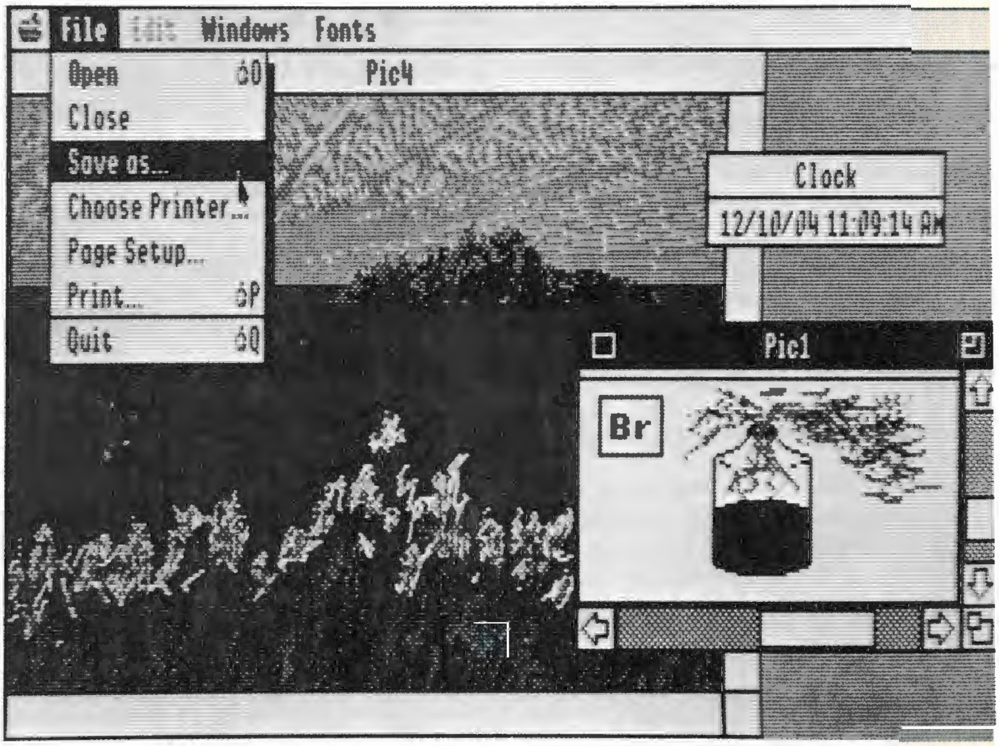
Apple® II

Apple IIGS Toolbox Reference

Volume 1



Addison-Wesley Publishing Company, Inc.
Reading, Massachusetts Menlo Park, California New York Don Mills,
Ontario Wokingham, England Amsterdam Bonn Sydney Singapore
Tokyo Madrid Bogotá Santiago San Juan

 APPLE COMPUTER, INC.

Simultaneously published in the United States and Canada.

ISBN 0-201-17746-3
ABCDEFGHIJ-DO-8987
First printing, December 1987

WARRANTY INFORMATION

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

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

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

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

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

Contents

Volume 1

Contents

Volume 2

Figures and tables

Preface **Roadmap to the Apple IIGS Technical Manuals xxiii**

Chapter 2 **Using the Apple IIGS Tool Sets 2-1**

Chapter 3 **Apple Desktop Bus Tool Set 3-1**

Chapter 4 **Control Manager 4-1**

Chapter 9 Integer Math Tool Set 9-1

Chapter 10 LineEdit Tool Set 10-1

Chapter 11 List Manager 11-1

Chapter 14 Miscellaneous Tool Set 14-1

Chapter 15 Print Manager 15-1

Preface

Roadmap to the Apple IIGS Technical Manuals

The Apple® IIGS™ personal computer has many advanced features, making it more complex than earlier models of the Apple II. To describe it fully, Apple has produced a suite of technical manuals. Depending on the way you intend to use the Apple IIGS, you may need to refer to a select few of the manuals, or you may need to refer to most of them.

The technical manuals are listed in Table P-1. Figure P-1 is a diagram showing the relationships among the different manuals.

Table P-1
Apple IIGS technical manuals

Title	Subject
Technical Introduction to the Apple IIGS	What the Apple IIGS is
Apple IIGS Hardware Reference	Machine internals—hardware
Apple IIGS Firmware Reference	Machine internals—firmware
Programmer's Introduction to the Apple IIGS	Concepts and a sample program
Apple IIGS Toolbox Reference, Volume 1	How the tools work and some toolbox specifications
Apple IIGS Toolbox Reference, Volume 2	More toolbox specifications
Apple IIGS Programmer's Workshop Reference	The development environment
Apple IIGS Programmer's Workshop Assembler Reference	Using the APW Assembler
Apple IIGS Programmer's Workshop C Reference	Using C on the Apple IIGS
Apple IIGS ProDOS 16 Reference	Apple IIGS operating system and loader
ProDOS 8 Technical Reference Manual	Standard Apple II operating system
Human Interface Guidelines: The Apple Desktop Interface	Guidelines for the desktop interface
Apple Numerics Manual	Numerics for all Apple computers

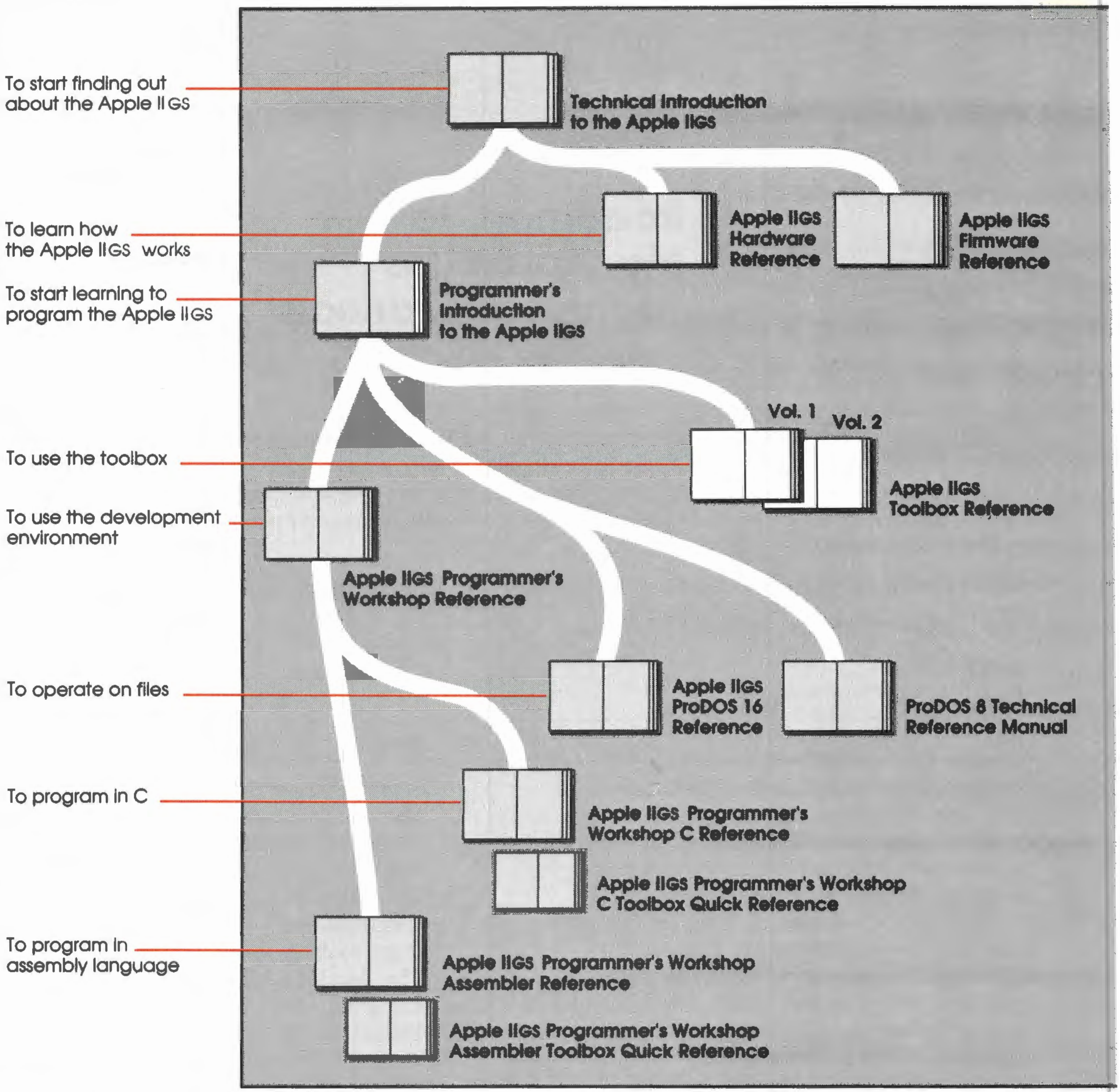

To start finding out about the Apple IIGS
Technical Introduction to the Apple IIGS
To learn how the Apple IIGS works
Apple IIGS Hardware Reference
Apple IIGS Firmware Reference
To start learning to program the Apple IIGS
Programmer's Introduction to the Apple IIGS
Vol. 1
Vol. 2
To use the toolbox
Apple IIGS Toolbox Reference
To use the development environment
Apple IIGS Programmer's Workshop Reference
To operate on files
Apple IIGS ProDOS 16 Reference
ProDOS 8 Technical Reference Manual
To program in C
Apple IIGS Programmer's Workshop C Reference
Apple IIGS Programmer's Workshop C Toolbox Quick Reference
To program in assembly language
Apple IIGS Programmer's Workshop Assembler Reference
Apple IIGS Programmer's Workshop Assembler Toolbox Quick Reference

Figure P-1
Roadmap to the technical manuals

Introductory manuals

These books are introductory manuals for developers, computer enthusiasts, and other Apple IIGS owners who need technical information. As introductory manuals, their purpose is to help the technical reader understand the features of the Apple IIGS, particularly the features that are different from other Apple computers. Having read the introductory manuals, the reader will refer to specific reference manuals for details about a particular aspect of the Apple IIGS.

The technical introduction

The *Technical Introduction to the Apple IIGS* is the first book in the suite of technical manuals about the Apple IIGS. It describes all aspects of the Apple IIGS, including its features and general design, the program environments, the toolbox, and the development environment.

Whereas the *Apple IIGS Owner's Guide* is an introduction from the point of view of the user, the *Technical Introduction* describes the Apple IIGS from the point of view of the program. In other words, it describes the things the programmer has to consider while designing a program, such as the operating features the program uses and the environment in which the program runs.

The programmer's introduction

When you start writing Apple IIGS programs, the *Programmer's Introduction to the Apple IIGS* provides the concepts and guidelines you need. It is not a complete course in programming, only a starting point for programmers writing applications that use the Apple Desktop Interface (with windows, menus, and the mouse). It introduces the routines in the Apple IIGS Toolbox and the program environment they run under. It includes a sample event-driven program that demonstrates how a program uses the toolbox and the operating system. An **event-driven program** waits in a loop until it detects an event such as a click of the mouse button.

Machine reference manuals

There are two reference manuals for the machine itself: the *Apple IIGS Hardware Reference* and the *Apple IIGS Firmware Reference.* These books contain detailed specifications for people who want to know exactly what's inside the machine.

The hardware reference manual

The *Apple IIGS Hardware Reference* is required reading for hardware developers, and it will also be of interest to anyone else who wants to know how the machine works. Information for developers includes the mechanical and electrical specifications of all connectors, both internal and external. Information of general interest includes descriptions of the internal hardware, which provide a better understanding of the machine's features.

The firmware reference manual

The *Apple IIGS Firmware Reference* describes the programs and subroutines that are stored in the machine's read-only memory (ROM), with two significant exceptions: Applesoft BASIC and the toolbox, which have their own manuals. The *Firmware Reference* includes information about interrupt routines and low-level I/O subroutines for the serial ports, the disk port, and for the Apple Desktop Bus™ interface, which controls the keyboard and the mouse. The *Firmware Reference* also describes the Monitor, a low-level programming and debugging aid for assembly-language programs.

The toolbox manuals

Like the Macintosh, the Apple IIGS has a built-in toolbox. This volume of the *Apple IIGS Toolbox Reference* introduces concepts and terminology and tells how to use some of the tools. The *Apple IIGS Toolbox Reference,* Volume 2, contains information about the rest of the tools and also tells how to write and install your own tool set.

Of course, you don't have to use the toolbox at all. If you only want to write simple programs that don't use the mouse, or windows, or menus, or other parts of the **desktop user interface,** then you can get along without the toolbox. However, if you are developing an application that uses the desktop interface, or if you want to use the Super Hi-Res graphics display, you'll find the toolbox to be indispensable.

The Programmer's Workshop manual

The Apple IIGS Programmer's Workshop (APW) is the development environment for the Apple IIGS computer. APW is a set of programs that enables developers to create and debug application programs on the Apple IIGS. The *Apple IIGS Programmer's Workshop Reference* includes information about the APW Shell, Editor, Linker, and utility programs; these are the parts of the workshop that all developers need, regardless of which programming language they use.

The APW reference manual describes the way you use the workshop to create an application and includes a sample program to show how this is done. In addition, this manual documents the APW Shell to provide the information necessary to write an APW utility or a language compiler for the workshop.

Included in the APW reference manual are complete descriptions of two standard Apple IIGS file formats: the **text file format** and the **object module format.** The text file format is used for all files written or read as "standard ASCII files" by Apple IIGS programs running under ProDOS 16. The object module format is used for the output of all APW compilers and for all files loadable by the Apple IIGS System Loader.

Programming-language manuals

Apple currently provides a 65816 assembler and a C compiler. Other compilers can be used with the workshop, provided that they follow the standards defined in the *Apple IIGS Programmer's Workshop Reference.*

There is a separate reference manual for each programming language on the Apple IIGS. Each manual includes the specifications of the language and of the Apple IIGS libraries for the language and describes how to use the assembler or compiler for that language. The manuals for the languages Apple provides are the *Apple IIGS Programmer's Workshop Assembler Reference* and the *Apple IIGS Programmer's Workshop C Reference.*

❖ *Note:* The *Apple IIGS Programmer's Workshop Reference* and the two programming-language manuals are available through the Apple Programmer's and Developer's Association.

Operating-system manuals

There are two operating systems that run on the Apple IIGS: ProDOS® 16 and ProDOS 8. Each operating system is described in its own manual: *ProDOS 8 Technical Reference Manual* and *Apple IIGS ProDOS 16 Reference.* ProDOS 16 uses the full power of the Apple IIGS. The ProDOS 16 manual describes the features of ProDOS and includes information about the System Loader, which works closely with ProDOS 16. If you are writing programs for the Apple IIGS, whether as an application programmer or a system programmer, you are almost certain to need the *ProDOS 16 Reference*.

ProDOS 8, previously just called *ProDOS,* is the standard operating system for most Apple II computers with 8-bit CPUs. It also runs on the Apple IIGS. As a developer of Apple IIGS programs, you need the *ProDOS 8 Technical Reference Manual* only if you are developing programs to run on 8-bit Apple II computers.

