,350},           /* list rectangle */
      ListControl{{              /* list type */
            0,                   /* flag */
            fCtlProcRefNotPtr+RefIsResource,
                                 /* more flags */
            0,                   /* ref con */
            16,                  /* num members in list */
            0,                   /* list view (let list mgr calc) */
            0,                   /* list type */
            1,                   /* list start (start at top ) */
            10,                  /* ListMemHeight */
            5,                   /* ListMemSize */
            ListID               /* ListRef (id of list record) */
      }};
};

resource rListRef (ListID) {
      { 0x9001,memNormal,
        0x9002,memSelected,
        0x9003,memDisabled,
        0x9004,memNormal,
        0x9005,memNormal,
        0x9006,memNormal,
        0x9007,memNormal,
        0x9008,memNormal,
        0x9009,memNormal,
        0x900A,memNormal,
        0x900B,memNormal,
        0x900C,memNormal,
        0x900D,memNormal,
        0x900E,memNormal,
        0x900F,memNormal,
        0x9010,memNormal
      };
};


resource rpString (0x9001) {
      "Item One"
};
```

```
resource rpString (0x9002) {
      "Item Two"
};
resource rpString (0x9003) {
      "Item Three"
};
resource rpString (0x9004) {
      "Item Four"
};
resource rpString (0x9005) {
      "Item Five"
};
resource rpString (0x9006) {
      "Item Six"
};
resource rpString (0x9007) {
      "Item Seven"
};
resource rpString (0x9008) {
      "Item Eight"
};
resource rpString (0x9009) {
      "Item Nine"
};
resource rpString (0x900A) {
      "Item Ten"
};
resource rpString (0x900B) {
      "Item Eleven"
};
resource rpString (0x900C) {
      "Item Twelve"
};
resource rpString (0x900D) {
      "Item Thirteen"
};
resource rpString (0x900E) {
      "Item Fourteen"
};
resource rpString (0x900F) {
      "Item Fifteen"
};
```

```
resource rpString (0x9010) {
      "Item Sixteen"
};
```

```
/*****************************************************************/
/*
/* Menus
/*
/*****************************************************************/

#define AppleMenuID           $1100
#define FileMenuID            $1200
#define EditMenuID            $1300

#define AboutID               $1101

#define CloseID                 255
#define QuitID                $1202


#define UndoID                  250
#define CutID                   251
#define CopyID                  252
#define PasteID                 253
#define ClearID                 254


resource rMenuBar ($1000) {
      {
            AppleMenuID,
            FileMenuID,
            EditMenuID,
      };
};
```

```
resource rMenu (AppleMenuID) {
      AppleMenuID,
      RefIsResource*MenuTitleRefShift+RefIsResource*ItemRefShift+
            fAllowCache,
      AppleMenuID,
      { AboutID };
};

resource rMenu (FileMenuID) {
      FileMenuID,
      RefIsResource*MenuTitleRefShift+RefIsResource*ItemRefShift+
            fAllowCache,
      FileMenuID,
      { CloseID,
      QuitID };
};

resource rMenu (EditMenuID) {
      EditMenuID,
      RefIsResource*MenuTitleRefShift+RefIsResource*ItemRefShift+
            fAllowCache,
      EditMenuID,
      {
            UndoID,
            CutID,
            CopyID,
            PasteID,
            ClearID
      };
};

resource rMenuItem (AboutID) {
      AboutID,
      "","",
      0,
      RefIsResource*ItemTitleRefShift+fDivider,
      AboutID
};
```

```
resource rMenuItem (UndoID) {
      UndoID,
      "","",
      0,
      RefIsResource*ItemTitleRefShift,
      UndoID
};

resource rMenuItem (CutID) {
      CutID,
      "X","x",
      0,
      RefIsResource*ItemTitleRefShift,
      CutID
};

resource rMenuItem (CopyID) {
      CopyID,
      "C","c",
      0,
      RefIsResource*ItemTitleRefShift,
      CopyID
};

resource rMenuItem (PasteID) {
      PasteID,
      "V","v",
      0,
      RefIsResource*ItemTitleRefShift,
      PasteID
};

resource rMenuItem (ClearID) {
      ClearID,
      "","",
      0,
      RefIsResource*ItemTitleRefShift,
      ClearID
};
```

```
resource rMenuItem (CloseID) {
      CloseID,
      "W","w",
      0,
      RefIsResource*ItemTitleRefShift,
      CloseID
};

resource rMenuItem (QuitID) {
      QuitID,
      "Q","q",
      0,
      RefIsResource*ItemTitleRefShift,
      QuitID
};


resource rPString (AppleMenuID,noCrossBank) {
      "@"
      };
resource rPString (FileMenuID,noCrossBank) {
      "File"
      };
resource rPString (EditMenuID,noCrossBank) {
      "Edit"
      };
resource rPString (AboutID,noCrossBank) {
      "About BusyBox..."
      };
resource rPString (CloseID,noCrossBank) {
      "Close"
      };

resource rPString (UndoID,noCrossBank) {
      "Undo"
      };
resource rPString (CutID,noCrossBank) {
      "Cut"
      };
resource rPString (CopyID,noCrossBank) {
      "Copy"
      };
```

```
resource rPString (PasteID,noCrossBank) {
      "Paste"
      };
resource rPString (ClearID,noCrossBank) {
      "Clear"
      };
resource rPString (QuitID,noCrossBank) {
      "Quit"
      };



/*--------------------------------------------------------------------*/
/*
/* Program...
/*
/* The Program windows use IDs in the $A000 range.
/*
/*--------------------------------------------------------------------*/

#define Program1              $A001
#define Program2              $A002
#define Program3              $A003
#define Program4              $A004
#define Program5              $A005
#define Program6              $A006
```

```
resource rWindParam1 (Prog1Window) {
      fTitle+fMove+fZoom+fClose,
                              /* frame bits */
      Prog1Window,            /* title id */
      0,                      /* ref con */
      {0,0,0,0},              /* zoom rect */
      0,                      /* color table id */
      {0,0},                  /* origin */
      {400,640},              /* data size */
      {200,640},              /* max height-width */
      {1,1},                  /* scroll amount, hor,ver */
      {0,0},                  /* page amount */
      0,                      /* wInfo ref con */
      0,                      /* wInfo height */
      {30,4,180,500},         /* window position */
      infront,                /* wPlane */
      Program1,               /* control ref */
      refIsResource*0x0100+refIsResource
                              /* descriptor */
};


resource rpString (Prog1Window) {
      "Main Program"
};


resource rControlList (Program1) {
      {
            Program1,
      };
};
```

```
resource rControlTemplate (Program1) {
      TextEdit1,              /* control id */
      {0,0,0,0},              /* control rectangle */
      EditTextControl{{       /* control type */
            0x0000,           /* flag */
     fCtlCanBeTarget+fCtlWantEvents+fctlProcRefNotPtr+fctlTellAboutSize,
                              /* more flags */
            0,                /* ref con */
            fReadOnly+fNoWordWrap,
                              /* text flags */
            {0xFFFF,0xFFFF,0xFFFF,0xFFFF},
                              /* indent rect */
            0xFFFFFFFF,       /* vert bar */
            0,                /* vert amount */
            0,                /* hor bar */
            0,                /* hor amount */
            0,                /* style ref */
            dataIsTextBlock+RefIsResource*8,
                              /* text descriptor */
            Program1,         /* text ref */
            0                 /* text length */



      }};
};

read rText (Program1) "busy.p";
```

```
resource rWindParam1 (Prog2Window) {
      fTitle+fMove+fZoom+fClose,
                              /* frame bits */
      Prog2Window,            /* title id */
      0,                      /* ref con */
      {0,0,0,0},              /* zoom rect */
      0,                      /* color table id */
      {0,0},                  /* origin */
      {400,640},              /* data size */
      {200,640},              /* max height-width */
      {1,1},                  /* scroll amount, hor,ver */
      {0,0},                  /* page amount */
      0,                      /* wInfo ref con */
      0,                      /* wInfo height */
      {30,4,180,500},         /* window position */
      infront,                /* wPlane */
      Program2,               /* control ref */
      refIsResource*0x0100+refIsResource
                              /* descriptor */
};


resource rpString (Prog2Window) {
      "Event Unit"
};


resource rControlList (Program2) {
      {
      Program2,
      };
};
```

```
resource rControlTemplate (Program2) {
      Program2,                  /* control id */
      {0,0,0,0},                 /* control rectangle */
      EditTextControl{{          /* control type */
            0x0000,              /* flag */
     fCtlCanBeTarget+fCtlWantEvents+fctlProcRefNotPtr+fctlTellAboutSize,
                                 /* more flags */
            0,                   /* ref con */
            fReadOnly+fNoWordWrap,
                                 /* text flags */
            {0xFFFF,0xFFFF,0xFFFF,0xFFFF},
                                 /* indent rect */
            0xFFFFFFFF,          /* vert bar */
            0,                   /* vert amount */
            0,                   /* hor bar */
            0,                   /* hor amount */
            0,                   /* style ref */
            dataIsTextBlock+RefIsResource*8,
                                 /* text descriptor */
            Program2,            /* text ref */
            0                    /* text length */

      }};
};

read rText (Program2) "uevent.p";
```

```
resource rWindParam1 (Prog3Window) {
      fTitle+fMove+fZoom+fClose,
                              /* frame bits */
      Prog3Window,            /* title id */
      0,                      /* ref con */
      {0,0,0,0},              /* zoom rect */
      0,                      /* color table id */
      {0,0},                  /* origin */
      {400,640},              /* data size */
      {200,640},              /* max height-width */
      {1,1},                  /* scroll amount, hor,ver */
      {0,0},                  /* page amount */
      0,                      /* wInfo ref con */
      0,                      /* wInfo height */
      {30,4,180,500},         /* window position */
      infront,                /* wPlane */
      Program3,               /* control ref */
      refIsResource*0x0100+refIsResource
                              /* descriptor */
};


resource rpString (Prog3Window) {
      "Menu Unit"
};


resource rControlList (Program3) {
      {
      Program3,
      };
};
```

```
resource rControlTemplate (Program3) {
      Program3,                  /* control id */
      {0,0,0,0},                 /* control rectangle */
      EditTextControl{{          /* control type */
                  0x0000,        /* flag */
     fCtlCanBeTarget+fCtlWantEvents+fctlProcRefNotPtr+fctlTellAboutSize,
                                 /* more flags */
            0,                   /* ref con */
            fReadOnly+fNoWordWrap,
                                 /* text flags */
            {0xFFFF,0xFFFF,0xFFFF,0xFFFF},
                                 /* indent rect */
            0xFFFFFFFF,          /* vert bar */
            0,                   /* vert amount */
            0,                   /* hor bar */
            0,                   /* hor amount */
            0,                   /* style ref */
            dataIsTextBlock+RefIsResource*8,
                                 /* text descriptor */
            Program3,            /* text ref */
            0                    /* text length */

      }};
};

read rText (Program3) "umenu.p";
```

```
resource rWindParam1 (Prog4Window) {
      fTitle+fMove+fZoom+fClose,
                              /* frame bits */
      Prog4Window,            /* title id */
      0,                      /* ref con */
      {0,0,0,0},              /* zoom rect */
      0,                      /* color table id */
      {0,0},                  /* origin */
      {400,640},              /* data size */
      {200,640},              /* max height-width */
      {1,1},                  /* scroll amount, hor,ver */
      {0,0},                  /* page amount */
      0,                      /* wInfo ref con */
      0,                      /* wInfo height */
      {30,4,180,500},         /* window position */
      infront,                /* wPlane */
      Program4,               /* control ref */
      refIsResource*0x0100+refIsResource
                              /* descriptor */
};


resource rpString (Prog4Window) {
      "Window Unit"
};


resource rControlList (Program4) {
      {
      Program4,
      };
};
```

```
resource rControlTemplate (Program4) {
      Program4,                  /* control id */
      {0,0,0,0},                 /* control rectangle */
      EditTextControl{{          /* control type */
            0x0000,              /* flag */
     fCtlCanBeTarget+fCtlWantEvents+fctlProcRefNotPtr+fctlTellAboutSize,
                                 /* more flags */
            0,                   /* ref con */
            fReadOnly+fNoWordWrap,
                                 /* text flags */
            {0xFFFF,0xFFFF,0xFFFF,0xFFFF},
                                 /* indent rect */
            0xFFFFFFFF,          /* vert bar */
            0,                   /* vert amount */
            0,                   /* hor bar */
            0,                   /* hor amount */
            0,                   /* style ref */
            dataIsTextBlock+RefIsResource*8,
                                 /* text descriptor */
            Program4,            /* text ref */
            0                    /* text length */