All-Apple manuals

In addition to the Apple IIGS manuals mentioned above, there are two manuals that apply to all Apple computers: *Human Interface Guidelines: The Apple Desktop Interface* and *Apple Numerics Manual.* If you develop programs for any Apple computer, you should know about these manuals.

The *Human Interface Guidelines* manual describes Apple's standards for the desktop interface of any program that runs on Apple computers. If you are writing a commercial application for the Apple IIGS, you should be familiar with the contents of this manual.

The *Apple Numerics Manual* is the reference for the Standard Apple Numeric Environment (SANE™), a full implementation of the *IEEE Standard for Binary Floating-Point Arithmetic* (IEEE Std 754-1985). The functions of the Apple IIGS SANE tool set match those of the Macintosh™ SANE package and of the 6502 assembly-language SANE software. If your application requires accurate or robust arithmetic, you'll probably want to use the SANE routines in the Apple IIGS. The *Apple IIGS Toolbox Reference* tells how to use the SANE routines in your programs. The *Apple Numerics Manual* is the comprehensive reference for the SANE numerics routines.

How to use this manual

The first chapter of this manual introduces and defines the toolbox, gives the names of the various managers and tool sets, and provides a brief description of the kind of routines included under each manager and tool set. This chapter gives you an overview of what you can expect to find in the toolbox.

Chapter 2 gives you general instructions on how to load and start up tool sets. The chapter also discusses how you set up your program to make a tool call from 65816 assembly language and from C.

Important
If you're still completely lost after the first two chapters and don't know where to begin, you probably need more information than this reference manual can provide. You should read the *Programmer's Introduction to the Apple IIGS* before continuing with this manual.

After the introductory chapters, the book presents the first tool set chapter. The tool sets are presented in alphabetical order by tool set or manager name; thus, the Apple Desktop Bus is described first, and the Window Manager is described in the last chapter of Volume 2.

Each tool set chapter introduces the tool set and then previews the routines. The preview is a good place to look if you know the type of routine you want but don't know its name. In addition, the preview is a good way for you to quickly scan the specific capabilities of the tool set.

After the preview, one or more sections discuss the general concepts of the tool set. One of the sections is usually entitled "Using the *XXXX* Tool Set." This section tells you about the general flow of the commands in the tool set and also provides a table indicating what other tool sets the tool set depends upon.

Finally, each tool set chapter presents the specifications for all of the tool set's routines. The standard housekeeping routines are described at the beginning of the routines. The rest of the routines are presented in alphabetical order so that you can find them easily; exceptions to the alphabetical order are Chapter 14, "Miscellaneous Tool Set," and Chapter 21, "Sound Tool Set."

Each individual routine begins with the routine name and number. Following a short description are stack and parameter diagrams (if applicable), which show how the stack should look for assembly-language programmers and define the length and function of each parameter. The length is defined as shown in Table P-2.

Table P-2
Parameter length
on the Apple IIGS

Term	Length
Byte	8 bits
Word	2 bytes; 16 bits
Long	4 bytes; 32 bits

The function of each parameter remains constant, regardless of what language you're using to access the capabilities of the tools. To make this manual as useful and non-language-specific as possible, many of the parameters are also identified by a pseudo-type; that is, the type listed may or may not exist as a formal definition in a programming language, but the **pseudo-type** at least provides some additional information about the nature of the parameter. The pseudo-types used are defined in Table P-3.

❖ *Note:* In fact, some of the pseudo-types may actually conflict with the type definitions in some programming languages; for example, Pascal does not define BOOLEAN in the same way as the BOOLEAN pseudo-type does.

Table P-3
Stack diagram pseudo-types

Type	Length	Definition
POINTER	Long	Points to an address (see Chapter 12, "Memory Manager")
HANDLE	Long	Points to a pointer (see Chapter 12, "Memory Manager")
BOOLEAN	Word	TRUE is nonzero, FALSE is 0
RECT	Four words	Data structure specifying coordinates of a rectangle as top, left, bottom, right (see Chapter 16, "QuickDraw II," in Volume 2)
POINT	Two words	Y and X coordinates of a point (see Chapter 16, "QuickDraw II," in Volume 2)
INTEGER	Word	16-bit signed or unsigned value
LONGINT	Long	32-bit signed or unsigned value
FIXED	Long	32-bit signed value with 16 bits of fraction
FRAC	Long	32-bit signed values with 30 bits of fraction
EXTENDED	N/A	80-bit signed floating-point values with 64 bits of fraction

After the stack diagrams, the possible tool set errors that can occur during the routine are listed. Also listed separately is the C synopsis of the routine, with the parameters and the appropriate types. Finally, any additional information for an individual routine, such as an example, figure, or table, is provided after the C synopsis.

At the end of each tool set chapter is a summary of the tool set's constants, data structures, and error codes.

A comprehensive index for both volumes appears at the end of Volume 2.

Other materials you'll need

To use the routines described in this manual, you will need an Apple IIGS with at least one external disk drive. The toolbox routines require only the base computer, although the Apple IIGS Programmer's Workshop and many application programs may require more memory.

You will also need an Apple IIGS system disk. A system disk contains the RAM-based tool sets, ProDOS 16, ProDOS 8, the System Loader, and other system software necessary for proper functioning of the computer. A system disk may also contain application programs.

Notations and conventions

To help make this manual more understandable, the following conventions and definitions have been used throughout.

Typographic conventions

Each new term introduced in this manual is printed first in **boldface.** This lets you know that the term has not been defined earlier and also indicates that an entry for it appears in the glossary, which is in Volume 2. Parameter, field, and bit names are given in *italics,* indicating that the name is replaced by a value when used in actual code. Constant names are given in `Courier` typeface.

Watch for these

The following conventions mark special messages:

❖ *Note:* Text set off in this manner presents sidelights or interesting points of information.

Important

Text set off in this manner presents important information or instructions.

Warning

Text set off in this manner indicates that the system will fail if the instructions are not followed.

Future toolbox enhancements

Apple is continually improving the performance and capabilities of the Apple IIGS Toolbox. Many of the performance improvements will not affect this manual; inevitably, however, enhancements will be added that this manual does not document. To be certain that you have the latest information about the toolbox, you can contact the Apple Programmer's and Developer's Association. APDA is administered by the A.P.P.L.E. cooperative in Renton, Washington.

Apple Programmer's and Developer's Association
290 SW 43 Street
Renton, WA 98055
(206) 251–6548

Chapter 1

Introducing the Apple IIGS Toolbox

What is a tool set?

A software **tool set**, in the Apple IIGS environment, is a collection of related routines (or functions) that provides one major capability. Each routine performs a fundamental operation and converts zero or more inputs to zero or more outputs and side effects. For example, QuickDraw II provides routines that handle graphics on the Apple IIGS. SetPenSize and SetPenMode, for example, are routines within that tool set that set the pen size and pen mode.

The tool sets, then, are routines that are always available to perform many common tasks. Many of the capabilities of the Apple IIGS are easily accessed through the tool sets. For example, even the Memory and Event Managers are considered to be tool sets on the Apple IIGS. (*Manager,* by the way, is another name for a collection of routines. Some of the tool sets are called *XXX Tool Set;* others are called *XYZ Manager.*)

❖ *Macintosh programmers:* You'll be familiar with this concept from your work with the Macintosh Toolbox.

What can the tool sets do for you?

The tool sets provide capabilities that allow your application to concentrate on its specific business rather than on background tasks.

A number of the tools are included in ROM. This approach makes those tools available to all programs without using disk space or memory. Additional tools are available in RAM. However, you don't need to keep track of where a particular function is or even whether it is in ROM or RAM. A tool set called the Tool Locator, which allows tools and applications to communicate, takes care of the necessary bookkeeping functions.

The Tool Locator is automatically initialized when ProDOS 16 is booted; thereafter, the Tool Locator does its work behind the scenes. To use the Apple IIGS Toolbox in the simplest fashion, you'll need to load and start the appropriate tool sets; after that you don't need to know anything but the name of the routine and how to call it from the appropriate programming language. (Calling information is in Chapter 2, "Using the Apple IIGS Tool Sets.")

The tool sets thus provide their capabilities at a minimum cost; their bookkeeping functions are almost automatic, the interface to them is simple, and the applications you write will not be rendered obsolete by any future changes in the hardware.

The Tool Locator is also flexible enough to allow you to extend the scope of the tool sets by writing your own, and it is powerful enough to keep track of both the Apple tools and your tools. You can write and install your own tool sets if you wish and still have the Apple tool sets available when you need them.

Are there any limitations?

There is at least one important point to consider when you are planning to call an Apple IIGS tool from your application: The tools are designed to run in **full native mode** rather than in Apple II emulation mode. In full native mode, the e, m, and x registers are all set to 0, which provides a 16-bit accumulator and 16-bit index registers. Almost all of the tools require this mode and simply will not work if the machine is in any other state. The limited exceptions to this rule are documented under the individual calls described in later chapters of this manual.

What kinds of tool sets are provided?

In this section, we simply list the categories of tool sets and the groups of routines within each tool set. The listing does not contain definitions of the individual routines for each tool set; for that summary, look at the first few pages of the chapter describing the appropriate tool set. To find an individual routine, look it up by its name in the index or look under the appropriate tool set.

A summary such as this can seem like teasing; in fact, that's part of its purpose. At this point, we wish to introduce you to the entire range of the Apple IIGS Toolbox routines and encourage you to use as many of the tool sets as possible.

The big five

Five tool sets provide the basic framework upon which the other tool sets build. All of these tools must be used in every event-driven application. The tools in this group are as follows:

Tool Locator: Provides the mechanism for dispatching tool calls. This tool allows you to get away with not knowing where in memory the tool sets reside; the Tool Locator knows and retrieves them when you make a tool call.

Memory Manager: Allocates all memory available to the application. When your application needs memory, it must request it from the Memory Manager.

Miscellaneous Tool Set: Includes mostly system-level routines that must be available for other tool sets.

QuickDraw II: Controls the graphics environment and draws simple objects and text. Other tools call QuickDraw II to draw such elements as windows.

Event Manager: Traps events as they happen, maintains a queue of the events, and passes the events on to the application.

Desktop interface tool sets

The interface tool sets control the desktop interface. You will almost always use the Window and Menu Managers and the Line Edit Tool Set to adhere to the *Human Interface Guidelines: The Apple Desktop Interface.* You should use the other tool sets if your application needs their features (for example, you will need the Dialog Manager if your application uses dialog boxes). Many of the interface tool sets are also needed to support desk accessories. The tool sets are as follows:

Window Manager: Updates and maintains windows.

Control Manager: Presents **controls,** which are objects on the screen that the user can manipulate with the mouse to cause instant action or change settings.

Menu Manager: Controls and maintains the pull-down menus and the items in the menus.

LineEdit Tool Set: Presents text on the screen and allows the user to edit that text.

Dialog Manager: Implements **dialog boxes,** which appear on the screen when an application needs more information to carry out a command.

Scrap Manager: Supports the **desk scrap,** which allows data to be copied from one application to another (or from one place to another within an application).

Desk Manager: Enables applications to support desk accessories, which are mini-applications that can be run at the same time as another application.

Standard File Operations Tool Set: Presents the standard user interface when a file is to be saved or opened.

List Manager: Presents the user with a list from which to choose (for example, the Font Manager uses the List Manager to arrange the list of fonts).

Font Manager: Provides information to applications as to how many fonts are available and what the characteristics of those fonts are.

QuickDraw II Auxiliary: Adds some capabilities to QuickDraw II, particularly the ability to collect drawing calls into a picture.

Math tool sets

Integer Math Tool Set: Supports mathematics routine with integers, long integers, and signed fractional numbers. Also converts integers and hex and decimal numbers from one form to another.

SANE Tool Set: Supports the Standard Apple Numerics package, which provides IEEE standard extended-precision calculations.

Printer tool set

Print Manager: Allows your application to use standard QuickDraw II routines to print text or graphics on a printer.

Sound tool sets

Sound Tool Set: Supports the sound tool set interfaces and provides the basic sound capabilities.

Note Synthesizer: Not described in this manual.

Note Sequencer: Not described in this manual.

Specialized tool sets

Apple Desktop Bus Tool Set: Controls Apple Desktop Bus activity.

Scheduler: Prevents a tool call from crashing the system by asking for a temporarily unavailable system resource.

Text Tool Set: Provides an interface between Apple II character device drivers and applications running in native mode.

Groups of routines within each tool set

This section describes the major functional groups of the routines within each tool set. The tool sets are organized in the order in which they appear in later chapters in the two volumes of this manual.

Every tool set or manager includes a class of routines known as housekeeping routines. Included among the routines are boot initialization and application startup calls, an application shutdown call, a reset call, a call that returns the version number of the particular tool set, and a call that returns the status of the tool set.

Apple Desktop Bus Tool Set

These routines receive and send data to the ADB microcontroller.

Control Manager

Initialization and termination routines: Create, show, hide, draw, highlight, and dispose of controls.

Mouse location routines: Provide the ability to find out whether the user pressed the mouse button while the cursor was in a control.

Control movement routines: Move a control or allow the user to drag the control around the screen.

Control record access routines: Allow your application access to some fields of the control record.

Miscellaneous routines: Add some capabilities, such as returning the height and width of the size box control.

Desk Manager

Installation routines: Install new desk accessories (NDAs) or classic desk accessories (CDAs) into the system.

Classic desk accessory routines: Activate the Desk Manager and display the CDA menu, change the names of the built-in CDAs, and return the pointer to the names of the built-in CDAs.

New desk accessory routines: Open and close NDAs, add the names of the NDAs to a specified menu, return the total number of NDAs currently installed, and handle events relating to desk accessories.

State-saving routines: Save and restore areas used by the Desk Manager; these routines must not be called by an application.

Dialog Manager

Dialog creation and disposal routines: Create and close modal or modeless dialog boxes.

Item creation and removal routines: Add and remove items from dialog boxes.

Dialog event-handling routines: Handle events in dialog boxes, including the standard Cut, Copy, Paste, and Delete commands.

Alert routines: Invoke alerts, including those containing predefined icons.

Item manipulation routines: Allow you to

- Change or return text for items in dialog boxes
- Set or return the item type, display rectangle, or current value of an item
- Return the ID of the first item, or the next item after a specified item
- Set or return the default button item
- Return or reset the alert stage of an alert
- Hide, show, or find a dialog item
- Call the standard dialog filter
- Enable or disable a dialog item
- Redraw a part of a dialog in a specified update region

Miscellaneous dialog routines: Establish the sound procedure for alerts and specify the font for the dialog and alert window.

Event Manager

Event accessing routines: Check events to see if they are of interest to the application and returns them if appropriate.

Mouse status routines: Provide the ability to read the location of the mouse and the status of the mouse buttons.

Event queue routines: Allow you to place or remove events into the event queue.

Miscellaneous Event Manager routines: Allow you to

- Check the number of ticks (sixtieths of a second) since the system was last started
- Return the suggested maximum difference between ticks that determines a double mouse-click
- Return the number of ticks between blinks of the caret marking the insertion point
- Specify the system event mask
- Allow use of an alternative pointing device, such as a graphics tablet, in place of or in conjunction with the mouse

- Return the address of the direct page used by the Event Manager
- Generate a switch event

Font Manager

Font family routines: Allow you to

- Return the number of font families available
- Return the number and name of a particular font family
- Return the family name and characteristics of a family specified by number
- Return the family number corresponding to a font family name
- Add a family number and name to the Font Manager's list of known font families

Font routines: Allow you to

- Find a specified font, or the *best-fit* available font if the specified font isn't available, and load the font if necessary
- Make a specified font in memory purgeable or unpurgeable
- Return the number of fonts currently available to the Font Manager that fit a specified description
- Return the font ID and the characteristics of a particular font
- Load a specified font into memory (if it is not already there) and make it current and unpurgeable
- Make a specified font the current or system font
- Return the font ID of the current or system font
- Add a variation of a preexisting font family to the Font Manager's collection of available fonts

Menu and dialog routines: Append the names of available font families to a specified menu, display a dialog box enabling the user to select a new font, size, and style, or translate from a menu item ID into a font family number or vice versa.

Integer Math Tool Set

Math routines: Allow you to

- Support multiplication and division of Integer, Longint, Fixed, and Frac numbers
- Take two signed Integers and produce a Fixed number as a ratio of the numerator and denominator
- Take a Frac input and return a rounded Frac square root
- Take a Fixed input (radians) and return its Frac cosine or its Frac sine
- Take two like inputs and return a Fixed arc tangent (in radians) of their coordinates
- Return the high- or low-order word of a long input
- Convert from one type of value to another

Integer Math string routines: Convert between a binary value and an ASCII character string representing that value, allowing you to

- Convert integers to hex, Long, or decimal format ASCII strings
- Convert Longs to decimal format ASCII strings
- Convert hex ASCII strings to Integer or Long
- Convert decimal format ASCII strings to Integers or Longs

LineEdit Tool Set

Edit record routines: Initialize an edit record, dispose of an edit record, copy text into an edit record, return a handle to the text of a specified edit record, or return the length of the text of an edit record.

Insertion point and selection range routines: Cause the caret at the insertion point to blink, control the selection range, and highlight or unhighlight the selection range or caret.

Editing routines: Replace, cut, copy, paste, delete, or insert the selection range or at the caret of an edit record, as appropriate.

Text display routines: Draw the text of an edit record, including routines that use justification, word wrap, and embedded changes.

Scrap handling routines: Copy to and from the LineEdit scrap, return a handle to the scrap, and set or return the size of the LineEdit scrap.

Miscellaneous LineEdit routines: Set the *leHiliteHook* and *leCaretHook* field of a specified edit record.

List Manager

List routines: Create a list control, reset a list control, alphabetize a specified list, and return a pointer to the list control's definition procedure.

Member routines: Draw one or all members of a list, search a list for the next selected member, and select a member.

Memory Manager

Memory allocation routines: Create new blocks, reallocate purged blocks, dispose of old blocks, and purge unlocked, purgeable blocks.

Block information and free space routines: Return the handle of a block, check a handle to see whether it is valid, set or return the size of a block, compact memory space, return the total number of free bytes or the size of the largest free block, and return the total amount of memory.

Locking and purge level routines: Lock, unlock, and set the purge levels of memory blocks.

Other Memory Manager routines: Copy a specified number of bytes from a source to a destination.

Menu Manager

Initialization and termination routines: Create a default menu bar, allocate space for a menu list and its items, free memory, compute standard sizes for the menu bar and menus, and set menu dimensions.

User interaction routines: Draw highlighted titles, pull down menus, handle user interaction when a mouse button is clicked on a menu bar, map a character to the associated menu and item for that character, and attempt to refresh the screen.

Menu drawing routines: Draw the current menu bar, highlight or unhighlight the title of a menu, and flash the current menu bar.

Menu and item shuffling routines: Insert a menu into the menu list after a specified menu item or at the front of the list, remove a menu from the menu list, insert a menu item into a menu after a specified menu item or at the front of the menu list, and remove an item from the current menu.

Menu bar access routines: Allow you to

- Set a new system bar or return the handle of the current system menu bar
- Return the handle of the current menu bar
- Set or return the normal, inverse, and outline colors of the current menu bar
- Set or return the starting position for the leftmost title within the current menu bar
- Return the number of items in a specified menu

Menu record access routines: Allow you to

- Return a handle to a menu record
- Set or return the width of a menu title
- Set the state or return the current state of a menu
- Set or return the title for a menu
- Specify a new menu number

Item record access routines: Allow you to

- Set or return the name of a menu item
- Enable or disable a menu item
- Check or uncheck a menu item; that is, display or not display a check mark to the left of the item
- Set or return the character to be displayed or not displayed to the left of the item
- Set or return the text style for a menu item
- Set or return the values for a menu item, such as whether it is underlined and highlighted
- Specify the ID number of a menu item
- Determine how many times all menu items should blink when selected

Miscellaneous Menu Manager routines: Return a pointer to the Menu Manager port, adjust screen resolution and redraw the current menu bar, and reinitialize the palettes needed for the color Apple logo.

Miscellaneous Tool Set

The miscellaneous tools are a collection of various routines. Their capabilities are summarized below.

Battery RAM routines: Write or read data to and from the Battery RAM and write and read data to and from a specified Battery RAM parameter.

Clock routines: Set or return the current time in various ways.

Firmware entry routine: Allow you to use some Apple II emulation-mode entry points.

Get system address routine: Retrieves the address of some important system parameters referenced by the firmware.

Tick counter routine: Retrieves the current value of the tick counter.

Interrupt control routines: Enable or disable certain interrupt sources and return the status of the interrupts.

Mouse and absolute clamp routines: Allow you to

- Initialize, set, position, home, and read the values for the mouse
- Set and get the clamp values for the mouse
- Return the interrupt status for the mouse
- Set and get the clamp values for an absolute device

Packing and munging routines: Pack bytes into, and unpack bytes from, a special format that uses less storage space and manipulate bytes in a string of bytes.

Heartbeat routines: Install or delete a specified task in the Heartbeat Interrupt Service queue and remove all tasks from the Heartbeat Interrupt Service queue.

System bell routine: Calls the Apple II monitor entry point BELL1.

System Failure Manager: Calls a system failure handler, allowing you to provide your own failure message or code, and halts program execution.

User ID Manager routines: Create and delete ID tags used for memory management and return the status of an ID tag.

Vector initialization routines: Set or return the vector address for a specified interrupt manager or handler.

Print Manager

Print record and dialog routines: Fill a print record with default values; check the contents of a print record for compatibility; and conduct a style, job, or Choose Printer dialog with the user.

Printing routines: Initialize a GrafPort for use in printing, close a printing GrafPort, begin a new page, end the printing of the current page, print a spooled document, and print all or part of a specified pixel map.

Error handling routines: Return the last printer error code left during the printing loop by Print Manager routines and set the printer error code.

Printer driver and port driver routines: Return the version number of the currently installed printer driver and the currently installed port driver.

QuickDraw II

Global environment routines: Set up the graphics environment for QuickDraw II and the other tools, allowing you to

- Specify (*set*) or return (*get*) the settings for the scan-line bytes, the color tables, and the color-table entries
- Set or get the system font
- Set or get the maximum width or size of a pixel map that is being drawn
- Clear the screen
- Turn Super Hi-Res graphics mode on or off
- Set or get the maximum width or size of a pixel map that is being drawn
- Set the size of the clipping and text buffers, either with or without padding
- Save and restore the buffer sizing information

GrafPort routines: Set up the GrafPort for QuickDraw II and the other tool sets, allowing you to

- Open, initialize, and close a GrafPort
- Set or get the current Grafport or the GrafPort's location information
- Set, get, or change the size of, adjust the origin of, or move the drawing location
- Set, get, or change the current clip regions

Pen and pattern routines: Control pen and pattern information, allowing you to

- Hide or show the pen and set or get the current values for the pen state, size, mode, pattern, and mask
- Set or get the background pattern
- Move the current pen location to a point or a relative distance

Font routines: Control the pen and pattern information, allowing you to

- Set or get the current font, font ID, font flags, and font globals
- Set or get the text face and mode
- Set or get the *spExtra* and *chExtra* fields
- Set or get the foreground and background colors

Miscellaneous GrafPort routines: Allow you to

- Set or get the clip region, visible region, and handle to the visible region
- Set or get various fields in the GrafPort record, such as *picSave, rgnSave, polySave, userField,* and *sysField*
- Set or get the pointer to the *grafProcs* record

Line drawing routines: Draw a line from the current pen position to either a specified point or a relative distance.

Rectangle, region, polygon, oval, round rectangle, and arc drawing routines: Allow you to do the following with their respective objects:

- Frame; that is, draw the boundary of the object using the current pattern and pen size
- Paint; that is, paint the interior of the object with the current pen mode and pen pattern
- Erase; that is, fill the interior of the object with the background pattern
- Invert; that is, invert the pixels in the interior of the object
- Fill; that is, fill the interior of the object with a specified pen pattern

Pixel transfer routines: Allow you to scroll or shift a region of pixels or to transfer pixels.

Text drawing and measuring routines: Allow you to

- Draw a character, text, Pascal string, or C string
- Get the width of a character, text, Pascal string, or C string
- Get a rectangle that describes the area overwritten by a character, text, Pascal string, or C string

Calculations with rectangles: Allow you to

- Set the dimensions of a rectangle
- Offset or inset a rectangle
- Calculate the intersection of two rectangles and place the intersection in a third
- Calculate the union of two rectangles and place the union in a third
- Determine whether a point is in a particular rectangle or copy points to the upper left and lower right of a rectangle
- Determine whether two rectangles are equal
- Determine whether a rectangle is empty

Calculations with points: Allow you to

- Add two points together or subtract a point from another point
- Set a point to specified values
- Determine whether two points are equal
- Convert a point from local to global coordinates, and vice versa

Calculations with regions: Allow you to

- Create a new region or dispose of a region
- Copy contents from one region to another
- Set a region to a rectangle or make it empty
- Open or close a temporary region
- Offset or inset a region
- Calculate the union of two regions and place the union in a third
- Calculate the intersection of two regions and place the intersection in a third
- Calculate the difference between two regions and place the difference in a third
- Calculate the difference between the union and the intersection of two regions and place the result in a third
- Determine whether a point is in a particular region
- Determine whether a rectangle intersects a particular region
- Determine whether two regions are equal
- Determine whether a region is empty

Calculations with polygons: Allow you to open, close, dispose of, or offset a polygon.

Mapping and scaling utilities: Allow you to

- Map points, rectangles, and regions from a source to a destination
- Scale points from a source to a destination

Cursor-handling routines: Allow you to

- Set or get the current settings for the cursor
- Show or hide the cursor
- Obscure the cursor
- Reinitialize the cursor

Miscellaneous QuickDraw II utilities: Allow you to

- Return a pseudo-random number
- Set a seed value for a random number generator
- Get the values for a specified pixel
- Return a pointer to a specified ROM table
- Indicate whether the cursor drawing code should use scan line interrupts
- Set up a specified record of pointers

QuickDraw II Auxiliary

Picture routines: Allocate memory for recording of a picture definition, insert comments into the current open picture, draw pictures, close pictures, and remove pictures.

Miscellaneous QuickDraw II Auxiliary routines: Copy a pixel image from one place to another and change the cursor to a watch cursor.

SANE Tool Set

The SANE Tool Set routines provide an entry to the SANE functions, which contain

- Numeric scanners and formatter
- Elementary functions, financial functions, and random number generator
- Basic arithmetic operations, comparsions, conversions, environmental control, and IEEE auxiliary operations

Scheduler

This tool set contains a routine that allows your own tool set or interrupt handler to add a task to the Scheduler's queue; this prevents tasks from calling a currently busy system resource.

Scrap Manager

This tool set contains routines that allow you to

- Write or read the desk scrap
- Clear the contents of the scrap and increment the scrap count
- Append data to data in the appropriate scrap
- Copy scrap information of the appropriate type to a specified handle
- Return the current scrap count
- Return a flag indicating the current state of the scrap
- Return a copy of the handle for the scrap of a specified type
- Return the size of the specified scrap
- Set or return a pointer to the pathname used for the Clipboard file

Sound Tool Set

Sound Tool Set routines: Allow you to

- Write and read a specified number of bytes to and from DOC RAM
- Set and get the volume for a sound generator or change the system volume
- Start or stop the sound for a particular generator
- Return the status of a specifed generator or the status of all generators
- Set up the entry points for the system and the user sound interrupt handler
- Return the current Free-Form Synthesizer sound-playing status
- Return the jump table address for the low-level routines

Low-level sound routines: These routines, designed for quick access, allow you to

- Write and read any register within the DOC
- Write and read a specified Ensoniq RAM location
- Write and read the next DOC or RAM location

Standard File Operations Tool Set

This tool set contains routines that allow you to display a standard or custom Open File or Save File dialog box and a routine to determine whether file names will be displayed in all uppercase letters or with uppercase and lowercase letters.

Text Tool Set

Text global routines: Allow you to set and return the global parameters for the input, output, and error output devices.

I/O directing routines: Allow you to set the type and location, and return the type, of the input, output, and error output devices.

Text routines: Allow you to

- Initialize a text device
- Pass a control code to a text device
- Execute a status call to a text device
- Combine a specified character, Pascal-type string, C-type string, line, or block of characters with the output global AND mask and OR mask and write the character to the output text device or error output text device
- Read a character obtained from the input text device, combine it with the input global masks, and return the character on the stack
- Read a character string or block of characters from the input text device, combine it with the input global masks, and write it to a specified memory location

Tool Locator

This tool set contains routines that allow you to

- Ensure that one or more system tool sets are available and have specified minimum version numbers
- Unload a specified tool set from memory
- Return an entry in the function pointer table for a specified function in a specified tool set
- Return the pointer to the function pointer table of a specified tool set
- Install the pointer to a function pointer table in the appropriate tool pointer table
- Set or return the pointer to the work area for a specified tool set
- Display a simulated dialog box on the Super Hi-Res display or on the 40-column text screen
- Save and restore the state of the text screen
- Allow applications to communicate with each other

Window Manager

Initialization and termination routines: Allow you to

- Control the addition of regions to and the subtraction of regions from the desktop and control the current desktop pattern
- Create a window
- Remove a window
- Close the Window Manager's GrafPort and open a new GrafPort in the other Super Hi-Res resolution

Window record and global access routines: Allow you to

- Return a pointer to the Window Manager port
- Set the icon font for the Window Manager
- Set or get a value that is inside the window record and is reserved for the application's use
- Set or get the title of a window
- Set or get the color of a window's frame
- Return a pointer to the the active window
- Return a pointer to the next window in the window list after a specified window
- Determine whether a specified window is a system or an application window
- Set or get the bit flag that describes a specified window's frame type
- Return a handle to a specified window's structure, content, or update region
- Set or get a pointer to the routine that is called to draw, hit test, and otherwise define a window's frame and behavior
- Return the handle to the first control in the window's control list
- Set or get the rectangle to be used as the zoomed or unzoomed size for the contents of a specified window
- Determine whether a specified window is a system window
- Mark a specified window as a system window
- Set or get the origin of the window's port
- Specify the mask used to put the horizontal origin on a grid
- Make a specified window the current port and set its origin
- Set or get the height and width of the data area of a window
- Set or get the maximum values to which a window's content region can grow
- Set or get the number of pixels by which TaskMaster will scroll the content region when the user selects the arrows on window frame scroll bars
- Set or get the number of pixels by which TaskMaster will scroll the content region when the user selects the page regions on window frame scroll bars
- Set or get the pointer to the routine that draws the content region of a window

Information bar routines: Allow you to

- Set or get a pointer to a specified window's draw information bar procedure
- Set or get the value associated with the draw information bar routine for a window
- Set the information rectangle to the coordinates of the information bar rectangle
- Draw or hit test outside your application's information bar definition procedure
- Put the Window Manager back into a global coordinate system

Window shuffling routines: Allow you to

- Make a specified window the active window
- Make a specified window invisible
- Make a specified window visible if it was invisible and then draw the window
- Show or hide a window
- Bring a specified window to the front of all other windows and redraw the windows as necessary
- Send a specified window behind another specified window, redrawing any exposed windows

Window drawing routines: Highlight or unhighlight a specified window and redraw the entire desktop and all the windows.