      }};
};

read rText (Program4) "uwindow.p";
```

```
resource rWindParam1 (Prog5Window) {
      fTitle+fMove+fZoom+fClose,
                                /* frame bits */
      Prog5Window,              /* title id */
      0,                        /* ref con */
      {0,0,0,0},                /* zoom rect */
      0,                        /* color table id */
      {0,0},                    /* origin */
      {400,640},                /* data size */
      {200,640},                /* max height-width */
      {1,1},                    /* scroll amount, hor,ver */
      {0,0},                    /* page amount */
      0,                        /* wInfo ref con */
      0,                        /* wInfo height */
      {30,4,180,500},           /* window position */
      infront,                  /* wPlane */
      Program5,                 /* control ref */
      refIsResource*0x0100+refIsResource
                                /* descriptor */
};


resource rpString (Prog5Window) {
      "Utility Unit"
};


resource rControlList (Program5) {
      {
      Program5,
      };
};
```

```
resource rControlTemplate (Program5) {
      Program5,                  /* control id */
      {0,0,0,0},                 /* control rectangle */
      EditTextControl{{          /* control type */
            0x0000,              /* flag */
     fCtlCanBeTarget+fCtlWantEvents+fctlProcRefNotPtr+fctlTellAboutSize,
                                 /* more flags */
            0,                   /* ref con */
            fReadOnly+fNoWordWrap,
                                 /* text flags */
            {0xFFFF,0xFFFF,0xFFFF,0xFFFF},
                                 /* indent rect */
            0xFFFFFFFF,          /* vert bar */
            0,                   /* vert amount */
            0,                   /* hor bar */
            0,                   /* hor amount */
            0,                   /* style ref */
            dataIsTextBlock+RefIsResource*8,
                                 /* text descriptor */
            Program5,            /* text ref */
            0                    /* text length */

      }};
};

read rText (Program5) "uutils.p";
```

```
resource rWindParam1 (Prog6Window) {
      fTitle+fMove+fZoom+fClose,
                              /* frame bits */
      Prog6Window,            /* title id */
      0,                      /* ref con */
      {0,0,0,0},              /* zoom rect */
      0,                      /* color table id */
      {0,0},                  /* origin */
      {400,640},              /* data size */
      {200,640},              /* max height-width */
      {1,1},                  /* scroll amount, hor,ver */
      {0,0},                  /* page amount */
      0,                      /* wInfo ref con */
      0,                      /* wInfo height */
      {30,4,180,500},         /* window position */
      infront,                /* wPlane */
      Program6,               /* control ref */
      refIsResource*0x0100+refIsResource
                              /* descriptor */
};


resource rpString (Prog6Window) {
      "Globals Unit"
};


resource rControlList (Program6) {
      {
      Program6,
      };
};
```

```
resource rControlTemplate (Program6) {
      Program6,    /* control ID */
      {0,0,0,0},                /* control rectangle */
      EditTextControl{{         /* control type */
            0x0000,             /* flag */
     fCtlCanBeTarget+fCtlWantEvents+fctlProcRefNotPtr+fctlTellAboutSize,
                                 /* more flags */
            0,                  /* ref con */
            fReadOnly+fNoWordWrap,
                                 /* text flags */
            {0xFFFF,0xFFFF,0xFFFF,0xFFFF},
                                 /* indent rect */
            0xFFFFFFFF,         /* vert bar */
            0,                  /* vert amount */
            0,                  /* hor bar */
            0,                  /* hor amount */
            0,                  /* style ref */
            dataIsTextBlock+RefIsResource*8,
                                 /* text descriptor */
            Program6,           /* text ref */
            0                   /* text length */

      }};
};

read rText (Program6) "uglobals.p";
```

The `uEvent.p` module

This section contains the source code for the `uEvent.p` module, which implements the main event loop for the BusyBox program. This code was written in Pascal.

```
{*************************************************************************}
{*
{* BusyBox uEvent (interface)
{*
{* Copyright (c)
{* Apple Computer, Inc. 1986-1990
{* All Rights Reserved.
{*
{* This file contains the interface to the code which implements the
{* main event loop used by the BusyBox program.
{*
{*************************************************************************}

UNIT uEvent;

INTERFACE

USES
      types,
      GSOS,
      memory,
      locator,
      quickdraw,
      events,
      resources,
      controls,
      windows,
      lineedit,
      dialogs,
      menus,
      stdfile,
      IntMath,
      Fonts,
      Desk,
```

```
      uGlobals,
      uUtils,
      uWindow,
      uMenu;


procedure MainEvent;     {Main event handling loop, which repeats }
                         { until Quit.}

IMPLEMENTATION

{$R-}

var
      LastWindow  : GrafPortPtr;
                              { This private global is used in }
                              { CheckFrontW to prevent extra work when }
                              { the windows have not changed. It is }
                              { initialized at the beginning of }
                              { MainEvent. }
```

```
{*****************************************************************}
{
{ DoControls
{
{ This procedure is called when an inControl message is returned
{ by TaskMaster.
{
{ When this routine gets control, the ID of the control that was
{ selected is in TaskData4.   The control handle is in TaskData2,
{ and the part code is in TaskData3.
{
{*****************************************************************}
procedure DoControls;
var
      TheID : integer;
begin
      TheID := Event.wmTaskData4;

      if (ButButtonsID <= TheID) and (TheID <= Prog6ID) then
            OpenThisWindow(TheID);
end;



{*****************************************************************}
{
{ CheckFrontW
{
{ This routine checks the front window to see if any changes need
{ to be made to the menu items.
{
{ We do this so that the edit items are active only when a desk
{ accessory is active.
{
{*****************************************************************}
procedure CheckFrontW;
```

```
var
      theWindow   : GrafPortPtr;

begin       {of CheckFrontW}
      { Get the front window into local storage.}
      theWindow := FrontWindow;

      { If the LastWindow is this window, we are all set. }
      if theWindow = lastWindow then Exit(CheckFrontW);

      { If there are no windows, everything should be disabled. }
      if theWindow = nil then
            begin
                  SetMenuFlag ($0080,EditMenuID);
                  DrawMenuBar;
            end

      else
            begin
                  { Otherwise we look at the window and see what to do. }

                  if GetSysWFlag (theWindow) <> false then
                        begin       {Set up for DA windows.}
                              SetMenuFlag ($FF7F,EditMenuID);
                              DrawMenuBar;
                        end
                  else
                        begin       {Set up for our windows.}
                              SetMenuFlag ($0080,EditMenuID);
                              DrawMenuBar;
                        end;
            end;

            { Remember this for next time. }
            lastWindow := theWindow;
end;  {of CheckFrontW}
```

```
{*********************************************************************}
{
{ MainEvent
{
{ This is the main part of the program.   The program cycles in this
{ loop until the user chooses Select.
{
{*********************************************************************}
procedure MainEvent;


var
      code : integer;


begin       {of MainEvent}
      Event.wmTaskMask := $001FFFFF;      { Allow TaskMaster to do }
                                          { everything. }
      Done  := false;                     { Done flag will be set by }
                                          { Quit item. }
      LastWindow  := NIL;                 { Init this for CheckFrontW. }

      repeat
            CheckFrontW;
            code := TaskMaster ($FFFF,Event);
            case code of
                  wInGoAway   : DoCloseTop;
                  wInSpecial,
                  wInMenuBar  : DoMenu;
                  wInControl  : DoControls;
            end;
      until Done;
end;        {of MainEvent}


END.
```

The `uGlobals.p` module

This section contains the source code for `uGlobals.p`. This Pascal module defines the global variables for the BusyBox program.

```
{************************************************************************}
{*
{*
{* BusyBox Globals (interface)
{*
{* Copyright (c)
{* Apple Computer, Inc. 1986-1990
{* All Rights Reserved.
{*
{* This file contains the global variables used by the BusyBox
{* program.
{*
{************************************************************************}

UNIT uGlobals;

INTERFACE

USES
      types,
      locator,
      memory,
      quickdraw,
      intMath,
      events,
      controls,
      windows,
      lineedit,
      dialogs,
      STDFile;
```

```
const

            AppleMenuID = $1100;
                  AboutItem   = $1101;
            FileMenuID  = $1200;
                  CloseItem   = 255;      {For DA's}
                  QuitItem    = $1202;
            EditMenuID  = $1300;
                  UndoItem    = 250;      {For DA's}
                  CutItem     = 251;      {For DA's}
                  CopyItem    = 252;      {For DA's}
                  PasteItem   = 253;      {For DA's}
                  ClearItem   = 254;      {For DA's}

            NumWindows  = 14;
            NumWindowsMin1 = 13;

            ButButtonsID = 1;
            ButStatTextID = 2;
            ButLineEditID = 3;
            ButPicturesID = 4;
            ButPopUpsID = 5;
            ButTextEditID = 6;
            ButListsID = 7;

            Prog1ID = 8;
            Prog2ID = 9;
            Prog3ID = 10;
            Prog4ID = 11;
            Prog5ID = 12;
            Prog6ID = 13;


var
      MyMemoryID  : integer;  {Application ID assigned by Memory Mgr}
      Done        : boolean;  {True when quitting}
      StaggerCount : integer; {Used to stagger windows as they open }
      Event       : WmTaskRec;
                              {All events are returned here}

      WindowList  : array [0..NumWindowsMin1] of WindowPtr;
```

```
procedure InitGlobals;        {Setup variables}

IMPLEMENTATION

procedure InitGlobals;
      begin       {of InitGlobals}
            MyMemoryID := MMStartup;
            StaggerCount := 0;
      end;        {of InitGlobals}


END.
```

The uMenu.p module

This section contains the source code for uMenu.p. This Pascal module implements menus for the BusyBox program.

```
{***************************************************************************}
{*
{* BusyBox uMenu (interface)
{*
{* Copyright (c)
{* Apple Computer, Inc. 1986-1990
{* All Rights Reserved.
{*
{* This file contains the interface to the code that implements
{* menus in the BusyBox program.
{*
{***************************************************************************}

UNIT uMenu;

INTERFACE

USES
      types,
      locator,
      quickdraw,
      fonts,
      INTMATH,
      events,
      memory,
      controls,
      gsos,
      windows,
      lineedit,
      dialogs,
      menus,
      desk,
      STDFILE,
      resources,
```

```
      uGlobals,
      uUtils,
      uWindow;


procedure DoMenu;        {Execute a menu item}
procedure SetUpMenus;    {Install menus and redraw menu bar}


IMPLEMENTATION

{$R-}

procedure DoQuitItem;

      {Private routine to set Done flag if the "Quit" item was selected}

      begin       {of DoQuitItem}
            Done := true;
      end;        {of DoQuitItem}



procedure DoAboutItem;
            var
                  ignore : integer;
      begin       {of DoAboutItem}
            ignore := AlertWindow(4,NIL,Ptr(1));
      end;        {of DoAboutItem}


procedure DoMenu;

      {Procedure to handle all menu selections.  Examines the }
      {Event.TaskData menu item ID word from TaskMaster (from Event }
      {Manager) and calls the appropriate routine.  While the routine }
      {is running the menu title is still highlighted.  After the }
      {routine returns, we remove the highlighting.}


      var   menuNum : integer;
            itemNum : integer;
```

```
        begin {of DoMenu}

              menuNum := HiWord (Event.wmTaskData);
              itemNum := LoWord (Event.wmTaskData);

              case itemNum of
                    AboutItem   :     DoAboutItem;
                    CloseItem   :     DoCloseTop;
                    QuitItem    :     DoQuitItem;
                    UndoItem    :     ;
                    CutItem     :     ;
                    CopyItem    :     ;
                    PasteItem   :     ;
                    ClearItem   :     ;
              otherwise
                    ;
              end;

              HiliteMenu (false,menuNum);
                          {Remove highlighting}
                          { *** MAX *** }

        end;  {of DoMenu}


procedure SetUpMenus;

      {Procedure to install our menu titles and their items in the }
      {system menu bar and to redraw it so we can see them}

      var height : integer;

      begin {of SetUpMenus}
            SetSysBar(NewMenubar2(RefIsResource,ref($1000),NIL));
            SetMenuBar(NIL);

            FixAppleMenu (AppleMenuID);   {Add DAs to Apple menu  }
            height := FixMenuBar;         {Set sizes of menus}
            DrawMenuBar;                  {...and draw the menu bar!}
      end;  {of SetUpMenus}

END.
```

The `uUtils.p` module

This section contains the source code for `uUtils.p`. This Pascal module contains various utility routines for the BusyBox program.

```
{*************************************************************************}
{*
{* BusyBox uUtils (interface)
{*
{* Copyright (c)
{* Apple Computer, Inc. 1986-1990
{* All Rights Reserved.
{*
{* This file contains the interface to the code that implements
{* various utility routines used by the BusyBox program.
{*
{*************************************************************************}

Unit uUtils;

INTERFACE

USES
      types,
      locator,
      intMath;


CONST
      srcCopy     =     $0000;

FUNCTION IntToString (i : Integer): STR255;
FUNCTION LongToString (l : LongInt): STR255; { test }
FUNCTION IsToolError: BOOLEAN;
PROCEDURE INC(VAR anIndex : Integer);
PROCEDURE Dec(VAR anIndex: Integer);

IMPLEMENTATION

{$R-}
```

```
FUNCTION IntToString (i : Integer): STR255;
var
      size,
      count : Integer;
      num : longInt;
      str : string[20];
BEGIN
      num := i;
      size := 0;
      Long2Dec(num, @str, 19, true);
      FOR count := 1 to 19 DO
        BEGIN
            IF (str[count] = '-') OR
                  ((str[count] >= '0') AND (str[count] <= '9')) THEN
                  BEGIN
                        size := size + 1;
                        IntToString[size] := str[count];
                  END;
        END;
      IntToString[0] := char(size);
END;


FUNCTION LongToString (l : LongInt): STR255; { test }
var
      size,
      count : Integer;
      num : longInt;
      str : string[20];
```

```
BEGIN
      num := 1;
      size := 0;
      Long2Dec(num, @str, 19, true);
      FOR count := 1 to 19 DO
        BEGIN
            IF (str[count] = '-') OR
                  ((str[count] >= '0') AND (str[count] <= '9')) THEN
                  BEGIN
                        size := size + 1;
                        LongToString[size] := str[count];
                  END;
        END;
      LongToString[0] := char(size);
END;


FUNCTION IsToolError: BOOLEAN;
BEGIN
      IsToolError := FALSE;
      if ToolErrorNum <> 0 then
      IsToolError := TRUE;
END;

PROCEDURE INC(VAR anIndex : Integer); {increase integer param by 1}
BEGIN
      anIndex := anIndex + 1;
END;

PROCEDURE Dec(VAR anIndex: Integer); {decrease integer param by 1}
BEGIN
      anIndex := anIndex - 1;
END;

END.
```

The `uWindow.p` module

This section contains the source code for `uWindow.p`. This Pascal module implements windows for the BusyBox program.

```
{*************************************************************************}
{*
{* BusyBox uWindow (interface)
{*
{* Copyright (c)
{* Apple Computer, Inc. 1986-1990
{* All Rights Reserved.
{*
{* This file contains the interface to the code that implements
{* windows in the BusyBox program.
{*
{*************************************************************************}

UNIT uWindow;

INTERFACE

USES
      types,
      GSOS,
      locator,
      quickdraw,
      fonts,
      MEMORY,
      intMath,
      events,
      controls,
      windows,
      lineedit,
      dialogs,
      menus,
      DESK,
      STDFILE,
      resources,
      TextEdit,
```

```
      uGlobals,
      uUtils;

var
      TheMainWindow,
      ButtonsWindow,
      StatTextWindow,
      LineEditWindow,
      PicturesWindow,
      PopUpsWindow,
      TextEditWindow,
      ListsWindow             : GrafPortPtr;

procedure SetUpWindows;
                        {Initialize variables for stacking windows}
procedure DrawThisWindow;
procedure DoCloseTop;
procedure OpenThisWindow      (CtlID : integer);



IMPLEMENTATION

{$R-}

const

      MainWindowID = $2000;


{*********************************************************************}
{
{ DrawThisWindow
{
{ This routine draws the contents of all the windows.
{
{*
{* Warning: Do not make any calls that use the libraries or use
{* short addressing without setting the dbr to ~globals.
{*
{*********************************************************************}
```

```
procedure DrawThisWindow;
      begin
            DrawControls(GetPort);
      END;

{*****************************************************************}
{
{ DoCloseTop
{
{ This routine closes the topmost window. We do a little work to
{ prevent the main window from being closed.
{
{*****************************************************************}
procedure DoCloseTop;
      var
            k : integer;
            TempWin : GrafPortPtr;

      begin
            {Get the front window into a local variable }
            TempWin := FrontWindow;

            {Start the count at 1 since we never close the main window }
            k := 1;

            {Find the window entry, close the window, and zero the }
            {entry repeat}

                  if TempWin = WindowList[k] then
                        begin
                              CloseWindow(TempWin);
                              WindowList[k] := NIL;
                              k := NumWindows;
                        end
                  else
                        k := k+1;
            until k >= NumWindows;
      end;
```

```
{*********************************************************************}
{
{ OpenThisWindow
{
{ This routine either opens the specified window or makes it active
{ if it is already open.
{
{ If it is not open, we open it with NewWindow2 invisibly, adjust the
{ window's location, and then show and select the window.
{
{
{ ID values for controls in the main window are assumed here to be from
{ 1...n
{
{*********************************************************************}
procedure OpenThisWindow      (CtlID : integer);
      begin
            if WindowList[CtlID] = NIL then
                  begin
                        WindowList[CtlID] :=
                              NewWindow2(NIL,
                              0,
                              @DrawThisWindow,
                              NIL,
                              2,
                              Ref(POINTER(MainWindowID+CtlID)),
                              rWindParam1);
                        if CtlID < ProgIID then
                              begin
                                    MoveWindow (50+8*StaggerCount,
                                          50+8*StaggerCount,
                                          WindowList[CtlID]);
                                    StaggerCount := StaggerCount+1;
                              end;
                        ShowWindow(WindowList[CtlID]);
                        SelectWindow(WindowList[CtlID]);
                  end
            else SelectWindow(WindowList[CtlID]);
      end;
```

```
{*********************************************************************}
{
{ SetUpWindows
{
{ Sets up WindowList record for use throughout the program.
{
{*********************************************************************}
procedure SetUpWindows;
      var
            k : integer;

      begin       {of SetUpWindows}
            { Zero out the entries in the window list }
            for k := 0 to NumWindows-1 do WindowList[k] := NIL;

            { Open the main window }
            WindowList[0] := NewWindow2(NIL,
                                        0,
                                        @DrawThisWindow,
                                        NIL,
                                        2,
                                        ref(MainWindowID),
                                        rWindParam1);
      end;        {of SetUpWindows}