User interaction routines: Allow you to

- Determine which part of which window, if any, the cursor was in when the user pressed the mouse button
- Pull around a dotted outline of a specified window, following the movements of the mouse until the mouse button is released
- Pull around a size image of a specified window, following the movements of the mouse until the mouse button is released
- Track the mouse until the mouse button is released, highlighting the close region so long as the mouse location remains inside it and unhighlighting it when the mouse moves outside it
- Track the mouse until the mouse button is released, highlighting the zoom region so long as the mouse location remains inside it and unhighlighting it when the mouse moves outside it
- Use TaskMaster to call GetNextEvent and handle the event if possible

Window sizing and positioning routines: Move a window, enlarge or shrink a window, and switch the size and position of a window between its current size and position and its maximum size.

Update region routines: Allow you to

- Accumulate rectangles and regions into the update region of the window whose GrafPort is the current port
- Remove rectangles and regions from the update region of the window whose GrafPort is the current port
- Replace the visible region of the window's GrafPort with the intersection of the visible region and the update region and then set the window's update region to an empty region
- Restore the normal visible region of a specified window's GrafPort that was changed by BeginUpdate

Miscellaneous Window Manager routines: Pin a point inside a rectangle and look for a visible window that needs updating.

Chapter 2

Using the Apple IIGS Tool Sets

Starting up the required tool sets

Three tool sets are required for any application using the Apple IIGS tool sets: the Tool Locator, the Memory Manager, and the Miscellaneous Tool Set.

As the name implies, the Tool Locator is the tool that finds all other Apple IIGS tool sets. Thus, the application must start up the Tool Locator before it calls any other tool sets. The call to start the Tool Locator is

TLStartup

without any parameters. Once the Tool Locator is running, you can begin starting up the other tools you need.

Next, you need to start up the Memory Manager, which performs the housekeeping task of assigning memory. Because all of the other tools ask the Memory Manager for any memory space they need, the Memory Manager must be started up before any other tool set except the Tool Locator. The call to start the Memory Manager is

MMStartup

without any input parameters. The call returns a user ID for this execution of the application, which other managers and tool sets need to reference in order to get memory space. You need to store this ID for later use.

Next, you must start up the Miscellaneous Tool Set. Don't be misled by the name; this set of routines is crucial to the success of an event-driven application, since other tool sets need to use various calls in the Miscellaneous Tool Set. The call to start up the Miscellaneous Tool Set is

MTStartup

The next step is to start up QuickDraw II, which is the tool set responsible for manipulating graphics on the Apple IIGS. Many of the other tool sets call QuickDraw II to draw their graphics, particularly the tool sets controlling the desktop interface. Therefore, QuickDraw II must be started up before those other tools.

Important

QuickDraw II and several other tool sets require some direct-page space. Because the Memory Manager has already been started up, you can obtain the direct-page space from it. You then provide the appropriate address in the startup call for the appropriate tool set (and can thus find the total direct-page space needed for the tool sets you're using by totaling the pages needed in the startup calls), such as in the QDStartUp routine that follows.

The starting address for a page of direct-page space must be page aligned (that is, must be a multiple of $100). If you need more information about assigning direct-page space, see the *Programmer's Introduction to the Apple IIGS.*

To start QuickDraw II, you call the QDStartup routine and provide the following:

- The starting location for QuickDraw II's direct-page space (QuickDraw II needs three consecutive pages)
- The Master Scan Line Control Byte (Master SCB), which controls the basic properties of the lines that will appear on the screen, such as the resolution and color tables
- The maximum width, in bytes, of the largest pixel image that will be drawn (0 equals the entire screen)
- The user ID of the program requesting the space (the user ID was provided by the Memory Manager)

Now start up the Event Manager, which provides the basic support for event-driven applications by monitoring the following:

- The user's actions, such as those involving the mouse and keyboard
- The actions taken by other managers, such as the Window and Control Managers

To start the Event Manager, call the EMStartup routine and provide the following:

- The starting location for the Event Manager's direct page (the Event Manager needs one page of direct-page space)
- The maximum number of events that the event queue can hold
- The borders for the mouse or cursor, called the **clamp values**
- The user ID of the application (the user ID was provided by the Memory Manager)

The basic structure of the tools is now in place; in fact, the information up to this point is so generic that you may wish to place it in a single module. You can then either use that module for all of your applications or copy and modify it slightly when necessary.

Loading and starting up other tool sets

When the required tool sets are in place, you can then load all other tool sets your application will use. To simplify things, and to ensure that the correct version of each tool set is available, you will usually want to load all remaining tool sets you want for your application at this time. Loading all tool sets also saves you the trouble of determining which tool sets are in ROM and which are in RAM.

You load the tools by using the Tool Locator LoadTools routine, which needs as input a pointer to a tool table that you provide. That table must contain the following information:

- The total number of tool sets to be loaded with this call
- The tool set number of each tool set to be loaded, followed by the minimum acceptable version number of each tool set

The format of the information in the tool table is shown in the section "LoadTools" in Chapter 24, "Tool Locator." The tool set numbers are listed in Table 2-1 and also in Table 24-2 in the section "LoadTools."

Table 2-1
Tool set numbers

Tool set number		Tool set name	Tool set number		Tool set name
$01	#01	Tool Locator	$0F	#15	Menu Manager
$02	#02	Memory Manager	$10	#16	Control Manager
$03	#03	Miscellaneous Tool Set	$11	#17	Loader
$04	#04	QuickDraw II	$12	#18	QuickDraw II Auxiliary
$05	#05	Desk Manager	$13	#19	Print Manager
$06	#06	Event Manager	$14	#20	LineEdit Tool Set
$07	#07	Scheduler	$15	#21	Dialog Manager
$08	#08	Sound Tool Set	$16	#22	Scrap Manager
$09	#09	Apple Desktop Bus Tool Set	$17	#23	Standard File Operations Tool Set
$0A	#10	SANE Tool Set	$18	#24	Disk Utilities
$0B	#11	Integer Math Tool Set	$19	#25	Note Synthesizer
$0C	#12	Text Tool Set	$1A	#26	Note Sequencer
$0D	#13	Reserved for Apple use	$1B	#27	Font Manager
$0E	#14	Window Manager	$1C	#28	List Manager

Important

Any RAM-based tools you wish to use must be located in the TOOLS subdirectory of the SYSTEM directory.

After you have loaded the tools, you must then start up each tool set. Because each tool set depends upon the presence of other tool sets, certain tool sets must be started up for others to work. In addition, these tool sets must be started up in a prescribed order. This order is shown in Table 2-2, with tool sets lower on the list depending upon the presence of all the tool sets higher on the list. Thus, the Menu Manager depends upon the presence of all of the tool sets up to the Control Manager; those tool sets must all be started up before the Menu Manager.

When you shut down the tools before you quit your application, you must shut them down in the reverse order from which they were started up; that is, the last one started up must be shut down first, the next-to-the-last one started up must be shut down next, and so on.

Table 2-2
Tool set startup order

Tool set number		Tool set name
$01	#01	Tool Locator
$02	#02	Memory Manager
$03	#03	Miscellaneous Tool Set
$04	#04	QuickDraw II
$06	#06	Event Manager
$0E	#14	Window Manager
$10	#16	Control Manager
$0F	#15	Menu Manager
$14	#20	LineEdit Tool Set
$15	#21	Dialog Manager
$05	#05	Desk Manager
$17	#23	Standard File Operations Tool Set
$16	#22	Scrap Manager
$1C	#28	List Manager
$13	#19	Print Manager
$1B	#27	Font Manager

In addition, if you are using QuickDraw II Auxiliary, it must be started up after QuickDraw II. You may assume that any other tool sets do not need to be started up in any particular order; that is, they may be started up or shut down at any time.

Tool sets require the presence of certain minimum versions of other tool sets. Those versions are given in the section "Using the XXX Tool Set" in each chapter. The versions are also summarized, along with the tool set startup order, in Appendix C, "Tool Set Dependencies and Startup Order," in Volume 2.

Calling the correct routine

The toolbox routines are available at the time of publication from 65C816 assembly language and Apple IIGS Workshop C. The general rules for accessing the tool sets are outlined in the following sections.

Calling a routine from assembly language

The interface files that allow the tool sets to be accessed from assembly language are included in the Apple IIGS Programmer's Workshop. They consist of equate files that provide symbolic constant and data structure field offsets and macro files that allow each tool call to be invoked by name. The steps to use a particular routine are as follows:

1. Make the macro accessible by using the MCOPY assembler directive for the appropriate file (for example, `MCOPY M16.Quickdraw` for the QuickDraw II routines). You may wish to use the APW MacGen utility to create a single macro file containing only the macros that you're using (see the *Apple IIGS Programmer's Workshop Reference* for more about MacGen).
2. If the routine has any output, push room for it onto the stack.
3. If the routine has any inputs, push them onto the stack in the specified order. The input parameters are described under each individual call in the two volumes of this manual.
4. Invoke the appropriate macro by entering the name of the routine preceded by an underscore character; for example, `_QDStartUp`.
5. Pull any output from the top of the stack. The output values are described under each individual call in the two volumes of this manual.
6. Check for errors, if necessary, by examining the carry flag (c flag). If it is set to 1, an error occurred and the A register contains the error code in the following format:

   High-order byte = tool set number
   Low-order byte = error number

   With this method, an error can be properly identified, even if it occurs during a call from one tool set, but doesn't actually show up until a call from another tool set; for example, a QuickDraw II call can pass an error message from the Memory Manager. The error codes for each tool set are listed at the end of the relevant chapter, and all error codes are summarized in Appendix B, "Error Codes," in Volume 2.

Calling a routine from C

The interface libraries that allow the tool sets to be accessed from the C programming language are included in the Apple IIGS Workshop C. These libraries contain the function definitions for the tools. The steps to use a particular routine are as follows:

1. Make the routine accessible by using an `#include` statement that includes the appropriate file (for example, `#include <Quickdraw.h>` for the QuickDraw II routines). The included file will provide the function declarations and the necessary constants and data structures.
2. Invoke the call by entering its name and supplying the correct parameters. The parameters should be supplied according to Pascal-style conventions; that is, the parameters are pushed from left to right rather than from right to left. The parameters for each routine are described in the individual routine descriptions in the two volumes of this reference.
3. Examine the global variable `_toolErr` for errors, if necessary. If the variable is equal to 0, no errors occurred; otherwise, the error number will be present. The error numbers for each tool set are listed at the end of the relevant chapter and compiled in Appendix B, "Error Codes," in Volume 2.

In cases where a routine returns more than one value on the stack, the C definition usually calls for those values to be returned as a data structure. The definitions for these structures are in the appropriate C interface file. The type name for the structure is the return type as shown in the C synopsis for the routine. The field names for the structure are identical to the labels shown in the *Stack after call.*

Two exceptions to this rule are routines that return an X and Y coordinate and those that return a font ID. These are declared in C as returning a *LongWord* rather than a *Point* or a *FontID.* Frequently, you will only need to pass that result as is to another routine that takes a *Point* or a *FontID* as a parameter.

If you wish to access the individual fields of such results, you can't use the dot operator (that is, *.fieldname*) method. You can, for example, access them by the method `(FontID *)(&result) -> fontStyle`.

Passing parameters

Most input and output parameters for the tool calls are passed on the stack. The parameters and parameter-passing method are defined by each routine. Usually, the parameters are passed on the stack, with the routine pulling input parameters off and leaving any output parameters on the stack for the calling program to handle. The method and parameters for each routine are described under the relevant routine in the tool set chapters.

Return from a call

When a routine finishes, the routine returns control directly back to the application. The state of all flags and registers upon return from a tool set call is summarized in Table 2-3.

Table 2-3
Flags and registers on return from a call

Flag or register	Value*
n flag	As set by routine
v flag	As set by routine
m flag	Unchanged (must be 0)
x flag	Unchanged (must be 0)
d flag	Set to 0
i flag	Unchanged
z flag	As set by routine
c flag	As set by routine or error flag
e flag	Unchanged (must be 0)
A register	As set by routine or, if carry flag is set, A = error code
X register	As set by routine
Y register	As set by routine
S register	Parameters have been removed from stack
D register	Unchanged
P register	See preceding list of flags
DB register	Unchanged
PB register	Unchanged
PC register	Address following call

* "Unchanged" means that the value is the same as it was just before the routine call.

Chapter 3

Apple Desktop Bus Tool Set

The **Apple Desktop Bus (ADB)** is a method and a protocol for connecting input devices, such as keyboards and mice, with personal computers. The personal computer is considered to be the host during the communication, and it controls the communication on the bus by issuing **ADB commands** to the devices.

The **Apple Desktop Bus Tool Set** sends commands and data between the Apple Desktop Bus microcontroller and the rest of the system. Typically, the tool set is used to control ADB activity, but other commands, which are used by diagnostic routines and the Control Panel, are available.

Under normal circumstances, you won't need to use the ADB tool set. However, if you want to change how the system interfaces with existing or additional ADB devices, this tool set will be indispensable.

More details about the bus can be found in the *Apple IIGS Firmware Reference* and the *Apple IIGS Hardware Reference*. This chapter assumes that you understand how the ADB works.

A preview of the Apple Desktop Bus Tool Set routines

To introduce you to the capabilities of the Apple Desktop Bus Tool Set, all of its routines are grouped by function and briefly described in Table 3-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the Apple Desktop Bus Tool Set routines (discussed in alphabetical order).