END.
```

Glossary

absolute: Characteristic of a load segment or other program code that must be loaded at a specific address in memory and never moved. Compare **relocatable.**

accelerator card: An expansion card that contains another processor that shares the work normally performed only by the computer's main microprocessor. An accelerator card speeds up processing time.

accessory: See **desk accessory.**

accumulator: The register in a computer's central processor or microprocessor where most computations are performed.

ACIA: Abbreviation for *Asynchronous Communications Interface Adapter;* a type of communications IC used in some Apple computers. An ACIA converts data from parallel to serial form and vice versa. It handles serial transmission and reception and RS-232-C signals under the control of its internal registers, which can be set and changed by firmware or software.

acronym: A word formed from the initial letter or letters of the main parts of a compound term, such as ROM (from *read-only memory*) or Fortran (from *Formula Translator*).

activate: To make a nonactive window active by clicking anywhere inside it.

activate event: An event generated by the Window Manager when an inactive window becomes the active window.

active window: The frontmost window on the desktop; the window where the next action will take place. An active window's title bar is highlighted.

Adaptive Differential Pulse Code Modulation (ADPCM): An algorithm for digitizing audio samples. Used in the Apple IIGS Audio Compression and Expansion Tool Set for compressing audio samples.

ADB: See **Apple Desktop Bus.**

ADB device table: A structure in the system heap that lists all devices connected to the Apple Desktop Bus.

address: (1) A number that specifies the location of a single byte of memory. Addresses can be given as decimal or hexadecimal integers. The Apple IIGS has addresses ranging from 0 to 16,777,215 (in decimal), or from $00 0000 to $FF FFFF (in hexadecimal). A complete address consists of a 4-bit **bank** number ($00 to $FF) followed by a 16-bit address within that bank ($0000 to $FFFF). (2) In data transmission, a code for a specific terminal. Multiple terminals on one communication line, for example, must have unique addresses.

ADPCM: See **Adaptive Differential Pulse Code Modulation.**

ADSR: Acronym for **attack, decay, sustain,** and **release.** These terms describe the paradigm for representing sounds in terms of a sound **envelope.**

alert: A warning or report of an error in the form of an alert box, a sound from the computer's speaker, or both.

alert window: Similar to a modal dialog box; used to present urgent or important information to the user. You create alert windows with the `AlertWindow` Window Manager tool call.

algorithm: A step-by-step procedure for solving a problem or accomplishing a task.

allocate: To reserve an area of memory for use.

American Standard Code for Information Interchange: See **ASCII.**

amplitude: The maximum vertical distance of a periodic wave from the horizontal line about which the wave oscillates.

AND: A logical operator that produces a TRUE result if both of its operands are true, and a FALSE result if either or both of its operands are false. Compare **exclusive OR, NOT, OR.**

ANSI: Acronym for *American National Standards Institute,* which sets standards for many technical fields and provides the most common standard for computer terminals.

Apple Desktop Bus (ADB): A low-speed, input-only serial bus with connectors on the back panel of the computer that you use to attach the keyboard, mouse, and other Apple Desktop Bus devices, such as graphics tablets, hand controls, and specialized keyboards.

Apple key: See **Command key.**

Apple menu: The menu farthest to the left in the menu bar, indicated by an Apple symbol, from which you choose **desk accessories.**

Apple I: The first Apple computer. It was built in a garage in California by Steve Jobs and Steve Wozniak.

AppleTalk network system: The system of network software and hardware used in various implementations of Apple's communications network.

Apple II: A family of computers, including the original Apple II, the Apple II Plus, the Apple III, the Apple IIe, the Apple IIc, and the Apple IIGS. Compare **standard Apple II.**

Apple IIc: A transportable personal computer in the Apple II family, with a disk drive and 80-column display capability built in.

Apple IIe: A personal computer in the Apple II family with seven expansion slots and an auxiliary memory slot that allow the user to enhance the computer's capabilities with peripheral and auxiliary cards.

Apple IIGS: A personal computer in the Apple II family; GS stands for *graphics and sound.* The Apple IIGS features super high-resolution graphics, 15-voice sound capabilities, and 256K of RAM with a memory expansion slot for adding from 1 to 8 megabytes of RAM.

Apple IIGS Interface Libraries: A set of **interfaces** that enable you to access Toolbox routines from C.

Apple IIGS Programmer's Workshop (APW): The development environment for the Apple IIGS computer. It consists of a set of programs that facilitate the writing, compiling, and debugging of Apple IIGS applications.

Apple IIGS tools: See **toolbox.**

Apple II Pascal: A software system for the Apple II family that lets you create and execute programs written in the Pascal programming language. Apple II Pascal was adapted by Apple Computer from the University of California, San Diego, Pascal Operating System (UCSD Pascal).

Apple II Plus: A personal computer in the Apple II family with eight expansion slots and 48K of RAM, expandable to 64K with a language card in slot 0.

Apple III: An Apple computer; part of the Apple II family. The Apple III offered a built-in disk drive and built-in RS-232-C (serial) port. Its memory was expandable to 256K.

application: On the Apple IIGS, a program (such as the APW Shell) that accesses ProDOS 16 and the Toolbox directly, and that can be called or exited via the QUIT call. ProDOS 16 applications are file type $B3.

application software: A collective term for **application programs.**

APW: see **Apple IIGS Programmer's Workshop.**

APW Debugger: A 65816 assembly-language code debugger provided with the Apple IIGS Programmer's Workshop.

APW Editor: The program within the Apple IIGS Programmer's Workshop that allows you to enter, modify, and save source files for all APW languages.

APW Linker: The linker supplied with the Apple IIGS Programmer's Workshop.

APW Shell: The shell program of the Apple IIGS Programmer's Workshop. The APW Shell provides the interface between APW programs and ProDOS and between the user and APW.

argument: (1) A value on which a function or statement operates; it can be a number or a variable. For example, in the BASIC statement `VTAB 10`, the number 10 is the argument. (2) A piece of information included on the command line in addition to the command; the shell passes this information to the command, which then modifies its execution in some particular way. Filenames, for example, are often supplied as arguments to commands, so that a command will operate on the named file.

argument list: All the arguments passed to a program.

arithmetic expression: A combination of numbers and arithmetic operators (such as 3 + 5) that indicates some operation to be carried out.

arithmetic operation: One of the five actions computers can perform with numbers: addition, subtraction, multiplication, division, and exponentiation.

arithmetic operator: An operator, such as +, that combines numeric values to produce a numeric result. Compare **Boolean operator.**

array: An ordered collection of information of a given, defined type. Each element of the array can be referred to by a numerical subscript.

arrow keys: The four directional keys in the lower-right corner of the keyboard. You can use the arrow keys to move around in an application.

ASCII: Acronym for *American Standard Code for Information Interchange* (pronounced "ASK-ee"). A standard that assigns a unique binary number to each text character and control character. ASCII code is used for representing text inside a computer and for transmitting text between computers or between a computer and a peripheral device.

aspect ratio: The ratio of an image's width to its height. For example, a standard video display has an aspect ratio of 4:3.

assembly code: A source file written in a low-level programming language that corresponds to a specific computer's binary machine language.

assembly language: A low-level programming language in which individual machine-language instructions are written in a symbolic form that's easier to understand than machine language itself. Each assembly-language instruction produces one machine-language instruction. Because assembly-language programs require very little translation, they can be very fast.

Asynchronous Communications Interface Adapter: See ACIA.

attack: That portion of a sound **envelope** during which the sound increases from silence to its peak loudness. See also **ADSR.**

auto-key event: An event generated repeatedly when the user presses and holds down a character key on the keyboard or keypad.

auto-repeat feature: A feature of keys on computer keyboards; when a key is pressed down and held, the computer will automatically repeat that key's character until the key is released.

background activity: A program or process that runs while the user is engaged with another application.

back panel: The rear surface of the computer, which includes the power switch, the power connector, and connectors for peripheral devices.

backspace: To move to the left in a line of text, erasing the character or selection; thus synonymous with *delete.*

bank: A 64K (65,536-byte) portion of the Apple IIGS internal memory. An individual bank is specified by the value of one of the 65C816 microprocessor's bank registers.

bank-switched memory: On Apple II computers, the part of **language card** memory in which two 4K portions of memory have the same address range ($D000 to $DFFF).

BASIC: Acronym for *Beginners All-purpose Symbolic Instruction Code;* a high-level programming language designed to be easy to learn. Two versions of BASIC are available from Apple Computer for use with all Apple II–family systems: Applesoft BASIC (built into the firmware) and Integer BASIC.

battery RAM: RAM on the Macintosh and Apple IIGS clock chips. A battery preserves the clock settings and the RAM contents when the power is off. Control Panel settings are kept in battery RAM.

binary: (adj.) Characterized by having two different components or by having only two alternatives or values available; sometimes used synonymously with **binary system.**

binary digit: The smallest unit of information in the binary number system; a 0 or a 1. Also called a *bit.*

binary file format: The ProDOS 8 loadable file format, consisting of one absolute memory image along with its destination address. A file in binary file format has ProDOS file type $06 and is referred to as a *BIN file.* The System Loader cannot load BIN files.

binary system: (1) A number system that uses only 0 and 1 as digits. Because computers can keep track of only two states (on and off), engineers code data in terms of 0's and 1's. (2) The representation of numbers in the base-2 system, using only the two digits 0 and 1. For example, the numbers 0, 1, 2, 3, and 4 become 0, 1, 10, 11, and 100 in binary notation. The binary system is commonly used in computers because the values 0 and 1 can easily be represented in a variety of ways, such as the presence or absence of current, positive or negative voltage, or a white or black dot on the display screen. A single binary digit—a 0 or a 1—is called a *bit*. Compare **hexadecimal system.**

bit: A contraction of *binary digit.* The smallest unit of information that a computer can hold. The value of a bit (1 or 0) represents a simple two-way choice, such as yes or no, on or off, positive or negative, something or nothing. See also **binary system.**

bit image: A collection of bits in memory that represents a two-dimensional surface. For example, the screen is a visible bit image.

bitmap: (1) A set of bits that represents the graphic image of an original document in memory. (2) A set of bits that represents the positions and states of a corresponding set of items, such as pixels. In QuickDraw, a pointer to a bit image, the row width of that image, and its boundary rectangle.

bitmapped character: A character that exists in a computer file or in memory as a bitmap, is drawn as a pixel pattern on the graphics screen, and is sent to the printer as graphics data.

bitmapped display: A display whose image is a representation of bits in an area of RAM called the **screen buffer.** With such a display, each dot, or **pixel,** on the screen corresponds, or is "mapped," to a bit in the screen buffer.

board: See **printed-circuit board.**

Boolean operator: An operator, such as AND, that combines logical values to produce a logical result, such as true or false. Named for mathematician and logician George Boole. Also known as a *logical operator.* Compare **arithmetic operator.**

boot: Another way to say *start up.* A computer boots by loading a program into memory from an external storage medium such as a disk. Starting up is often accomplished by first loading a small program, which then reads a larger program into memory. The program is said to "pull itself up by its own bootstraps"—hence the term *bootstrapping* or *booting.*

boot device: The peripheral device that reads an operating system's initial startup instructions.

boot disk: See **startup disk.**

bootstrap: See **boot.**

branch: (v.) To pass program control to a line or statement other than the next in sequence. (n.) A statement that performs the act of branching.

buffer: (1) An area of memory set aside for the specific purpose of holding data until it is needed. (2) A "holding area" of the computer's memory where information can be stored by one program or device and then read at a different rate by another; for example, a print buffer. In editing functions, an area in memory where deleted (cut) or copied data is held. In some applications, this area is called the *Clipboard*. See also **type-ahead buffer.**

bug: An error in a program that causes it not to work as intended. The expression reportedly comes from the early days of computing when an itinerant moth shorted a connection and caused a breakdown in a room-sized computer.

button: (1) A pushbutton-like image in dialog boxes where you click to designate, confirm, or cancel an action. Compare **mouse button.**

byte: A unit of information consisting of a fixed number of **bits.** On Apple II systems, one byte consists of a series of eight bits and can take any value between 0 and 255 ($0 and $FF hexadecimal). The value can represent an instruction, number, character, or logical state. See also **kilobyte, megabyte.**

C: A portable, high-level language that also offers very low-level operations, making it a flexible and efficient language for both application and system programming.

call: (n.) (1) A request from the keyboard or from a procedure to execute a named procedure. (2) A request issued by the CPU or a program to the SCSI card firmware. (v.) To request the execution of a subroutine, function, or procedure.

Cancel button: A button that appears in a dialog box. Clicking it cancels the command.

Caps Lock key: A key that, when engaged, causes subsequently typed letters to appear in uppercase; its effect is like that of the Shift key except that it doesn't affect numbers and other nonletter symbols.

card: (1) A printed-circuit board that plugs into one of the computer's expansion slots, allowing the computer to use one or more peripheral devices such as disk drives. (2) A printed-circuit board or card connected to the bus in parallel with other cards. Also called a *peripheral card,* a *device,* or a *module.*

caret: A generic term meaning a symbol that indicates where something should be inserted in text. The specific symbol used onscreen is a vertical bar (|).

carriage return (CR): A nonprinting ASCII character (decimal 13, hexadecimal $0D) that ordinarily causes a printer or display device to place the next character on the left margin; that is, to end a line of text and start a new one. It's used to end paragraphs. A carriage return, however, does not move the print head or cursor down to the next line; the line feed (LF) character does that. Even though you can't see carriage returns, you can delete them the same way you delete other characters. In APW C, carriage return (`\r`) is equal to newline (`\n`).

carry flag: A status bit in the microprocessor, used as an additional high-order bit with the accumulator bits in addition, subtraction, rotation, and shift operations.

case sensitive: Able to distinguish between uppercase characters and lowercase characters. Programming languages are case sensitive if they require all uppercase letters, all lowercase letters, or proper use of uppercase and lowercase. Instant Pascal, however, is not case sensitive; you can use any combination of uppercase and lowercase letters you like.

cathode-ray tube (CRT): An electronic device, such as a television picture tube, that produces images on a phosphor-coated screen. The phosphor coating emits light when struck by a focused beam of electrons. A CRT is a common display device used with personal computers.

CCITT: Abbreviation for *Consultative Committee on International Telegraphy and Telephony;* an international committee that sets standards and makes recommendations for international communication. The CCITT interface standard is considered mandatory in Europe; it is very similar to the RS-232 standard used in the United States.

central processing unit (CPU): The "brain" of the computer; the microprocessor that performs the actual computations in machine language.

channel: A queue that's used by an application to send commands to the Sound Manager.

character: Any symbol that has a widely understood meaning and thus can convey information. Some characters—such as letters, numbers, and punctuation—can be displayed on the monitor screen and printed on a printer.

character code: An integer representing the character that a key or key combination stands for.

character key: (1) Any of the keys on a computer keyboard—such as letters, numbers, symbols, and punctuation marks—used to generate text or to format text; any key except Caps Lock, Command, Control, Esc, Option, and Shift. Character keys repeat when you press and hold them down. (2) A key that generates a keyboard event when pressed; that is, any key other than a **modifier key.**

check box: A small box associated with an option in a dialog box. When you click the check box, you may change the option or affect related options.

chip: See **integrated circuit.**

circuit board: A board containing embedded circuits and an attached collection of integrated circuits (chips). Sometimes called a *printed-circuit board* or *card.*

circuitry: A network of wires, chips, resistors, and other electronic devices and connections.

clamp: A memory location that contains the minimum and maximum excursion positions of the mouse cursor when the desktop is in use.

clear: (1) To erase information or commands from memory. (2) To erase data from memory or reset a control register. Clearing is usually done by loading the memory location or register to be cleared with zeros.

click: (v.) To position the pointer on something, and then press and quickly release the mouse button. (n.) The act of clicking.

Clipboard: The holding place for what you last cut or copied; a buffer area in memory. Information on the Clipboard can be inserted (pasted) into documents.

clipping region: The region to which an application limits drawing within a graphics port.

clock chip: A special chip in which parameter RAM and the current setting for the date and time are stored. This chip is powered by a battery when the system is off, thus preserving the information.

close: (1) To turn a window back into the icon that represents it by choosing the Close command or by clicking the close box on the left side of the window's title bar. (2) To terminate access to an open file. When a file is closed, its updated version is written to disk and all resources it needed when open (such as its I/O buffer) are released. The file must be opened before it can be accessed again.

close box: The small white box on the left side of the title bar of an active window. Clicking it closes the window.

code: (1) A number or symbol used to represent some piece of information. (2) The statements or instructions that make up a program.

command: (1) An instruction that causes a device such as a computer or printer to perform some action. A command can be typed from a keyboard, selected from a menu with a hand-held device (such as a mouse), or embedded in a program. (2) In the Standard C Library, a parameter that tells a function which of several actions to perform. (3) In the APW Shell, a word that tells APW which utility to execute. (4) An instruction that causes the target device to perform a specific operation. Commands are passed to the firmware in **calls.**

command code: One or more characters whose function is to change the way a program or device acts (as opposed to text, which is simply printed).

Command key: A key that, when held down while another key is pressed, causes a command to take effect. When held down in combination with dragging the mouse, the Command key lets you drag a window to a new location without activating it. The Command key is marked with a propeller-shaped symbol. On some machines, the Command key has both the propeller symbol and the Apple symbol on it.

compact: To rearrange allocated memory blocks in order to increase the amount of contiguous unallocated (free) memory. The Memory Manager compacts memory when needed.

compaction: The process of moving allocated blocks within a heap zone to collect the free space into a single block.

compatibility: The condition under which devices can work with each other.

compatible: Capable of running without problems on the computer system. Applications are normally written to run on specific types of computers; applications that run on a computer system are said to be "compatible" with the computer.

compile: To convert a program written in a high-level programming language (source code) into a file of commands in a lower-level language (object code) for later execution.

component: A part; in particular, a part of a computer system.

computer: An electronic device that performs predefined (programmed) computations at high speed and with great accuracy; a machine that is used to store, transfer, and transform information.

concatenate: Literally, "to chain together." (1) To combine two or more strings into a single, longer string by joining the beginning of one to the end of the other. (2) To combine two or more files.

configuration: (1) A general-purpose computer term that can refer to the way you have your computer set up. (2) The total combination of hardware components—central processing unit, video display device, keyboard, and peripheral devices—that make up a computer system. (3) The software settings that allow various hardware components of a computer system to communicate with one another.

configure: To change software or hardware actions by changing settings. For example, you give software the necessary settings for communicating with a printer. You can configure hardware (a printer or interface card) by resetting physical elements like DIP switches or jumper blocks. Configurations can also be set or reset in software.

content region: The area of a window that the application draws in.

context sensitive: Able to perceive the situation in which an event occurs. For example, an application program might present help information specific to the particular task you're performing, rather than a general list of commands; such help would be context sensitive.

control: (1) The order in which the statements of a program are executed. (2) An object in a window on the screen with which the user, by using the mouse, can cause instant action with visible results or change settings to modify a future action. The control is internally represented in a control record.

control character: A nonprinting character that controls or modifies the way information is printed or displayed. In the Apple II computer family, control characters have ASCII values between 0 and 31, and can be typed from a keyboard by holding down the Control key while pressing some other key.

control key: See modifier **key.**

Control key: A specific key on Apple II–family keyboards that produces **control characters** when used in combination with other keys.

Control Manager: The part of the toolbox that provides routines for creating and manipulating controls (such as buttons, check boxes, and scroll bars).

Control Panel: A desk accessory that lets you change the speaker volume, the keyboard repeat speed and delay, mouse tracking, color display, and other features.

control register: A special register that programs can read from and write to; similar to **soft switches.** The control registers are specific locations in the I/O space ($Cxxx) in bank $E0. They are accessible from bank $00 if I/O shadowing is on.

control template: Structure containing the information necessary for the `NewControl2` Control Manager tool call to create a new control.

coordinate: One of a pair of numbers that designates a position on a grid. The numbers correspond to the columns (vertical placement) and rows (horizontal placement) in a display grid.

CR: See **carriage return.**

crash: To cease to operate unexpectedly, possibly destroying information in the process. Compare **hang.**

CRT: See **cathode-ray tube.**

cursor: (1) A symbol displayed on the screen marking where the user's next action will take effect or where the next character typed from the keyboard will appear. (2) A mark on the screen that indicates your position on the command line or inside a file. The cursor is usually a small box or an underscore, and it usually blinks. (3) The term used in technical manuals for the **pointer** on the screen.

cut: To remove something by selecting it and choosing Cut from a menu. What you cut is placed on the Clipboard. In other editing applications, "Delete" serves the same function. See also **buffer.**

cut and paste: To move something from one place in a document to another in the same document or a different one. It's the computer equivalent of using scissors to clip something and glue to paste the clipping somewhere else.

debug: A colloquial term that means to locate and correct an error or the cause of a problem or malfunction in a computer program. Often synonymous with *troubleshoot*. See also **bug.**

debugger: A utility that allows you to analyze a program for errors that cause it to malfunction. For example, a debugger may allow you to step through execution of the program one instruction at a time.

decay: That portion of a sound **envelope** during which the sound falls off from its peak loudness to a sustained level. See also **ADSR.**

default: A value, action, or setting that a computer system assumes, unless the user gives an explicit instruction to the contrary. For example, unless told otherwise, the ImageWriter LQ begins printing with a left margin set to the default value of 0. Default values prevent a program from stalling or crashing if no value is supplied by the user.

default prefix: The pathname prefix attached by ProDOS 16 to a partial pathname when no prefix number is supplied by the application. The default prefix is equivalent to prefix number 0/.

delete: To remove something, such as a character or word from a file, or a file from a disk. Keys such as the Backspace key and the Delete key can remove one character at a time by moving to the left. The Cut command removes selected text and places it on the Clipboard; the Clear command removes selected text without placing it on the Clipboard. (The Undo command can reverse the action of Clear and of the Backspace or Delete key if it is used immediately.)

delta: The difference from something the program already knows. For example, mouse moves are represented as deltas compared to previous mouse locations. The name comes from the way mathematicians use the Greek letter delta (Δ) to represent a difference.

delta guide: A description of something new in terms of its differences from something the reader already knows about. The name comes from the way mathematicians use the Greek letter delta (Δ) to represent a difference.

deselect: A command to a device such as a printer to place it into a condition in which it will not receive data. A deselect command has an effect opposite to that of a **select** command.

desk accessory: A "mini-application" that is available from the **Apple menu** regardless of which application you're using.

Desk Manager: The part of the Toolbox that supports the use of desk accessories from an application.

desk scrap: See **Clipboard.**

desktop: Your working environment on the computer—the menu bar and the gray area on the screen. You can have a number of documents on the desktop at the same time. At the Finder level, the desktop displays the Trash and the icons (and windows) of disks that have been accessed.

desktop environment: A set of program features that make user interactions with an application resemble the way people work on a desktop. Commands appear as options in pull-down menus, and material being worked on appears in areas of the screen called **windows.** The user selects commands or other material by using the mouse to move a pointer around on the screen or by using keyboard equivalents.

device address: A value in the range $00 through $0F assigned to each device connected to the **Apple Desktop Bus.**

device driver: A program that manages the transfer of information between the computer and a peripheral device. See also **resource.**

dialog: See **dialog box.**

dialog box: (1) A box that contains a message requesting more information from you. Sometimes the message warns you that you're asking your computer to do something it can't do or that you're about to destroy some of your information. In these cases, the message is often accompanied by a beep. (2) A box that a Macintosh application displays to request information or to report that it is waiting for a process to complete. A dialog box is internally represented in a dialog record.

digit: (1) One of the characters 0 through 9, used to express numbers in decimal form. (2) One of the characters used to express numbers in some other form, such as 0 and 1 in binary or 0 through 9 and A through F in hexadecimal.

Digital Oscillator Chip (DOC): An integrated circuit in the Apple IIGS that contains 32 digital oscillators, each of which can generate a sound from stored digital waveform data.

dimmed: Used to describe words or icons that appear in gray. For example, menu commands appear dimmed when they are unavailable; folder icons are dimmed when they are open.

dimmed icon: An icon that represents an opened disk or folder or a disk that has been ejected. Double-clicking a dimmed disk or folder icon causes the window for the disk or folder to become the frontmost, active window. You can select and open a dimmed icon representing an ejected disk, but you cannot open the folders or documents on it unless you insert the disk.

direct page: A page (256 bytes) of bank $00 of Apple IIGS memory, any part of which can be addressed with a short (one-byte) address because its high-order address byte is always $00 and its middle address byte is the value of the 65C816 direct register. Co-resident programs or routines can have their own direct pages at different locations. The direct page corresponds to the 6502 processor's zero page. The term *direct page* is often used informally to refer to any part of the lower portion of the **direct-page/stack space.** See also **direct register, zero page.**

direct-page/stack space: A portion of bank $00 of Apple IIGS memory reserved for a program's direct page and stack. Initially, the 65C816 processor's direct register contains the base address of the space, and its stack register contains the highest address. In use, the stack grows downward from the top of the direct-page/stack space, and the lower part of the space contains direct-page data. See also **direct page, direct register, stack, stack register.**

direct register: A hardware register in the 65C816 processor that specifies the start of the direct page.

disabled: Describes a menu item or menu that cannot be chosen; the menu item or menu title appears dimmed. A disabled item in a dialog or alert box has no effect when clicked.

display: (1) A general term to describe what you see on the screen of your display device when you're using a computer; from the verb form, which means "to place into view." (2) Short for *display device*.

display color: The color currently being used to draw high-resolution or low-resolution graphics on the display screen.

display device: A device that displays information, such as a television set or video monitor.

display rectangle: A rectangle that determines where an item is displayed within a dialog or alert box.

display screen: The screen of the monitor; the area where you view text and pictures when using the computer. Also called simply the *screen*.