Table 3-1
Apple Desktop Bus Tool Set routines and their functions

Routine	Description
Housekeeping routines	
ADBBootInit	Initializes the ADB Tool Set; called only by the Tool Locator—must not be called by an application
ADBStartUp	Starts up the ADB Tool Set
ADBShutDown	Shuts down the ADB Tool Set
ADBVersion	Returns the version number of the ADB Tool Set
ADBReset	Resets the ADB Tool Set; called only when the system is reset—must not be called by an application
ADBStatus	Indicates whether the ADB Tool Set is active
ADB routines	
SendInfo	Sends data to the keyboard microcontroller or to an ADB device
ReadKeyMicroData	Receives data from the keyboard microcontroller
ReadKeyMicroMemory	Reads a data byte from the keyboard microcontroller memory
AsyncADBReceive	Receives data from an ADB device
SyncADBReceive	Receives data from an ADB device
AbsOn	Enables automatic polling of an absolute device (reserved for future use)
AbsOff	Disables automatic polling of an absolute device (reserved for future use)
ReadAbs	Determines whether automatic polling of an absolute device is on or off
SetAbsScale	Sets up scaling for absolute devices (reserved for future use)
GetAbsScale	Reads absolute device scaling values (reserved for future use)
SRQPoll	Adds a device to the SRQ list
SRQRemove	Removes a device previously installed by the SRQPoll routine from the SRQ list
ClearSRQTable	Clears the SRQ list of all entries

About the Apple Desktop Bus commands

As you work with the ADB Tool Set, it's important for you to understand that the ADB commands are not the same as the ADB Tool Set routines. The ADB commands are at a lower-level than the tool set routines; that is, the tool set routines often include an ADB command as an input parameter to the routine. The ADB Tool Set then interprets and issues the ADB command. In a similar fashion to other tool sets, then, the ADB tool set provides a higher-level interface to a lower-level function.

Using other Apple Desktop Bus devices

An application that allows the use of specific ADB devices other than the mouse or keyboard must use a driver for that device. The device driver must have both setup routines and data handling routines.

The setup routines for the device driver must identify the devices on the ADB, possibly by changing ADB addresses and handlers. The data handling routines must contain a completion routine, called by the system when data is received from a device, and any other routines that operate on the data.

Polling the Apple Desktop Bus for data

The method you use to poll the Apple Desktop Bus for data will vary, depending upon whether your application is designed for a single user or for multiple users.

Polling single-user applications

A special tool mechanism called the **SRQ list** can be used to poll the ADB for data from specific devices. The system automatically starts polling the devices in the list whenever any device on the bus asserts SRQ. If data is received, that device's completion routine is called.

The SRQPoll routine is the most efficient way for a single-user application to gather data from an ADB device. This mechanism assumes that the user rarely switches between devices. Whenever SRQ is detected, the system always starts looking for data by polling the last device used.

Polling multiuser applications

In multiuser applications, such as games for two or more players, the SRQ list doesn't work efficiently because it always gives priority to the last device that returned data. For these applications, each device should be polled separately using the AsyncADBReceive routine. Devices can then be read in an arbitrary fashion, with no device getting priority (unless an application wants that), and the application can also regulate how often data is read. The latter feature is very important since it allows a game to adjust to the number of players.

Each device must have a unique address for polling, and the application must specify those addresses. Here are two suggested methods for assigning unique addresses.

The ADB Change Address When Activated handler

The simplest method for assigning a unique address to each device is to ask a player to hold down the activator button (mouse button or Apple key). You can then use the ADB Change Address When Activated handler. This command changes the address of any device that is currently activated.

After you verify that a device has changed addresses, tell the player to release the button. Repeat the request for each player, giving each device a new and different address, until all of the players' devices have been assigned unique addresses.

If this technique doesn't suit your taste (you may not wish to ask each player to hold down the button), there is a more complicated, but more automated, way to assign unique addresses, detailed in the next section.

The Collision Detect handler

You can use the Collision Detect handler to move each ADB device to a unique address. You request the ID at a specific address (TALK-REG.3), which forces a collision between the devices at this address, and then issue the Change Address command using the Collision Detect Handler. Any device that did not detect a collision will change its address.

It's possible that two (or more) devices may not detect a collision and both will move to the new address. To alleviate this problem, you should move the devices between the new addresses many times, thus increasing the odds that the devices will collide and only a single device will be moved.

For example, if an application needs to distinguish four keyboards from each other, it can

1. Use the SendInfo routine to send the ADB command TALK and an appropriate register and address (TALK register 3, address 2).
2. Issue the Change Address command to address 8, with the Collision Detect handler (=$FE). Any device that didn't detect a collision (at least one) will change to the new address.
3. Repeat step 2, changing address 8 to 9.
4. Move any device that stays at address 8 (it lost the collision) back to address 2.
5. Continue swapping the device between addresses 8 and 9 another 30 times, always moving any losers back to address 2. Swapping 32 times yields very good statistical odds that only one device will have its address changed to 8 from the original keyboard address.
6. Repeat the command for each keyboard, using two open addresses (such as 9 and 10, then 10 and 11, and so on).

7. After each keyboard has been moved to a new address, the application should ask each user to press a key. The keypress can then be used to identify the address of each user.

Important
The ADBReset and the ADBShutDown tool set routines do not clear the SRQ list or the ABSOLUTE flag. If your application installs a device, the application should use the SendInfo routine to issue a RESET ADB command. See the section "SendInfo" in this chapter.

Using the Apple Desktop Bus Tool Set

This section discusses how the ADB Tool Set routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

The ADB Tool Set depends upon the presence of the tool sets shown in Table 3-2 and requires that at least the indicated version of the tool set be present.

Table 3-2
Apple Desktop Bus Tool Set—
other tool sets required

Tool set number		Tool set name	Minimum version needed
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0

Your application should make an ADBStartUp call before making any other ADB Tool Set calls.

❖ *Note:* At the time of publication, the ADBStartUp call was not an absolute requirement, because the Tool Locator automatically started up the ADB Tool Set atboot time. However, you should make the call anyway, to guarantee that your application remains compatible with all future versions of the system.

Your application should also make the ADBShutDown call when the application quits.

Some commands can return an error code that indicates busy. This usually means that part of another command is currently active. Rather than queue the command, the tool set puts the burden on the calling routine to try again. A calling routine can retry the call immediately or can try it again later (perhaps by installing a task into the Heartbeat chain to remind the routine to try again).

The following examples assume that you are using the APW equate and macro files. The first example enables SRQ on a device at address 7.

```
ENSRQ EQU *
        PEA  $0000                  ; Count of 0 bytes (lengthByte)
        PEA  $0000                  ; Dummy address (not used since lengthByte is 0)
        PEA  $0000
        PEA  enableSRQ+7            ; Enable SRQ of address 7
        _SendInfo
        BCS  ERROR
```

The next example shows how to make a tool call to change the handler of an ADB device at address 7. It uses the ADB Tool Set routine SendInfo to transmit 2 bytes to register 3 at address 7.

```
        LDA  #$0207                 ; Change device at address 7 to handler 2
        STA  DATABUF                ; into data buffer
        PEA  $0002                  ; Count of 2 bytes to be sent on ADB (lengthByte)
        Pushlong #DATABUF           ; Data-buffer address
        PEA  transmit2ADBBytes+$37  ; Transmit 2 data bytes to register 3, ADB address 7
        _SendInfo
        BCS  ERROR
```

The next example sends data to an ADB device.

```
DATASND EQU  *
        PEA  $0005                  ; 5 data bytes (4 data & 1 ADB command)
        Pushlong #DATA              ; Data-buffer address with AxBy
        PEA  transmitADBBytes+$4    ; Command to microcontroller (transmit 4 data bytes)
        _SendInfo
        BCS  ERROR


DATA      DS   $8A,1,2,3,4          ; First byte = ADB device command,
*                                   ; device at address 8, Listen, register 2
*                                   ; Other bytes are data
```

❖ *Note:* The first byte sent is transmitted directly as the ADB device command.

The last example explains how to poll a device at address 7, register 0 for data.

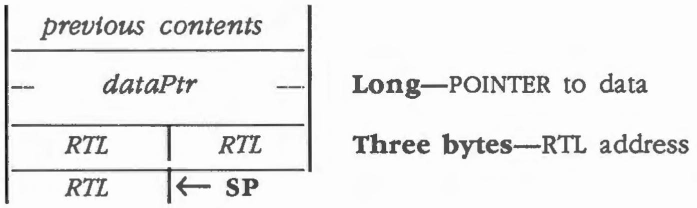
```
ADBPOLL anop
       PushLong #CPLTVC             ; Pointer to completion routine
       PEA  Talk+$07                ; Register 0, address 7 (if register 3, then TALK+$37)
       _AsyncADBREceive
       BCC  OK                             ; Everything OK
       CMP  BUSYERROR                      ; Check if busy error
       BEQ  ADBPOLL                 ; Poll again if busy
       BRA  ERROR
OK  EQU  *
END RTS                             ; End
```

Completion routines

All completion routines are called in 8-bit native mode. Two types of completion routines are currently defined: the AsyncADBReceive and the SRQ list.

Only a single completion routine can be active at a time. If an application wants to poll many devices sequentially, it should use the completion routine to initiate a poll of the next device.

The stack looks like this when the completion routine is called:



The data pointed to by *dataPtr* should look like Figure 3-1:

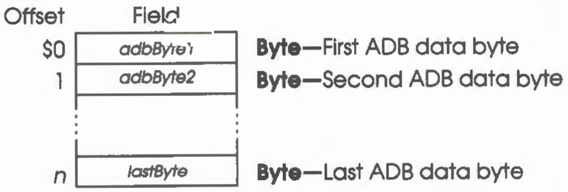


Figure 3-1
Apple Desktop Bus data

AsyncADBReceive completion routine

The **AsyncADBReceive completion routine** obtains data from a buffer pointed to by an address on the top of the stack. The first byte in the buffer contains the number of data bytes in the buffer. The first data byte received from the ADB is the next byte in the buffer, with subsequent data bytes received from the ADB stored sequentially in the buffer. The last (*n*th) byte received is the *n* +1 byte in the buffer.

```
CPLTVC          EQU *          ; Completion vector for AsyncADBReceive

                PHD            ; Move direct page onto stack minus 1

                TSC            ; Stack now has RTL address (3 bytes)

                TCD            ; Old direct page (2 bytes)

                LDA  [6]       ; Get length byte from buffer

                BEQ  ENDPOLL   ; No data remaining

                TAY            ; Set index

                INY            ; Index + 1 to get (length + 1) bytes

LP              LDA  [6],Y     ; Get data byte

                STA  BUF,Y     ; Move to application buffer

                DEY            ; Set index for next data byte

                BNE  LP

ENDPOLL         EQU *

                PLD            ; Restore direct page

                RTL            ; Return from completion routine
```

SRQ list completion routine

The **SRQ list completion routine** is very similar to the AsyncADBReceive routine. The only major difference is that an extra return address is on the stack when the routine is called. (These 3 extra bytes are left by the SRQ list handler.) Thus, the SRQ completion routine finds the data buffer address 3 bytes into the stack instead of on top of the stack.

```
SRQCPLT         EQU   *         ; Completion vector from SRQ list
                PHD             ; Save direct page
                TSC             ; Move direct page onto stack
                TCD             ; and perform indirect indexed long to
                LDA   [9]       ; get data length (NOT 0)
                TAY             ; Set index
                INY             ; Index + 1 to get (length + 1) bytes
                LDA   [9],Y     ; Get last data byte
                DEY             ; More data, etc.

ENDABS          PLD             ; Restore stack and return
                RTL
```

$0109 ADBBootInit

Initializes the ADB Tool Set; called only by the Tool Locator.

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 Call must not be made by an application.

$0209 ADBStartUp

Starts up the ADB Tool Set for use by an application.

❖ *Note:* At the time of publication, the ADBStartUp call was not an absolute requirement, because the Tool Locator automatically started up the ADB Tool Set at boot time. However, you should make the call anyway, to guarantee that your application remains compatible with all future versions of the system.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void ADBStartUp()
```

$0309 ADBShutDown

Shuts down the ADB Tool Set when an application quits.

Important

If your application has started up the ADB Tool Set, the application must make this call before it quits.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

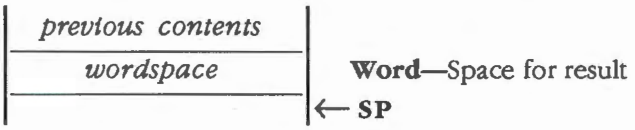
```
extern pascal void ADBShutDown()
```

$0409 ADBVersion

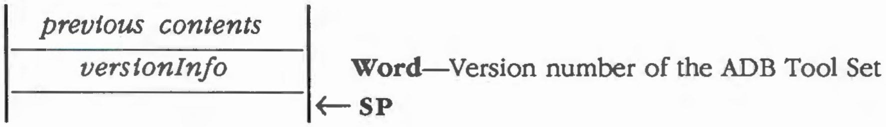
Returns the version number of the ADB Tool Set.

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal Word ADBVersion()
```

$0509 ADBReset

Resets the ADB Tool Set; called 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

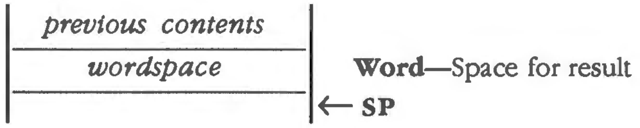
C Call must not be made by an application.

$0609 ADBStatus

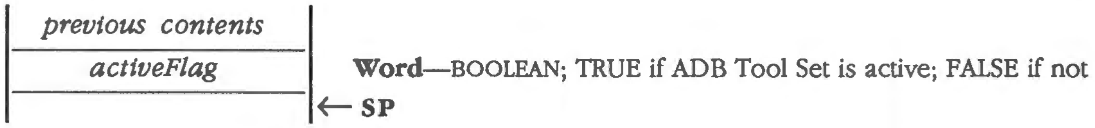
Indicates whether the ADB Tool Set is active.

Parameters

Stack before call



Stack after call


previous contents
activeFlag
Word—BOOLEAN; TRUE if ADB Tool Set is active; FALSE if not
← **SP**

Errors None

C `extern pascal Boolean ADBStatus()`

$0F09 AbsOn

Will enable automatic polling of an absolute device.

Important
At the time of publication, this routine was not implemented. The routine is reserved for future use.

$1009 AbsOff

Will disable automatic polling of an absolute device.

Important
At the time of publication, this routine was not implemented. The routine is reserved for future use.

$0D09 AsyncADBReceive

Receives data from an ADB device. The ADB command byte sent assumes that the command type is Talk, which tells the addressed device to send data to the host. Thus, the *adbCommand* to be sent is in the form

$C0+*xyabcd*

where $C0 is the Talk command, *xy* is the register, and *abcd* is the address.

Internally, AsyncADBReceive uses asynchronous communication; that is, the system sends the ADB command but doesn't wait for a response. The keyboard microcontroller notifies the system when the data is ready by sending a response byte that interrupts the system.

Important

The completion vector is called with 8-bit m and x flags and must return via an RTL instruction with the carry flag clear.