dispose: To permanently deallocate a memory block. The Memory Manager disposes of a memory block by removing its master pointer. Any handle to that pointer will then be invalid. Compare **purge.**

disposition: An attribute of the data set where the host components reside.

dithering: A technique for alternating the values of adjacent dots or pixels to create the effect of intermediate values. In printing color or displaying color on a computer screen, the technique of making adjacent dots or pixels different colors to give the illusion of a third color. For example, a printed field of alternating cyan and yellow dots appears to be green. Dithering can give the effect of shades of gray on a black-and-white display, or more colors on a color display.

dither pattern: The matrix of threshold values used to represent gray shades in a black-and-white electronic image.

DOC: See **Digital Oscillator Chip**.

double click: (n.) Two clicks in quick succession, interpreted as a single command. The action of a double click is different from that of a single click. For example, clicking an icon selects the icon; double-clicking an icon opens it.

double-click: (v.) To position the pointer where you want an action to take place, and then press and release the mouse button twice in quick succession without moving the mouse.

double-click time: The greatest interval between a mouse-up event and a mouse-down event that would qualify two mouse clicks as a double click.

drag: To position the pointer on something, press and hold the mouse button, move the mouse, and release the mouse button. When you release the mouse button, you either confirm a selection or move an object to a new location.

drag region: A region in a window frame; usually the title bar. Dragging inside this region moves the window to a new location and makes it the active window unless the Command key was down.

drop sample tuning: A technique for changing the pitch of a played sound that relies on skipping (or dropping) sound samples on playback. When samples are dropped at a fixed rate, the pitch of a sound can be raised in octave increments.

echo: To send an input character back to the originating device for display or verification; for example, to send each character of your message back to your monitor so you know it's been sent to another computer or to a printer.

edit: To change or modify. For example, to insert, remove, replace, or move text in a document.

editor: A program that helps you create and edit information of a particular form; for example, a text editor or a graphics editor.

edit record: A complete editing environment in TextEdit, which includes the text to be edited, the GrafPort and rectangle in which to display the text, the arrangement of the text within the rectangle, and other editing and display information.

e flag: One of three flag bits in the 65C816 processor that programs use to control the processor's operating modes. The setting of the e flag determines whether the processor is in native mode or emulation mode. See also **m flag, x flag.**

eject: (1) To remove a disk from a disk drive. (2) To move paper out of the printer. You can eject paper by pressing the Form Feed button or by turning the platen knob clockwise.

embedded: Contained within. For example, the string `'HUMPTY DUMPTY'` is said to contain an embedded space.

end-of-file (EOF): (1) In A/UX, the position of one byte past the last byte in a file (also known as the *logical end-of-file*); this is equal to the actual number of bytes in the file. If a program calls a routine that uses the physical end-of-file convention, the logical end-of-file is used instead. (2) The logical size of a ProDOS 16 file; it is the number of bytes that may be read from or written to the file. See also **logical end-of-file, physical end-of-file.**

Enter key: A key that confirms an entry or sometimes a command.

envelope: A graphic representation of a sound's loudness over time. The envelope typically consists of segments identified as **attack, decay, sustain,** and **release,** or **ADSR.**

error code: A number or other symbol representing a type of error.

event: A notification to an application of some occurrence, such as an interrupt created by a keypress, that the application may want to respond to.

exclusive OR: A logical operator that produces a true result if one of its operands is true and the other false, and a false result if its operands are both true or both false. Sometimes written as *XOR*. Compare **AND, NOT, OR.**

extended controls: Controls created with the `NewControl2` Control Manager tool call, rather than the `NewControl` call. Extended controls have new-style control records that contain more information than those created by `NewControl`.

fatal error: An error serious enough that the computer must halt execution.

field: (1) A data item separated from other data by blanks, tabs, or other specific delimiters. A particular type or category of information in a database management program. (2) A specific set of data that is related. A field is always defined by its size, given in bits or bytes, and usually has a name. (3) A string of ASCII characters or a value that has a specific meaning to some program. Fields may be of fixed length, or may be separated from other fields by field delimiters. For example, each parameter in a segment header constitutes a field. (4) In a BASIC file, a string of characters preceded by a return character and terminated by a return character. A field is written to a file by each PRINT statement not terminated by a semicolon. The INPUT command reads an entire field from a file.

filename: The name that identifies a file. The maximum character length of a filename and the rules for naming a file vary under different operating systems.

filter: A program or "mask" that alters data in accordance with specific criteria, a formula, or an algorithm.

firmware: Programs stored permanently in read-only memory (ROM). Such programs (for example, the Applesoft Interpreter and the Monitor program) are built into the computer at the factory. They can be executed at any time but cannot be modified or erased from main memory.

fixed: Describes blocks that are not movable in memory once allocated; also called *unmovable.* Program segments that must not be moved are placed in fixed memory blocks. Opposite of **movable.**

flag: A variable whose value indicates whether some condition holds or whether some event has occurred. A flag is used to control the program's actions at a later time. The value of a flag is usually 0 or 1.

flush: To update an open file (write all information in the I/O buffer to a disk) without closing it.

font: A complete set of characters in one design, size, and style. In traditional typography usage, *font* may be restricted to a particular size and style or may comprise multiple sizes, or multiple sizes and styles, of a ypeface design.

font class: A group of fonts that all use the same method of implementing different font styles, such as italic or bold.

font family: A complete set of characters for one typeface design, including all styles and sizes of the characters in that font. For example, the Geneva font family includes 9-point to 36-point characters in italic, bold, outlined, and other styles.

font number: The number by which you identify a font to QuickDraw or the Font Manager.

format: (n.) (1) The form in which information is organized or presented. (2) The general shape and appearance of a printer's output, including page size, character width and spacing, line spacing, and so on. (v.) To divide a disk into tracks and sectors where information can be stored. Blank disks must be formatted before you can save information on them for the first time; synonymous with *initialize.*

free block: A memory block containing space available for allocation.

free-form synthesizer: The part of the Sound Tool Set used to make complex music and speech.

garbage: A string of meaningless characters that bears no resemblance to your document. It's an indication that your computer and peripheral device are using different transmission rates or data formats.

GB: See **gigabyte.**

gigabyte (GB): A unit of measurement equal to 1024 (2^{10}) megabytes. Compare **byte, kilobyte, megabyte.**

GrafPort record: A data record used by QuickDraw to establish a **graphics port.**

graphics port: A complete drawing environment in QuickDraw (data type GrafPort), including such elements as a bitmap, a character font, patterns for drawing and erasing, and other graphics characteristics. Sometimes called a *GrafPort.*

handle: A pointer to a master pointer, which designates a relocatable block in the heap by double indirection. See also **memory handle.**

hang: To cease operation because either an expected condition is not satisfied or an infinite loop is occurring. A computer that's hanging is called a *hung system*. Compare **crash.**

heap: The area of memory in which space is dynamically allocated and released on demand, using the Memory Manager.

hertz (Hz): The unit of frequency of vibration or oscillation, defined as the number of *cycles per second*. Named for the physicist Heinrich Hertz. The 6502 microprocessor used in the Apple II systems operates at a clock frequency of about 1 million hertz, or 1 megahertz (MHz). The 68000 microprocessor used in the Macintosh operates at 7.8336 MHz.

hexadecimal system: The representation of numbers in the base-16 system, using the ten digits 0 through 9 and the six letters A through F. For example, the decimal numbers 0, 1, 2, 3, 4, . . . 8, 9, 10, 11, . . . 15, 16, 17 would be shown in hexadecimal notation as 00, 01, 02, 03, 04, . . . 08, 09, 0A, 0B, . . . 0F, 10, 11. Hexadecimal numbers are easier for people to read and understand than are binary numbers, and they can be converted easily and directly to binary form. Each hexadecimal digit corresponds to a sequence of four binary digits, or *bits*. Hexadecimal numbers are usually preceded by a dollar sign ($).

highlight: To make something visually distinct. For example, when you select a block of text using a word processor, the selected text is highlighted—it appears as light letters on a dark background, rather than dark on light. Highlighting is accomplished by inverting the display.

high-order: (adj.) Describes the most significant part of a numerical quantity. In normal representation, the *high-order bit* of a binary value is in the leftmost position; likewise, the *high-order byte* of a binary word or longword quantity consists of the leftmost eight bits. Compare **low-order.**

high-order byte: The more significant half of a memory address or other two-byte quantity. In the 6502 microprocessor used in the Apple II family of computers, the low-order byte of an address is usually stored first, and the high-order byte second. In the 68000 microprocessors used in the Macintosh family, the high-order byte is stored first. Compare **low-order byte.**

horizontal blanking interval: The time between the display of the rightmost pixel on one line and the leftmost pixel on the next line.

Hz: See **hertz.**

IC: See **integrated circuit.**

icon: An image that graphically represents an object, a concept, or a message. Icons on the outside of the computer can be used to show you where to plug cables, such as the disk drive icon on the back panel that marks the disk drive connector. Screen icons in mouse-based applications represent disks, documents, application programs, or other things you can select and open. A screen icon is a 32-by-32-bit image.

index register: A register in a computer processor that holds an index for use in indexed addressing. The 6502 and 65C816 microprocessors used in the Apple II family of computers have two index registers, called the *X register* and the *Y register*. The 68000 microprocessor used in Macintosh-family computers has 16 registers that can be used as index registers.

information window: The window that appears when you select an icon and choose Get Info from the File menu. It supplies information such as size, type, and date, and it includes a comment box for adding information.

insertion point: (1) The place in a document where something will be added, represented by a blinking vertical bar. You select the insertion point by clicking where you want to make the change in the document. (2) An empty selection range.

Installer: A utility program that lets you choose an Installation script for updating your system software or adding resources.

integrated circuit (IC): An electronic circuit—including components and interconnections—entirely contained in a single piece of semiconducting material, usually silicon. Often referred to as a *chip*.

interface: (n.) (1) The point at which independent systems or diverse groups interact. The devices, rules, or conventions by which one component of a system communicates with another. Also, the point of communication between a person and a computer. (2) The part of a program that defines constants, variables, and data structures, rather than procedures. In C, the compile-time and run-time linkage between your program and Toolbox routines. (3) The equipment that accepts electrical signals from one part of a computer system and renders them into a form that can be used by another part. (4) Hardware or software that links the computer to a device. (v.) To convert signals from one form to another and pass them between two pieces of equipment.

interrupt: (1) An electronic attention-getter; a signal sent to the microprocessor that is intended to force the microprocessor to stop its current activity and accept input from the device that sent the interrupt. (2) A temporary suspension in the execution of a program that allows the computer to perform some other task, typically in response to a signal from a peripheral device or other source external to the computer. (3) An exception that's signaled to the processor by a device, to notify the processor of a change in condition of the device, such as the completion of an I/O request.

IRQ: A 65C816 signal line that, when activated, causes an interrupt request to be generated.

item: In dialog and alert boxes, a control, icon, picture, or piece of text, each displayed inside its own display rectangle. See also **menu item.**

item list: A list of information about all the items in a dialog or alert box.

IWM: "Integrated Woz Machine"; the custom chip that controls the Apple 3.5-inch disk drives.

job: A process that can be stopped, restarted, and moved between foreground and background processing from the C shell.

job dialog: A dialog box that sets information about one printing job; associated with the Print command.

journaling mechanism: A mechanism that allows a program to feed events to the Toolbox Event Manager from some source other than the user.

justification: The horizontal placement of lines of text relative to the edges of the rectangle in which the text is drawn.

K: See **kilobyte.**

Kbit: See **kilobit.**

Kbyte: See **kilobyte.**

kern: To draw part of a character so that it overlaps an adjacent character.

kernel: (1) The central part of an operating system. ProDOS 16 is the kernel of the Apple IIGS operating system. (2) A program that manages the system hardware. For example, the kernel manages files, communicates with peripherals, and handles other low-level resource management tasks.

keyboard event: An event generated when the user presses a character key on the keyboard. A *key-down event* is generated when the user presses a character key; a *key-up event* is generated when the user releases a character key. *Auto-key events* are repeatedly generated when the user holds down a character key.

key-down event: An event generated when the user presses a character key on the keyboard or keypad. Compare **key-up event.**

keystroke equivalent: A keystroke that activates a control just as if the user had clicked in the control.

key-up event: An event generated when the user releases a character key on the keyboard or keypad. Compare **key-down event.**

kHz: See **kilohertz.**

kilobit (Kbit): A unit of measurement, 1024 bits, commonly used in specifying the capacity of memory integrated circuits. Not to be confused with **kilobyte.**

kilobyte (K): A unit of measurement consisting of 1024 (2^{10}) bytes. Thus, 64K memory equals 65,536 bytes. The abbreviation *K* can also stand for the number 1024, in which case *Kbyte* is used for kilobyte. See also **megabyte.**

kilohertz (kHz): A unit of measurement of frequency, equal to 1000 **hertz.** See also **megahertz.**

language card: Memory with addresses between $D000 and $FFFF on any Apple II–family computer. It includes two RAM banks in the $Dxxx space, called **bank-switched memory.** The language card was originally a peripheral card for slot 0 of the 48K Apple II or Apple II Plus that expanded memory capacity to 64K and provided space for an additional dialect of BASIC. The language card was also necessary for these machines to use ProDOS.

least significant bit: The binary digit in a number or data byte that contributes the smallest quantity to the value of the number; usually written at the right end of the number. Compare **most significant bit.**

list record: The internal representation of a list, where the List Manager stores all the information it requires for its operations on that list.

load: To transfer information from a peripheral storage medium (such as a disk) into main memory for use; for example, to transfer a program into memory for execution.

local coordinate system: The coordinate system local to a GrafPort, imposed by the boundary rectangle defined in its bitmap.

lock: (1) To prevent a memory block from being moved or temporarily purged. A block may be locked or unlocked by the Memory Manager. (2) To temporarily prevent a relocatable block from being moved during heap compaction.

logical end-of-file: The position of one byte past the last byte in a file; equal to the actual number of bytes in the file. Compare **physical end-of-file.**

logical operator: An operator, such as AND, that combines logical values to produce a logical result, such as true or false; sometimes called a *Boolean operator.*

low-order: (adj.) Describes the least significant part of a numerical quantity. In normal representation, the *low-order bit* of a binary number is in the rightmost position; likewise, the *low-order byte* of a binary word or longword quantity consists of the rightmost eight bits. Compare **high-order.**

low-order byte: The less significant half of a memory address or other two-byte quantity. In the 6502 microprocessor used in the Apple II family of computers, the low-order byte of an address is usually stored first, and the high-order byte second. The opposite is true for Macintosh computers. Compare **high-order byte.**

Macintosh: A family of Apple computers, including the Macintosh 128K, Macintosh 512K, Macintosh 512K enhanced, Macintosh Plus, Macintosh SE, and Macintosh II. Macintosh computers have high-resolution screens and use mouse devices for choosing commands and for drawing pictures.

Macintosh Programmer's Workshop (MPW): Apple's software development environment for the Macintosh family.

macro: (1) A user-defined command that tells an application to carry out a series of commands when you type the macro. (2) A recorded sequence of characters and commands, identified by a name and possibly triggered by a **keystroke.** (3) A single keystroke or command that a program replaces with several keystrokes or commands. For example, the APW Editor allows you to define macros that execute several editor keystroke commands; the APW Assembler allows you to define macros that execute instructions and directives. Macros are almost like higher-level instructions, making assembly-language programs easier to write and complex keystrokes easier to execute.

MB: See **megabyte.**

Mbit: See **megabit.**

megabit (Mbit): A unit of measurement equal to 1,048,576 (2^{16}) bits, or 1024 kilobits, commonly used in specifying the capacity of memory ICs. Not to be confused with **megabyte.**

megabyte (MB): A unit of measurement equal to 1024 kilobytes, or 1,048,576 bytes. See also **kilobyte.**

megahertz (MHz): One million **hertz.** See also **kilohertz.**

memory handle: The identifying number of a particular block of memory. It is a pointer to the master pointer to the memory block. A handle rather than a simple pointer is needed to reference a movable memory block.

menu: A list of choices presented by a program, from which you can select an action. In the desktop interface, menus appear when you point to and press menu titles in the **menu bar.** Dragging through the menu and releasing the mouse button while a command is highlighted chooses that command.

menu bar: The horizontal strip at the top of the screen that contains menu titles.

menu definition procedure: A procedure called by the Menu Manager when it needs to perform type-dependent operations on a particular menu (for example, when it needs to draw the menu).

menu item: A choice in a menu, usually a command to the current application. See also **item.**

Menu Manager: The part of the toolbox that deals with setting up menus and letting the user choose from them.

menu record: The internal representation of a menu, where the Menu Manager stores all the information it needs for its operations on that menu.

menu template: Data structure used to define menus, menu commands, and menu bars to the Menu Manager.

menu title: A word, phrase, or icon in the menu bar that designates one menu. Pressing on the menu title causes the title to be highlighted and its menu to appear below it.

m flag: One of three flag bits in the 65C816 processor that programs use to control the processor's operating modes. In **native mode,** the setting of the m flag determines whether the accumulator is 8 bits wide or 16 bits wide. See also **e flag, x flag.**

microprocessor: An integrated circuit on the computer's main circuit board. The microprocessor carries out software instructions by directing the flow of electrical impulses through the computer. The microprocessor is the **central processing unit (CPU)** of the microcomputer. Examples are the 6502 or 65C02 microprocessor used in the Apple IIe, the 65C816 microprocessor used in the Apple IIGS, and the 68000 microprocessor used in the Macintosh Plus.

MIDI: Acronym for *Musical Instrument Data Interface;* a standard interface for electronically created music.

millisecond (ms): One-thousandth of a second.

mnemonic: A type of abbreviation consisting of a series of letters and/or numbers that represent a longer or more complicated name or title. A mnemonic is characterized by being relatively easy to remember.

modifier key: A general term for a key that generates no keyboard events of its own but changes the meaning of other keys or mouse actions; for example, Caps Lock, Command, Control, Apple, Option, and Shift. When you hold down or engage a modifier key while pressing another key, the combination makes that other key behave differently. Sometimes called a *control key*. Compare **character key.**

most significant bit: The binary digit in a number or data byte that contributes the largest quantity to the value of the number; usually written at the left end of the number. For example, in the binary number 10110 (decimal value 22), the leftmost bit has the decimal value 16 (2^4). Compare **least significant bit.**

mouse: A small device you move around on a flat surface next to your computer. The mouse controls a pointer on the screen whose movements correspond to those of the mouse. You use the pointer to select operations, to move data, and to draw with in graphics programs.

mouse button: The button on the top of the mouse. In general, pressing the mouse button initiates some action on whatever is under the pointer, and releasing the button confirms the action. Compare **button.**

mouse-down event: An event generated when the user presses the mouse button.

mouse event: An event generated when the user presses and releases the mouse button. A *mouse-down event* is generated when the user presses the mouse button. A *mouse-up event* is generated when the user releases the mouse button.

mouse-up event: An event generated when the user releases the mouse button.

movable: A memory block attribute, indicating that the Memory Manager is free to move the block. Opposite of **fixed.** Only position-independent program segments may be in movable memory blocks. A block is made movable or fixed through Memory Manager calls.

move: To change the location of a memory block. The Memory Manager may move blocks to consolidate memory space.

MPW: See **Macintosh Programmer's Workshop.**

nanosecond (ns): One-billionth of a second.

native mode: The 16-bit operating configuration of the 65C816 microprocessor.

nibble: A unit of data equal to half a byte, or four bits. A nibble can hold any value from 0 to 15.

NOT: A unary logical operator that produces a TRUE result if its operand is false, and a FALSE result if its operand is true. Compare **AND, exclusive OR, OR.**

null: (1) An undefined value. Null is different from 0; 0 is a value just like other numbers, whereas null means no value at all (of the expected type). A *null string* does not contain anything. For example, `' '` is not a null string because it contains a space character; `''` represents a null string. (2) Any character or character code that has no meaning to the operating system or program interpreting it. (3) A type of attention cycle.

null event: An event reported when there are no other events to report.

open: To make available. You open files or documents in order to work with them. A file may not be read from or written to until it is open. In the desktop interface, opening an icon causes a window with the contents of that icon to come into view. You may then perform further actions in the window when it's active.

option: (1) Something chosen or available as a choice; for instance, one of several check box or radio button options. (2) An argument whose provision is optional.

Option key: A modifier key that gives a different meaning or action to another key you press or to mouse actions you perform. For example, you can use it to type foreign characters or special symbols contained in the optional character set. On the Apple IIGS and some models of the Apple IIe, the Option key replaces the Solid Apple key.

OR: A logical operator that produces a TRUE result if either or both of its operands are true, and a FALSE result if both of its operands are false. Compare **AND, exclusive OR, NOT.**

out-of-memory queue: A queue maintained by the Memory Manager. Queue elements **(out-of-memory routines)** refer to code to be executed when the Memory Manager detects an out-of-memory condition.

out-of-memory routines: Code executed by the Memory Manager when it detects an out-of-memory condition. The **out-of-memory queue** consists of a list of these routines.

override: To modify or cancel an instruction by issuing another one. For example, you might override a DIP switch setting on a printer with an escape sequence.

page: (1) The text and/or graphics that fits on a sheet of paper when printed, depending on the page format. (2) A screenful of information on a video display. In the Apple II family of computers, a page consists of 24 lines of 40 or 80 characters each. (3) (usually *Page*) An area of main memory containing text or graphic information being displayed on the screen. (4) A segment of main memory 256 bytes long and beginning at an address that is an even multiple of 256. Memory blocks whose starting addresses are an even multiple of 256 are said to be *page-aligned.*

page zero: See **zero page.**

parameter: (1) A value passed to or from a function or other routine. (2) An argument that determines the outcome of a command. For example, in the command `write(n,msg)`, `n` and `msg` are parameters.

parameter block: (1) A data structure used to transfer information between applications and certain Operating System routines. (2) A set of contiguous memory locations, set up by a calling program to pass parameters to and receive results from an operating-system function that it calls. Every call to ProDOS 16, to the APW Shell, or to SmartPort must include a pointer to a properly constructed parameter block.

parameter list: The list of characteristics whose value or condition determines the precise execution of a SCSI command.

Pascal: A high-level programming language with statements that resemble English phrases. Pascal was designed to teach programming as a systematic approach to problem solving. Named for the philosopher and mathematician Blaise Pascal.

Pascal-compatible function: A function written in Pascal that can be declared in C using the `pascal` specifier.

password: (1) A secret word that gives you, but no one else, access to your data or to messages sent to you through an information service. (2) A unique word or set of characters that must be entered before a registered user at a workstation can access a volume on a server.

password field: A field that does not echo user input, allowing protected data entry. Your program can specify the echo character; the default echo character is the asterisk (`*`).

paste: To place the contents of the Clipboard—whatever was last cut or copied—at the insertion point.

pattern: An 8-by-8-bit image used to define a repeating design (such as stripes) or tone (such as gray).

physical end-of-file: The position of one byte past the last allocation block of a file; equal to one more than the maximum number of bytes the file can contain. Compare **logical end-of-file.**

picture: (1) In HyperCard, any graphic or part of a graphic created with a Paint tool. Also, an imported MacPaint document or part of a MacPaint® document. (2) A saved sequence of QuickDraw drawing commands (and, optionally, picture comments) that you can play back later with a single procedure call. Also, the image resulting from these commands.

pixel: Short for *picture element;* the smallest dot you can draw on the screen. Also a location in video memory that corresponds to a point on the graphics screen when the viewing window includes that location. In the Macintosh monochrome display, each pixel can be either black or white, so it can be represented by a bit; thus, the display is said to be a **bitmap.** For color or gray-scale video, several bits in RAM may represent the image; in the Super Hi-Res display on the Apple IIGS, each pixel is represented by either two or four bits. Thus, the display is not a bitmap but rather a pixel map.

pointer: (1) A small shape on the screen that follows the movement of the mouse or shows where your next action will take place. The pointer can be an arrow, an I-beam, a crossbar, or a wristwatch. (2) An item of information consisting of the memory address of some other item. For example, Applesoft BASIC maintains internal pointers to the most recently stored variable, the most recently typed program line, and the most recently read data item, among other things. The 6502 uses one of its internal registers as a pointer to the top of the stack.

pop-up menu: A menu that "pops" out of its display rectangle when selected by the user. The two types of pop-up menus, **type 1** and **type 2 pop-up menus,** have different maximum sizes.

prefix: (1) The first part of a pathname—the name of the disk and, if you like, the name of a subdirectory. Applications that ask you to type a pathname usually let you set a prefix so you don't have to type the complete pathname every time you want to work with a document on a particular disk or in a particular subdirectory. Once the prefix is set, all you do is type the rest of the pathname. (2) A designation for a place that an application can store files. Many applications require the prefix to be the same as the pathname. Some applications allow you to set the prefix from within the application.

prefix number: A code used to represent a particular prefix. Under ProDOS 16, there are nine prefix symbols, consisting of the numerals 0 through 7 and the asterisk followed by a slash: `0/`, `1/`, ... `7/`, and `*/`.

press: (1) To position the pointer on something on the screen and then hold down the mouse button without moving the mouse. (2) To push a key down and then release it; you hold a key down only if you want to repeat a character or if you are using a modifier key with another key.

printed-circuit board: A hardware component of a computer or other electronic device, consisting of a flat, rectangular piece of rigid material, commonly fiberglass, to which integrated circuits and other electronic components are connected.

purge: To temporarily deallocate a memory block. The Memory Manager purges a block by setting its master pointer to NIL(0). All handles to the pointer are still valid, so the block can be reconstructed quickly. Compare **dispose.**

purgeable: A memory block attribute, indicating that the Memory Manager may purge the block if it needs additional memory space. Purgeable blocks have different **purge levels,** or priorities for purging; these levels are set by Memory Manager calls.

purgeable block: A relocatable block that can be purged from the heap.

purge level: An attribute of a memory block that sets its priority for purging. A purge level of 0 means that the block cannot be purged.