Parameters

Stack before call

previous contents	
-- *compPtr* --	**Long**—POINTER to completion routine
adbCommand	**Word**—ADB command, in form $C0+*xyabcd,* as described above
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0910	`cmndIncomplete`	Command not completed
$0982	`adbBusy`	ADB busy (command pending)

C

```
extern pascal void AsyncADBReceive(compPtr,adbCommand)

Pointer     compPtr;

Word      adbCommand;
```

$1609 ClearSRQTable

Clears the SRQ list of all entries.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void ClearSRQTable()
```

$1309 GetAbsScale

Reads absolute device scaling values, as set by the SetAbsScale routine. See the section "SetAbsScale" in this chapter.

Parameters

Stack before call

previous contents

-- *dataInPtr* -- **Long**—POINTER to ScaleRec; see "SetAbsScale" in this chapter

← **SP**

Stack after call

previous contents

← **SP**

Errors None

C

```
extern pascal void GetAbsScale(dataInPtr)

ScaleRecPtr     dataInPtr;
```

$1109 ReadAbs

Reads flags to determine whether automatic polling of absolute device is on or off.

Important

At the time of publication, this routine was not implemented. The routine is reserved for future use.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
autoABSPoll	**Word**—BOOLEAN; TRUE if polling on, FALSE if polling off
	← **SP**

Errors

None

C

Call must not be made by an application.

$0A09 ReadKeyMicroData

Receives data from the microcontroller.

Parameters

Stack before call

previous contents	
dataLength	**Word**—Number of bytes to be received (see Table 3-3)
— *dataPtr* —	**Long**—POINTER to data; NIL if no data (see Table 3-3)
adbCommand	**Word**—ADB command to be issued
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0910 `cmndIncomplete` Command not completed

C

```
extern pascal void ReadKeyMicroData(dataLength,dataPtr,adbCommand)

Word      dataLength;

Pointer     dataPtr;

Word      adbCommand;
```

Table 3-3
Apple Desktop Bus ReadKeyMicroData parameters

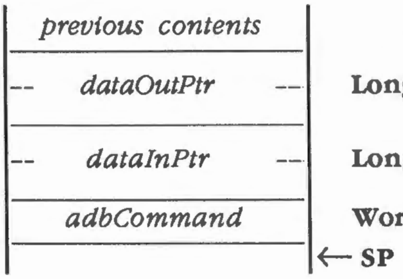
Com- mand	*dataLength*	Name	Action
0A	1	`readModes`	Read modes; data is a byte returning the mode
0B	3	`readConfig`	Read configuration; data is a data structure as follows: Byte Repeat delay/rate Byte Layout/language Byte ADB address, keyboard, and mouse
0C	1	`readADBError`	Read ADB error byte; data is a byte returning the error code
0D	1	`readVersionNum`	Read version number; data is a byte returning the version number
0E	1	`readAvailCharSet`	Read available character sets
0F	1	`readAvailLayout`	Read available keyboard layouts

$0B09 ReadKeyMicroMemory

Reads a data byte from keyboard microcontroller ROM or RAM.

Parameters

Stack before call

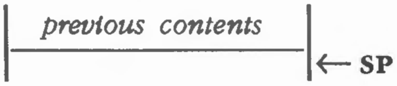


Long—POINTER to location to store results of read

Long—POINTER to keyboard memory location to be read

Word—ADB command to be issued, equal to $0009

Stack after call



Errors

$0910 `cmndIncomplete` Command not completed

C

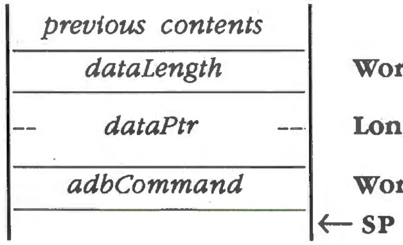
```
extern pascal void ReadKeyMicroMem(dataOutPtr,dataInPtr,adbCommand)

Pointer     dataOutPtr;

Pointer     dataInPtr;

Word      adbCommand;
```

$0909 SendInfo

Sends data to the microcontroller or an ADB device. The command and data to be sent are shown in Table 3-4.

Parameters

Stack before call

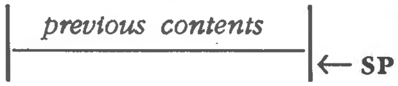


Word—Number of bytes of data to be sent (see Table 3-4)

Long—POINTER to data; NIL if no data (see Table 3-4)

Word—ADB command to be issued

Stack after call



Errors

$0910 cmndIncomplete Command not completed

C

```
extern pascal void SendInfo(dataLength,dataPtr,adbCommand)

Word      dataLength;

Pointer     dataPtr;

Word      adbCommand;
```

(continued)

SendInfo parameters

Table 3-4 summarizes the values for the SendInfo parameters. For more information about the ADB, see the *Apple IIGS Firmware Reference.*

Important

All of the commands in Table 3-4 that require more than a 1-byte transfer, except the Synch command, will automatically timeout in 10 milliseconds if there is no response. The Synch command will timeout in 20 milliseconds.

Table 3-4
Apple Desktop Bus SendInfo parameters

Command	*dataLength*	Name	Action
01	0	`abort`	Abort; no operation
02	0	`resetKbd`	Reset keyboard microcontroller
03	0	`flushKbd`	Flush keyboard
04	1	`setModes`	Set modes; data is a byte specifying the mode
05	1	`clearModes`	Clear modes; data is a byte specifying the mode
06	3	`setConfig`	Set configuration; data is a data structure as follows: Byte ADB addresss, keyboard, and mouse Byte Layout/language Byte Repeat delay/rate
07	4	`synch`	Synch; data is a data structure as follows: Byte Mode Byte ADB addresss, keyboard, and mouse Byte Layout/language Byte Repeat delay/rate
08	2	`writeMicroMem`	Write microcontroller memory; data is a data structure as follows: Byte Direct-page memory address Byte Data
10	0	`resetSys`	Reset system; pull the reset line low for 4 milliseconds
11	1	`keyCode`	Send ADB key code; data is a byte specifying the key code (for key codes, see Table 3-5). This command can be used to emulate an ADB keyboard by accepting key codes from a device and then sending them to the microcontroller to be processed as keystrokes. The command doesn't support buffering, nor will it process Reset Up or Reset Down codes; those codes must be handled before this command is used.

Table 3-4 (continued)
Apple Desktop Bus SendInfo parameters

Command	*dataLength*	Name	Action
40	0	`resetADB`	Reset Apple Desktop Bus; be careful with this command, because resetting an ADB keyboard clears all pending commands, including all key-up events. Thus, if a keystroke is used to launch this command, the key-up event will be lost, and the key will autorepeat until another key is pressed. All keys should be up before this command is executed.
4y	$n + 1$	`transmitADBBytes`	Transmit ADB bytes, where $y = 7 + n$ and n = number of bytes to transmit (n must be greater than or equal to 2 and less than or equal to 8); data is a data structure as follows: Byte ADB command (ADB type, address, register) Byte Data byte 1 Byte Data byte 2 . . . Byte Data byte 8
5x	0	`enableSRQ`	Enable SRQ (where x = ADB address in low nibble) **Important** Use the SRQPoll routine before calling SendInfo with the *enableSRQ* command.
6x	0	`flushADBDevBuf`	Flush buffer on ADB device (x = ADB address in low nibble); be careful with this command, because resetting an ADB keyboard clears all pending commands, including all key-up events. See the Reset ADB command in this table.
7x	0	`disableSRQ`	Disable SRQ (x = ADB address in low nibble); be careful with this command, because data pending may be lost when the command is executed. For example, if SRQ is disabled on the ADB keyboard, all key-up events may be lost. See the Reset ADB command in this table.
ry	2	`transmit2ADBBytes`	Transmit 2 ADB bytes (where $r = 8 +$ register, y = ADB address); data is a data structure as follows: Byte Data byte 1 Byte Data byte 2

(continued)

Apple Desktop Bus key codes

Table 3-5 summarizes the key codes that the Apple Desktop Bus microcontroller understands. Various keyboards can generate various codes. See the appropriate keyboard's hardware reference manual for that keyboard's actual codes.

Table 3-5
Key code specification

Key	Key code	Key	Key code	Key	Key code
ESC	$35	=	$51	(Key number 71)	$27
F1	$7A	/	$4B	Return	$24
F2	$7B	*	$43	4	$56
F3	$63	Tab	$30	5	$57
F4	$76	Q	$0C	6	$58
F5	$60	W	$0D	+	$45
F6	$61	E	$0E	L Shift	$38
F7	$62	R	$0F	Z	$06
F8	$64	T	$11	X	$07
F9	$65	Y	$10	C	$08
F10	$6D	U	$20	V	$09
F11	$67	I	$22	B	$0B
F12	$6F	O	$1F	N	$2D
F13	$69	P	$23	M	$2E
F14	$6B	[	$21	,	$2B
F15	$71	]	$1E	.	$2F
Reset	$7F7F	\	$2A	/	$2C
(Key number 18)	$32	|X>(Del)	$75	R Shift	$38
1	$12	End	$77	Up Arrow	$3E
2	$13	Page Down	$79	1	$53
3	$14	7	$59	2	$54
4	$15	8	$5B	3	$55
5	$17	9	$5C	L Control	$36
6	$16	-	$4E	L Option	$3A
7	$1A	Caps Lock	38	L Apple	37
8	$1C	A	00	Space	31
9	$19	S	01	R Apple	37
0	$1D	D	02	R Option	3A
-	$1B	F	03	R Control	36
=	$18	G	05	L Arrow	3B
Delete	$33	H	04	D Arrow	3D
Help	$72	J	26	R Arrow	3C
Home	$73	K	28	0	52
Page Up	$74	L	25	(Key number 104)	41
Clear	$47	;	29	Enter	4C

$1209 SetAbsScale

Sets up scaling for absolute devices. This routine will eventually allow a generic scaling desk accessory to support almost any size or brand graphics tablet.

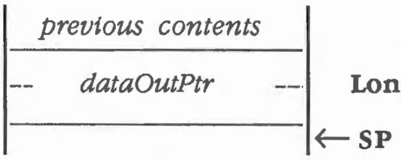
Important

At the time of publication, this routine was not implemented. The routine is reserved for future use.

The SetAbsScale routine will not perform scaling, but will leave the scaling to the absolute device.

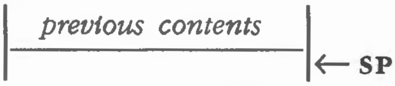
Parameters

Stack before call



Long—POINTER to scale record (see Figure 3-2)

Stack after call



Errors

None

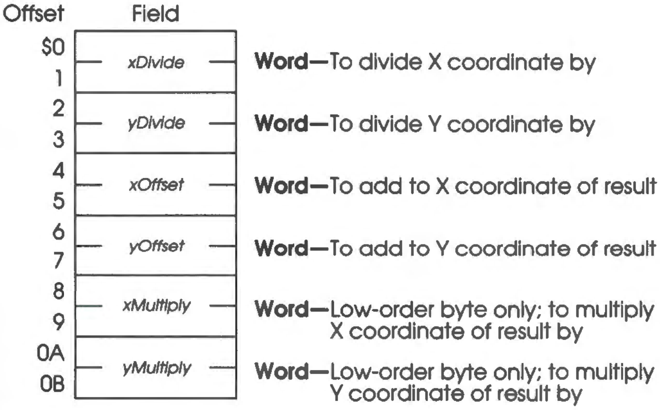
C

```
extern pascal void SetAbsScale(dataOutPtr)

ScaleRecPtr     dataOutPtr;
```

(continued)

Scale record

The scale record is shown in Figure 3-2.



Offset	Field	
$0 1	*xDivide*	**Word**—To divide X coordinate by
2 3	*yDivide*	**Word**—To divide Y coordinate by
4 5	*xOffset*	**Word**—To add to X coordinate of result
6 7	*yOffset*	**Word**—To add to Y coordinate of result
8 9	*xMultiply*	**Word**—Low-order byte only; to multiply X coordinate of result by
0A 0B	*yMultiply*	**Word**—Low-order byte only; to multiply Y coordinate of result by

Figure 3-2
Apple Desktop Bus scale record

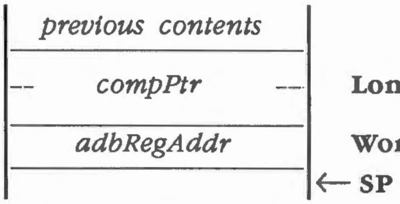
$1409 SRQPoll

Adds a device to the SRQ list (if the device exists) so that an application can be notified when this device has data. Whenever an SRQ is generated, the system automatically polls any device in the SRQ list to see if it has data ready. If data is available, the routine jumps to the specified completion routine with the data and notifies the application.

Important
The completion vector is called with 8-bit m and x flags and must return via an RTL instruction with the carry flag clear.

Parameters

Stack before call



Long—POINTER to completion routine

Word—ADB register and address

Stack after call

previous contents
← SP

Errors

$0910	`cmndIncomplete`	Command not completed
$0983	`devNotAtAddr`	Device not present at address
$0984	`srqListFull`	List full

C

```
extern pascal void SRQPoll(compPtr,adbRegAddr)
Pointer     compPtr;
Word      adbRegAddr;
```

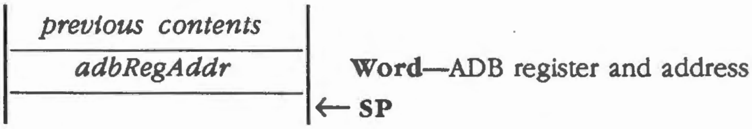
$1509 SRQRemove

Removes a device previously installed by the SRQPoll routine from the SRQ list.

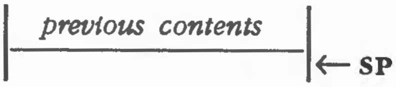
Important
The *adbRegAddr* must be the same as that used in the SRQPoll routine to install the device.

Parameters

Stack before call



Stack after call



Errors

$0910	`cmndIncomplete`	Command not completed
$0982	`adbBusy`	ADB busy (command pending)

C

```
extern pascal void SRQRemove(adbRegAddr)
Word      adbRegAddr;
```

$0E09 SyncADBReceive

Receives data from an ADB device. This routine is very similar to the AsyncADBReceive routine. However, SyncADBReceive needs the *inputWord* parameter, which specifies the ADB command Talk and the address and register. Thus, compute the command to be sent by using the form

$C0+*xyabcd*

where $C0 is the Talk command, *xy* is the register, and *abcd* is the address. When you have that result, swap its high and low nibbles and use that swapped form as the *inputWord*.

Important
The completion vector is called with 8-bit m and x flags and must return via an RTL instruction with the carry flag clear.

Parameters

Stack before call

previous contents	
inputWord	**Word**—ADB device command with high and low nibbles swapped
-- *compPtr* --	**Long**—POINTER to completion routine
adbCommand	**Word**—ADB command to be issued, equal to $0048
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0910	`cmndIncomplete`	Command not completed
$0982	`adbBusy`	ADB busy (command pending)

C

```
extern pascal void SyncADBReceive(inputWord,compPtr,adbCommand)

Word      inputWord;

Pointer     compPtr;