Quagmire register: On the Apple IIGS, the name given to the eight bits consisting of the speed control bit and the shadowing bits. Although Quagmire is not a real register, the Monitor program allows you to access those bits as if they were in a single register.

queue: A list in which entries are added at one end and removed at the other, causing entries to be removed in first-in, first-out (FIFO) order. Compare **stack.**

QuickDraw: The part of the toolbox that performs all graphic operations on the Macintosh screen.

quoting mechanism: Special syntax in the command line that tells the shell to interpret metacharacters literally, or to control the type of substitution allowed in the command.

RAM: See **random-access memory.**

random-access memory (RAM): The part of the computer's memory that stores information temporarily while you're working on it. A computer with 512K RAM has 512 kilobytes of memory available to the user. Information in RAM can be referred to in an arbitrary or random order, hence the term *random-access.* (As an analogy, a book is a random-access storage device in that it can be opened and read at any point.) RAM can contain both application programs and your own information. Information in RAM is temporary, gone forever if you switch the power off without saving it on a disk or other storage medium. An exception is the *battery RAM,* which stores settings such as the time and which is powered by a battery. (Technically, the read-only memory [ROM] is also *random access,* and what's called RAM should correctly be termed *read-write memory.*) Compare **read-only memory.**

read-only memory (ROM): Memory whose contents can be read but not changed; used for storing **firmware.** Information is placed into read-only memory once, during manufacture. It remains there permanently, even when the computer's power is turned off. Compare **random-access memory.**

read-write memory: Memory whose contents can be both read and changed (or *written to*). The information contained in read-write memory is erased when the computer's power is turned off and is permanently lost unless it has been saved on a disk or other storage device. Used synonymously with *random-access memory*. Compare **read-only memory.**

reference type: Indicates whether a storage location contains a pointer, a handle, or a resource ID for an object.

release: That portion of a sound **envelope** during which the note dies away to silence. See also **ADSR.**

relocatable: Characteristic of a load segment or other **OMF** program code that includes no references to specific address and so can be relocated at load time. A relocatable segment can be static, dynamic, or position independent. It consists of a code image followed by a relocation dictionary. Compare **absolute.**

relocatable block: A block that can be moved within the heap during compaction.

resource: Collection of data managed by the Resource Manager for other applications.

resource compiler: A program that creates resources from a textual description. The MPW Resource Compiler is named *Rez*.

resource file: A collection of one or more **resources.** The Resource Manager provides routines for accessing and updating resources in a resource file.

resource fork: The part of a file that contains data used by an application, such as menus, fonts, and icons. Sometimes called a *resource file.*

resource ID: A number that uniquely identifies a **resource** within the context of its **resource type.** The Resource Manager provides facilities to assign unique resource IDs. Compare **resource name.**

resource map: In a resource file, data that is read into memory when the file is opened and that, given a resource specification, leads to the corresponding resource data.

resource name: A series of characters that uniquely identify a **resource** within the context of its **resource type.** Note that resource names are not maintained by the system; it is your program's responsibility to assign and manage them. Compare **resource ID.**

resource type: A class of **resources** that share a common data layout. Individual instances of **resources** of a given type are identified by their unique **resource ID** or **resource name.**

ROM: See **read-only memory.**

run item: An element in the **run queue.** Run items specify program code to be executed by the Desk Manager at regular intervals.

run queue: A queue maintained by the Desk Manager that contains elements **(run items)** that specify code to be executed at regular intervals.

sample rate: The number of sound samples the Apple IIGS **Digital Oscillator Chip** plays per second.

scroll: (1) To move a document or directory in its window so that a different part of it is visible. (2) To move all the text on the screen upward or downward, and in some cases sideways.

scroll arrow: An arrow at either end of a scroll bar. Clicking a scroll arrow moves a document or directory one line. Pressing a scroll arrow moves a document continuously.

scroll bar: A rectangular bar that may be along the right or bottom of a window. Clicking or dragging in the scroll bar causes your view of the document to change.

scroll box: The white box in a scroll bar. The position of the scroll box in the scroll bar indicates the position of what's in the window relative to the entire document.

select: (v.) To designate where the next action will take place. To select using a mouse, you click an icon or drag across information. In some applications, you can select items in menus by typing a letter or number at a prompt, by using a combination keypress, or by using arrow keys. (n.) A command to a device such as a printer to place it into a condition to receive data.

selection: (1) The information or items that will be affected by the next command. The selection is usually highlighted. (2) A series of characters, or a character position, at which the next editing operation will occur. Selected characters in the active window are inversely highlighted. Also called *selection range.*

shadowing: (1) The process by which any changes made to one part of the Apple IIGS memory are automatically and simultaneously copied into another part. When shadowing is on, information written to bank $00 or $01 is automatically copied into equivalent locations in bank $E0 or $E1. Likewise, any changes to bank $E0 or $E1 are immediately reflected in bank $00 or $01. (2) A process through which the SCSI card takes over an additional slot to work with ProDOS in supporting four external device ports.

6502: The microprocessor used in the Apple II, the Apple II Plus, and early models of the Apple IIe. The 6502 is a MOS device with 8-bit data registers and 16-bit address registers.

65C02: A CMOS version of the 6502; the microprocessor used in the Apple IIc and Apple IIe.

65C816: The microprocessor used in the Apple IIGS. The 65C816 is a CMOS device with 16-bit data registers and 24-bit address registers.

64K Apple II: Any standard Apple II that has at least 64K of RAM. That includes the Apple IIc, the Apple IIe, and an Apple II or Apple II Plus with 48K of RAM and the language card installed.

size box: A box in the lower-right corner of some active windows. Dragging the size box resizes the window.

slot: A narrow socket inside some models of Apple computers for connecting circuit boards known as *interface cards;* each card handles communication between the computer and a peripheral device, sending and receiving data through a port or connector on the outside of the computer.

slot number: A way an application might ask you to describe the location of a peripheral device. In some models of the Apple II, there are seven general-purpose slots on the main circuit board for connecting peripheral devices to the computer. They are numbered from 1 to 7 with 1 on the left as you face the front of the computer. If your device is connected to a port instead of a slot, you can still use the application by typing the slot number that corresponds to the port.

soft switch: A means of changing some feature of the computer from within a program. For example, DIP switch settings on ImageWriter printers can be overridden with soft switches. Specifically, a soft switch is a location in memory that produces some special effect whenever its contents are read or written. Also called a *software switch*.

software pirate: A person who copies applications without the permission of the author. To copy software without permission is illegal.

sound buffer: A block of memory from which the sound generator reads the information to create an audio waveform.

stack: In a computer, a portion of memory that is used for temporary storage of operating data during operation of a program. The data on the stack are added (pushed) and removed (pulled or popped) in last-in, first-out (LIFO) order. *The stack* usually refers to the particular stack pointed to by the 65C816's **stack register.** Compare **queue.**

stack register: A hardware register in the 65C816 processor that contains the address of the top of the processor's **stack.**

standard Apple II: Any computer in the Apple II family except the Apple IIGS. That includes the Apple II, the Apple II Plus, the Apple IIe, and the Apple IIc.

start up: To get the system running. Starting up is the process of first reading an operating-system program from the disk and then running an application program. Synonymous with **boot.**

startup disk: A disk with all the necessary program files—such as the Finder and System files contained in the System Folder for the Macintosh—to set the computer into operation. Sometimes called a *boot disk.*

startup drive: The disk drive from which you started your application.

sustain: That portion of a sound **envelope** during which the note maintains a fairly constant loudness, before it dies away. See also **ADSR.**

synthesizer: (1) A hardware device capable of creating sound digitally and converting it into an analog waveform that you can hear. (2) A program that interprets Sound Tool Set commands and produces sound.

system software: The component of a computer system that supports application programs by managing system resources such as memory and I/O devices.

tab: (1) Short for *tabulator;* on typewriter keyboards, a key that allows you set automatic stops (*tab stops*) or margins for columns, as in a table of figures. (2) An ASCII character that commands a device such as a printer to start printing at a preset location (a *tab stop*). There are two such characters: horizontal tab (hex 09) and vertical tab (hex 0B). The horizontal tab character gives the same action as pressing the tab key on a typewriter.

Tab key: A key that, when pressed, generates the horizontal tab character. The key's action is to move the insertion point or cursor to the next tab marker, or, in a dialog box with more than one place to enter information, to the next rectangle. The Tab key thus works essentially like a typewriter tab key.

target control: That control that is currently the recipient of user actions (keystrokes and menu items).

tear-off menu: Any menu that you can detach from the menu bar by pressing the menu title and dragging beyond the menu's edge. The torn-off menu appears in a window or a mini-window on the desktop.

TextEdit record: Describes a TextEdit user session, whether or not that session is managed as a control.

toolbox: A collection of built-in routines that programs can call to perform many commonly needed functions. Functions within the Apple IIGS Toolbox are grouped into **tool sets.**

tool set: A group of related routines (usually in firmware) that perform necessary functions or provide programming convenience. They are available to applications and system software. The Memory Manager, the System Loader, and QuickDraw II are Apple IIGS tool sets.

type 1 pop-up menu: A **pop-up menu** that does not become larger than its window. Compare **type 2 pop-up menu.**

type 2 pop-up menu: A **pop-up menu** that becomes larger than its window if necessary to display its menu items. Compare **type 1 pop-up menu.**

type-ahead buffer: A **buffer** that accepts and holds characters that are typed faster than the computer can process them.

unload: To remove a load segment from memory. To unload a segment, the System Loader does not actually "unload" anything; it calls the Memory Manager to either **purge** or **dispose** of the memory block in which the code segment resides. The loader then modifies the Memory Segment Table to reflect the fact that the segment is no longer in memory.

unlock: To allow a relocatable block to be moved during heap compaction. Compare **lock.**

unmovable: See **fixed.**

unpurgeable: Having a **purge level** of 0. The Memory Manager is not permitted to purge memory blocks whose purge level is 0.

unpurgeable block: A relocatable block that can't be purged from the heap.

update event: An event generated by the Window Manager when a window's contents need to be redrawn.

User ID: An identification number that specifies the owner of every memory block allocated by the Memory Manager.

version: A number indicating the release edition of a particular piece of software. Version numbers for most system software (such as ProDOS 16 and the System Loader) are available through function calls.

void: In C, a data type used to declare a function that does not return a value.

waveform: The shape of a wave (a graph of a wave's amplitude over time).

waveform description: A sequence of bytes describing a waveform.

wildcard character: A character that may be used as shorthand to represent a sequence of characters in a pathname. A common wildcard character is the asterisk (*). As an example, if you were to request a listing of *.TEXT files in a particular application, you would see a list of all files ending with the suffix TEXT. In APW, the equal sign (=) and the question mark (?) can be used as wildcard characters.

window: (1) The area that displays information on a desktop; you view a document through a window. You can open or close a window, move it around on the desktop, and sometimes change its size, scroll through it, and edit its contents. (2) The portion of a collection of information (such as a document, picture, or worksheet) that is visible in a viewport on the display screen. Each window is internally represented in a window record.

window definition function: A function called by the Window Manager when it needs to perform certain type-dependent operations on a window (for example, drawing the window frame).

Window Manager: The part of the toolbox that provides routines for creating and manipulating windows.

Window Manager port: A GrafPort that has the entire screen as its PortRect and is used by the Window Manager to draw window frames.

word: (1) The computer's native unit of data. The Macintosh II uses a 32-bit word. A NuBus™ word is 32 bits long; a half-word is 16 bits. An SE Bus or 68000 word is 16 bits long; a half-word is 8 bits. For the Apple IIGS, a word is 16 bits (2 bytes) long. (2) For the shell and other programs, a string of nonblank characters bounded by the space character, the tab character, or the beginning or the end of the input line.

word wrap: The automatic continuation of text from the end of one line to the beginning of the next. Word wrap lets you avoid pressing the Return key at the end of each line as you type.

x flag: One of three flag bits in the 65C816 processor that programs use to control the processor's operating modes. In **native mode,** the setting of the x flag determines whether the index registers are 8 bits wide or 16 bits wide. See also **e flag, m flag.**

zero page: The first page (256 bytes) of memory in a standard Apple II computer (or in the Apple IIGS when running a standard Apple II program); also called *page zero*. Because the high-order byte of any address in this page is zero, only the low-order byte is needed to specify a zero-page address. This makes zero-page locations more efficient to address, in both time and space, than locations in any other page of memory. Compare **direct page.**

zoom box: A small box with a smaller box enclosed in it found on the right side of the **title bar** of some windows. Clicking the zoom box expands the window to its maximum size; clicking it again returns the window to its original size.

Index

J

K

L

M

N

Q

R

S

T

U

V

W, X, Y

Z

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh® computers and Microsoft® Word software. Proof and final pages were created on the Apple LaserWriter® printers. Line art was created using Adobe Illustrator™. POSTSCRIPT®, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type and display type are Apple's corporate font, a condensed version of Garamond. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.