Word      adbCommand;
```

Apple Desktop Bus Tool Set summary

This section briefly summarizes the constants, data structures, and tool set errors contained in the Apple Desktop Bus Tool Set.

Important

These definitions are provided in the appropriate interface file.

Table 3-6
Apple Desktop Bus Tool Set constants

Name	Value	Description
ReadKeyMicroData ADB commands		
`readModes`	$000A	Read modes of ADB command
`readConfig`	$000B	Read configuration of ADB command
`readADBError`	$000C	Read ADB error byte of ADB command
`readVersionNum`	$000D	Read version number of ADB command
`readAvailCharSet`	$000E	Read available character sets
`readAvailLayout`	$000F	Read available keyboard layouts
ReadKeyMicroMEM ADB command		
`readMicroMem`	$0009	Read data byte from keyboard microcontroller
SendInfo ADB commands		
`abort`	$0001	Abort; no operation
`resetKbd`	$0002	Reset keyboard microcontroller
`flushKbd`	$0003	Flush keyboard
`setModes`	$0004	Set modes
`clearModes`	$0005	Clear modes
`setConfig`	$0006	Set configuration
`synch`	$0007	Synch
`writeMicroMem`	$0008	Write microcontroller memory
`resetSys`	$0010	Reset system
`keyCode`	$0011	Send ADB key code
`resetADB`	$0040	Reset ADB
`transmitADBBytes`	$0047	Transmit ADB bytes
`enableSRQ`	$0050	Enable SRQ
`flushADBDevBuf`	$0060	Flush buffer on ADB device
`disableSRQ`	$0070	Disable SRQ
`transmit2ADBBytes`	$0080	Transmit 2 ADB bytes
`listen`	$0080	ADB Listen command
`talk`	$00C0	ADB Talk command

Table 3-7
Apple Desktop Bus Tool Set data structures

Name	Offset	Type	Definition
ReadConfigRec (configuration record for ReadKeyMicroData)			
rcRepeatDelay	$0000	Byte	Repeat delay/rate
rcLayoutOrLang	$0001	Byte	Keyboard layout/language
rcADBAddr	$0002	Byte	ADB address, keyboard, and mouse
SetConfigRec (configuration record for SendInfo)			
scADBAddr	$0000	Byte	ADB address, keyboard, and mouse
scLayoutOrLang	$0001	Byte	Keyboard layout/language
scRepeatDelay	$0002	Byte	Repeat delay/rate
SynchRec (record for SendInfo)			
synchMode	$0000	Byte	Mode
synchKybdMouseAddr	$0001	Byte	ADB address, keyboard, and mouse
synchLayoutOrLang	$0002	Byte	Keyboard layout/language
synchRepeatDelay	$0003	Byte	Repeat delay/rate
ScaleRec (scale record)			
xDivide	$00	Word	Value to divide X coordinate by
yDivide	$02	Word	Value to divide Y coordinate by
xOffset	$04	Word	Value to add to X coordinate of result
yOffset	$06	Word	Value to add to Y coordinate of result
xMultiply	$08	Word	Value to multiply X coordinate by
yMultiply	$0A	Word	Value to multiply Y coordinate by

Note: The actual assembly-language equates have a lowercase *o* (the letter) in front of all of the names given in this table.

Table 3-8
Apple Desktop Bus Tool Set error codes

Code	Name	Description
$0910	`cmndIncomplete`	Command not completed
$0911	`cantSync`	Can't synchronize with system
$0982	`adbBusy`	ADB busy (command pending)
$0983	`devNotAtAddr`	Device not present at address
$0984	`srqListFull`	SRQ list full

Chapter 4

Control Manager

The **Control Manager** is the part of the Apple IIGS Toolbox that deals with controls. A **control** is an object on the IIGS screen with which the user, using the mouse, can cause instant action with graphic results or can change settings to modify a future action. Using the Control Manager, your application can

- Create or dispose of controls
- Display or hide controls
- Monitor the user's operation of a control with the mouse and respond accordingly
- Read or change the setting or other properties of a control
- Change the location or appearance of a control

Your application performs these actions by calling the appropriate Control Manager routines. The Control Manager carries out the actual operations, but it's up to you to decide when, where, and how.

Controls are of various types, each with its own characteristic appearance on the screen and responses to the mouse. Each individual control has its own specific properties—such as its location, size, and setting—but controls of the same type behave in the same general way.

A preview of the Control Manager routines

To introduce you to the capabilities of the Control Manager, all Control Manager routines are grouped by function and briefly described in Table 4-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the Control Manager routines (discussed in alphabetical order).

Table 4-1
Control Manager routines and their functions

Routine	Description
Housekeeping routines	
CtlBootInit	Initializes the Control Manager; called only by the Tool Locator—must not be called by an application
CtlStartUp	Starts up the Control Manager for use by an application
CtlShutDown	Shuts down the Control Manager
CtlVersion	Returns the version number of the Control Manager
CtlReset	Resets the Control Manager; called only when the system is reset—must not be called by an application
CtlStatus	Indicates whether the Control Manager is active
Initialization and termination routines	
CtlNewRes	Reinitializes resolution and mode
NewControl	Creates a control, adds it to the beginning of a specified window's control list, and returns a handle to the new control
DisposeControl	Deletes a specified control and releases the memory occupied by the control record and any data structures associated with the control
KillControls	Disposes of all controls associated with a specified window
HideControl	Makes a specified control invisible by filling the region the control occupies with the background pattern of the window's GrafPort
EraseControl	Makes a specified control invisible, but does not add the control's enclosing rectangle to the window's update region
ShowControl	Makes a specified control visible
DrawControls	Draws all controls currently visible in a specified window
DrawOneCtl	Draws a specified control
HiliteControl	Changes the way a specified control is highlighted
Mouse location routines	
FindControl	Tells in which of the window's controls, if any, the cursor was in when the user pressed the mouse button
TestControl	Tests which part of a specified control contains a specified point
TrackControl	Follows mouse movements and responds appropriately until the mouse button is released
Control movement routines	
MoveControl	Moves a specified control to a new location within its window
DragControl	Pulls a dotted outline of the control around the screen, following the movements of the mouse until the button is released

Table 4-1 (continued)
Control Manager routines and their functions

Routine	Description
Control record access routines	
SetCtlTitle	Sets a specified control's title to a given string and redraws the control
GetCtlTitle	Returns the value in a specified control's *ctlData* field, which, for controls with titles, is the pointer to the control's title string
SetCtlValue	Sets a specified control's *ctlValue* field to a specified value and redraws the control to reflect the new setting
GetCtlValue	Returns a specified control's current *ctlValue* field
SetCtlAction	Sets a specified control's *ctlAction* field to a new action
GetCtlAction	Returns the current value of a specified control's *ctlAction* field
SetCtlRefCon	Sets a specified control's *ctlRefCon* field to a new value
GetCtlRefCon	Returns the current value of a specified control's *ctlRefCon* field
SetCtlParams	Sets new parameters in the control's definition procedure, which will set the values and redraw the control if necessary
GetCtlParams	Returns a specified control's additional parameters
Miscellaneous routines	
DragRect	Pulls a dotted outline of a specified rectangle around the screen, following the movements of the mouse until the button is released
GetCtlDpage	Returns the value of the Control Manager's direct page
GrowSize	Returns the height and width of the size box control, using the Control Manager's current icon font
SetCtlIcons	Provides a handle to a specified new icon font

Standard controls

Certain standard types of controls are predefined for you. Your application can easily use controls of these standard types, and it can also define its own custom control types. The predefined control types are as follows:

- **Buttons** cause an immediate or continuous action when clicked or pressed with the mouse. They appear on the screen as rectangles with a title centered inside.
- **Check boxes** retain and display a setting, either checked (on) or unchecked (off); clicking with the mouse reverses the setting. On the screen, a check box appears as a small square with a title to the right of the square; the box is either filled in with an X (checked) or empty (unchecked). Check boxes are frequently used to control or modify some future action instead of causing an immediate action of their own. More than one box may be checked at any one time.

- **Radio buttons** also retain and display an on-or-off setting. They're organized into families; only one button in a family can be on at a time. Clicking any button on turns off all the others in the family, like the buttons on a car radio. Radio buttons are used to offer a single choice among several alternatives; the radio button that's on is filled with a small black circle.
- **Size boxes** provide a graphic symbol to represent the idea of resizing something. For example, the Window Manager provides a size box that can be used to increase or decrease the size of the window.
- **Dials** display a quantitative setting or value, typically in some pseudo-analog form such as the position of a sliding switch, the reading on a thermometer scale, or the angle of a needle on a gauge; the setting may be displayed digitally as well. The control's moving part that displays the current setting is called the **indicator.** The user may be able to change a dial's setting by dragging its indicator with the mouse, or the dial may simply display a value not under the user's direct control (such as the amount of free space remaining on a disk).

The standard controls and a few other typical controls are illustrated in Figure 4-1.

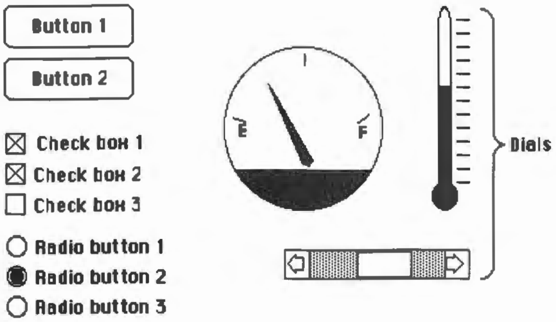


Figure 4-1
Standard and typical controls

Scroll bars

Scroll bars are predefined dials. The user clicks the mouse when the cursor is on a scroll bar arrow to scroll data a line at a time, or when the cursor is on a **paging region** to scroll a "page" at a time. The user can also drag the thumb box to any position within the scroll area. Although each of these may seem to behave like individual controls, they are all parts of a single control, the scroll-bar type of dial. You can define other dials of any shape or complexity if your application needs them.

Figure 4-2 shows the parts of the vertical and horizontal scroll bars.

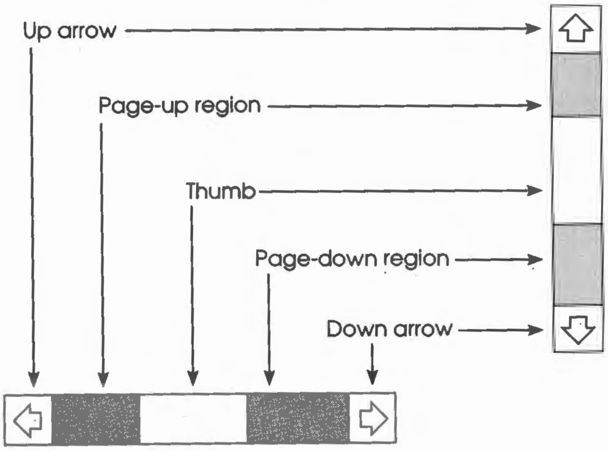


Figure 4-2
Parts of the scroll bars

Standard scroll bars are proportional; that is, they show the relationship between the total amount of data and the amount viewed and where the view is in the data, as illustrated in Figure 4-3.

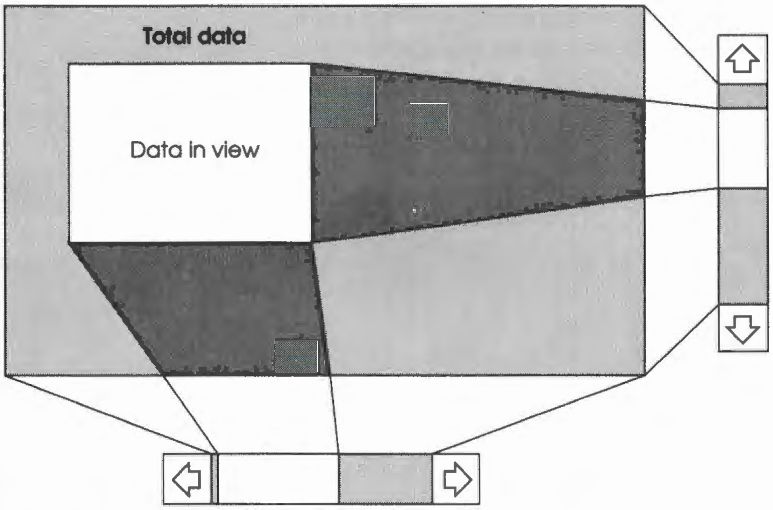


Figure 4-3
Scroll bar view

Assume that you want to use a single vertical scroll bar to the right of the text. The text has 300 lines, 30 of which can be displayed at one time. To set the scroll bar, you pass 30 for the view size and 300 for the data size in a SetCtlParams call.

When the user enters a line you want the scroll bar to be updated. To do this, you pass –1 for the view size because there was no change in the view, and you pass 301 for the data size to show the increased data size.

If you prefer, you can pass the view and data sizes as pixels rather than lines. If every line is 10 pixels high, and there are 300 lines, of which 30 can be displayed, you can pass 300 for the view size and 3000 for the data size. After the line is entered, you pass –1 for the view size (or 300 again) and 3010 for the data size. Because passing the number of pixels is proportionally equivalent to passing the number of lines, the scroll bar is identical using either method.

Active, inactive, and highlighted controls

A control may be **active** or **inactive.** Active controls respond to the user's mouse actions; inactive controls don't. A control is made inactive when it has no meaning or effect in the current context, such as an Open button when no document has been selected to open, or a scroll bar when there's currently nothing to scroll to.

An inactive control is shown in some special way, depending on its control type. Figure 4-4 illustrates some active and inactive controls.

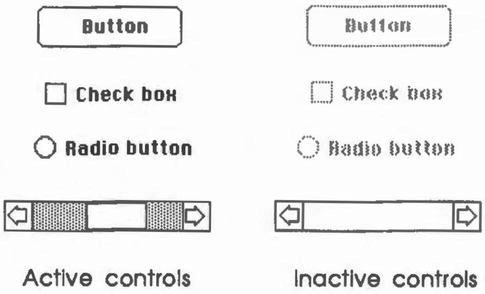


Figure 4-4
Active and inactive controls

Figure 4-4 also illustrates the two ways a scroll bar can be made inactive. The first way is to make the data size equal to or smaller than the view size. The second way is to pass 255 (`inactiveHilite`) to the Control Manager routine HiliteControl. This makes the scroll bar inactive in the same sense that the other controls are inactive.

You can also make controls invisible, which renders them inactive in the sense that they can't be selected.

If the user presses the mouse button when the cursor is over an active control, the control usually becomes highlighted. It's also possible for just a part of a control to become highlighted; for example, when the user presses the mouse button inside a scroll arrow in a scroll bar, the arrow, not the whole scroll bar, becomes highlighted. Figure 4-5 illustrates some highlighted controls.

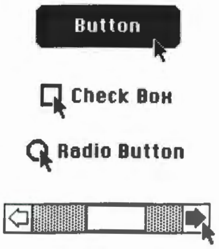


Figure 4-5
Highlighted active controls

Controls and windows

Every control belongs to a window. When the control is displayed, it appears within that window's content region; when the control is manipulated with the mouse, it acts on that window. All coordinates pertaining to the control (such as those describing its location) are given in the window's local coordinate system. Even the state of the control can be tied to the state of the window. A bit in the window's record can be set so the controls in the window will be considered inactive if the the window is inactive. See Chapter 25, "Window Manager," in Volume 2 for further information.

If you would like the controls in a window to scroll with the content region, make sure that the origin of the control's window is set to its scrolled value before you call the Control Manager.

Part codes

Some controls, such as buttons, are simple and straightforward. Others can be complex objects with many parts: for example, a scroll bar may have two scroll arrows, two paging regions, and a thumb. To allow different parts of a control to respond to the mouse in different ways, many of the Control Manager routines accept a **part code** as a parameter or return a part code as a result.

A part code is a number between 1 and 255 that stands for a particular part of a control. Each type of control has its own set of part codes. The part codes are assigned as shown in Table 4-2.

Table 4-2
Control Manager part codes

Code	Description	Code	Description
0	No part	11	Editable line
1	Reserved for internal use	12	User item
2	Simple button	13	Long static text
3	Check box	14	Icon
4	Radio button	15–31	Reserved for internal use
5	Up arrow	32–127	Reserved for application use
6	Down arrow	128	Reserved for internal use
7	Page up	129	Thumb
8	Page down	130–159	Reserved for internal use
9	Static text	160–253	Reserved for application use
10	Size box	254–255	Reserved for internal use

❖ *Note:* Some Control Manager routines need to give special treatment to the indicator of a dial, such as the thumb of a scroll bar. For the Control Manager to recognize them, such indicators always have part codes greater than 127.

Using the Control Manager

This section discusses how the Control Manager routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

The Control Manager depends upon the presence of the tool sets shown in Table 4-3 and requires that at least the indicated version of the tool set be present.

Table 4-3
Control Manager—other tool sets required

Tool set number		Tool set name	Minimum version needed
$01	#01	Tool Locator	1.2
$02	#02	Memory Manager	1.2
$03	#03	Miscellaneous Tool Set	1.2
$04	#04	QuickDraw II	1.2
$06	#06	Event Manager	1.0
$0E	#14	Window Manager	1.3

The first Control Manager call that your application must make is CtlStartUp. Conversely, when you quit your application, you must make the CtlShutDown call.

Where appropriate in your program, use the NewControl routine to add any controls you need. NewControl sets the control's owner to a specified window pointer and adds the control to the head of the window's control list. When you no longer need a control, call DisposeControl to remove it from its window's control list and erase it from the screen. To dispose of all of a window's controls at once, use KillControls.

❖ *Note:* The Window Manager procedure CloseWindow automatically disposes of all controls associated with a given window.

When the Window Manager routine TaskMaster (or the Event Manager routine GetNextEvent, if you're not using TaskMaster) reports that an update event has occurred for a window, your application should call DrawControls to redraw the window's controls as part of the process of updating the window.

When your application receives a mouse-down event, it should do the following:

1. If you used GetNextEvent to retrieve the mouse-down event, call FindWindow to determine which part of which window the cursor was in when the user pressed the mouse button. If you are using TaskMaster, this step is done for you.
2. If FindWindow or TaskMaster indicates that the mouse-down event was in the content region of the active window, use that window's control list.
3. If the event occurred in a content area, call FindControl with the pointer to the window to find out whether the event occurred on an active control.
4. If FindControl returns a control handle, call TrackControl to handle user interaction with the control. TrackControl handles the highlighting of the control and determines whether the mouse is still in the control when the mouse button is released. The routine also handles the dragging of the thumb in a scroll bar and responds to presses or clicks in the other parts of a scroll bar. When TrackControl returns the part code for a valid control, the application must respond appropriately.

The application's exact response to mouse activity in a control that retains a setting depends upon the current setting of the control (available from the GetCtlValue routine). For controls whose values can be set by the user, the SetCtlValue routine may be called to change the control's setting and redraw the control accordingly. For example, you can call SetCtlValue when a check box or radio button is clicked to change the setting and draw or clear the mark inside the control.

When you need to, you can call HideControl to make a control invisible or ShowControl to make it visible. Similarly, you can call MoveControl, which simply changes a control's location without pulling around an outline of it, at any time. For example, when the user changes the size of a document window that contains a scroll bar, you can call HideControl to remove the old scroll bar, MoveControl to change its location, and ShowControl to display it as changed.

Whenever necessary, you can read various attributes of a control with GetCtlAction, GetCtlTitle, GetCtlRefCon, or GetCtlParams; similarly, you can change those attributes with SetCtlAction, SetCtlTitle, SetCtlRefCon, or SetCtlParams.

Control Manager icon font

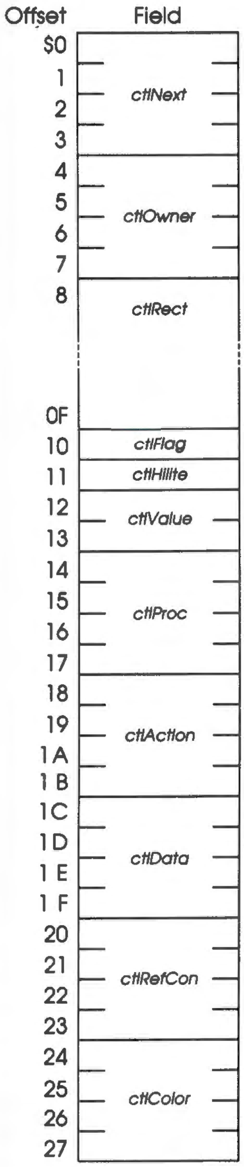
The standard control definition procedures use a font to draw some control parts and their highlighted states. If you would like to use different icons, you can replace the default font with the SetCtlIcons call, which replaces the font or returns the handle to the current font. The font format is shown in Table 4-4.

Table 4-4
Control Manager icon font format

Character	Description
0	Check box off and not highlighted
1	Check box off and highlighted
2	Check box on and not highlighted
3	Check box on and highlighted
4	Radio button off and not highlighted
5	Radio button off and highlighted
6	Radio button on and not highlighted
7	Radio button on and highlighted
8	Up arrow not highlighted
9	Up arrow highlighted
10	Down arrow not highlighted
11	Down arrow highlighted
12	Left arrow not highlighted
13	Left arrow highlighted
14	Right arrow not highlighted
15	Right arrow highlighted
16	Size icon

Control records

The **control record** contains fields that define the behavior and appearance of the control. The first few fields of every control record are defined as shown in Figure 4-6. Additional data can be appended to the end of those fields; the NewControl routine calls the control's definition procedure to find out the size of the record to allocate before the record is actually allocated. The generic control record is illustrated in Figure 4-6.



Offset	Field
$0	*ctlNext*
1	
2	
3	
4	*ctlOwner*
5	
6	
7	
8	*ctlRect*
0F	
10	*ctlFlag*
11	*ctlHilite*
12	*ctlValue*
13	
14	*ctlProc*
15	
16	
17	
18	*ctlAction*
19	
1A	
1 B	
1C	*ctlData*
1D	
1 E	
1 F	
20	*ctlRefCon*
21	
22	
23	
24	*ctlColor*
25	
26	
27	

Figure 4-6
Generic 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 zero value; as new controls are created, they're added to the beginning of the list.

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

ctlRect: The rectangle that defines the control's position and size in the local coordinates of the control's window. For simple buttons, radio buttions, and check buttons, if you leave the bottom and right coordinates of the rectangle as 0, the Control Manager will calculate the values for you.

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. The values for *ctlHilite* are as follows:

0	Control active; no highlighted parts
1–254	Part code of a highlighted part of the control
255	Control inactive

Only one part on a control can be highlighted at any one time, and no part can be highlighted on an inactive control. See the section "HiliteControl" in this chapter for more information.

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

ctlProc: For standard controls, the high-order byte determines the type of standard control, and bits 23–0 are zero. Values for standard controls are as follows:

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

For nonstandard controls, *ctlProc* is a pointer to the control definition procedure for this type of control.

ctlAction: Pointer to the control's custom action procedure, if any. The procedure TrackControl may call the custom action procedure to respond to the user's dragging the mouse inside the control. See the section "TrackControl" in this chapter for more information.

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

ctlRefCon: This field is reserved for application use.

ctlColor: Pointer to the control's color table, which is used by the control's definition procedure to draw the control. NIL causes a default to a standard color table defined by the control's definition procedure.

More fields can be added to the end of the control record to further define the control. See the section "Scroll Bar Control Record" in this chapter for an example and the section "Defining Your Own Controls" in this chapter for more information.

Control record fields used by standard controls are shown in the next sections.

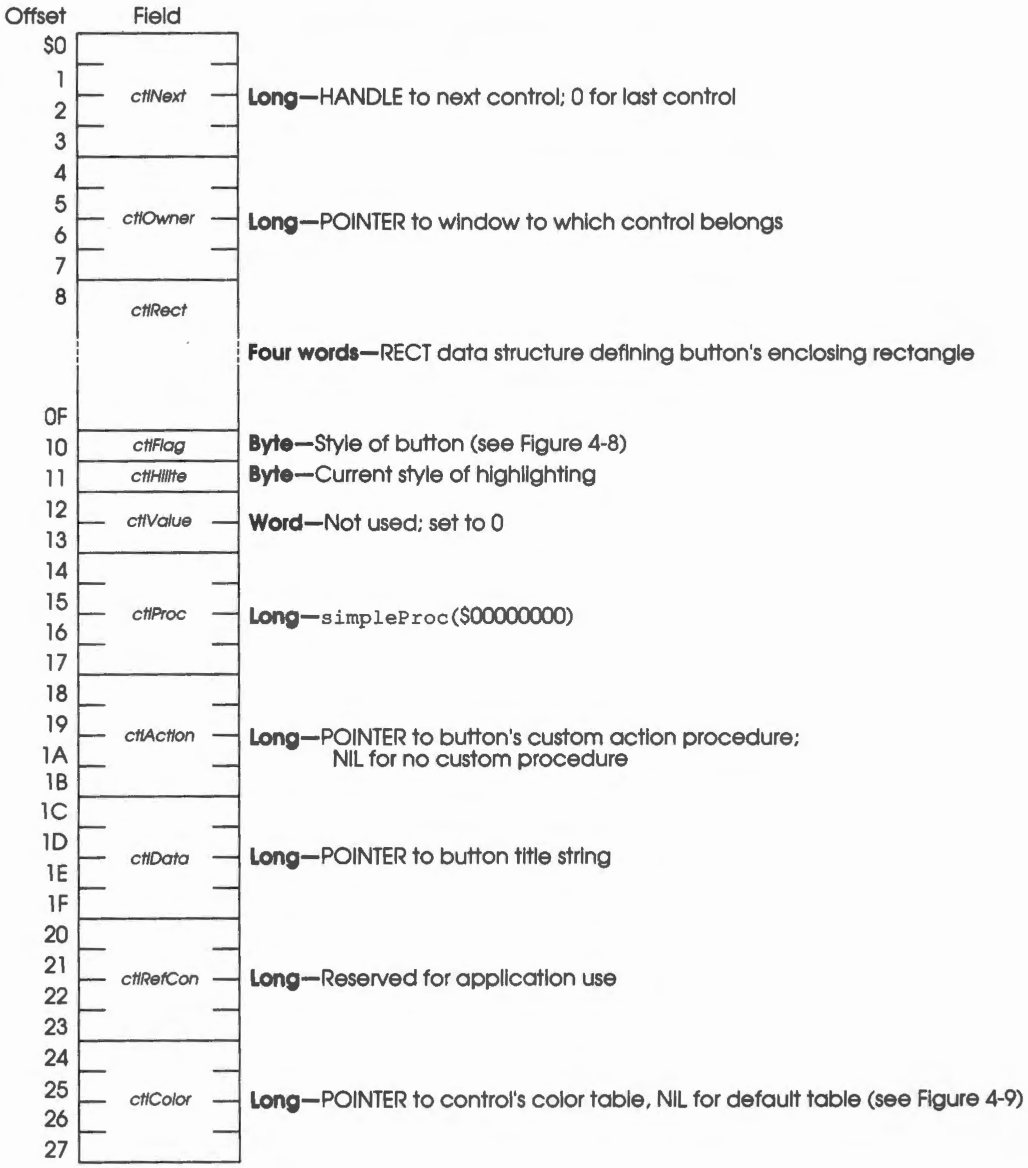
Simple button control records

A button causes an immediate or continuous action when the user clicks it with the mouse. A simple button can be round-cornered or square-cornered and can have a single or a bold outline. You should use the bold outline on buttons that the user can use to select default values by pressing the Return key. By convention, a default selection should never cause the destruction of something (you shouldn't, for example, use a default *Delete File*). Use a single outline for all other simple buttons.

A square-cornered button can have a drop shadow.

❖ *Note:* Simple buttons with thick outlines and drop shadows are the only standard controls that are drawn outside of their control rectangle.

Figure 4-7 shows a simple button control record.



Offset	Field	
$0–3	*ctlNext*	**Long**—HANDLE to next control; 0 for last control
4–7	*ctlOwner*	**Long**—POINTER to window to which control belongs
8–0F	*ctlRect*	**Four words**—RECT data structure defining button's enclosing rectangle
10	*ctlFlag*	**Byte**—Style of button (see Figure 4-8)
11	*ctlHilite*	**Byte**—Current style of highlighting
12–13	*ctlValue*	**Word**—Not used; set to 0
14–17	*ctlProc*	**Long**—`simpleProc`($00000000)
18–1B	*ctlAction*	**Long**—POINTER to button's custom action procedure; NIL for no custom procedure
1C–1F	*ctlData*	**Long**—POINTER to button title string
20–23	*ctlRefCon*	**Long**—Reserved for application use
24–27	*ctlColor*	**Long**—POINTER to control's color table, NIL for default table (see Figure 4-9)

Figure 4-7
Simple button control record

The *ctlFlag* value defines the button style, as shown in Figure 4-8.

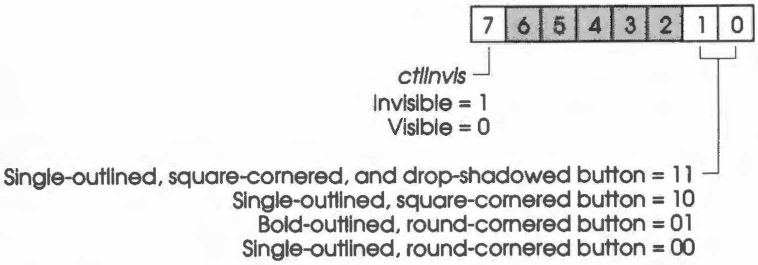

7
6
5
4
3
2
1
0
ctlInvis
Invisible = 1
Visible = 0
Single-outlined, square-cornered, and drop-shadowed button = 11
Single-outlined, square-cornered button = 10
Bold-outlined, round-cornered button = 01
Single-outlined, round-cornered button = 00

Figure 4-8
Simple button control flag

The *ctlColor* field points to the simple button color table, which is defined as shown in Figure 4-9.

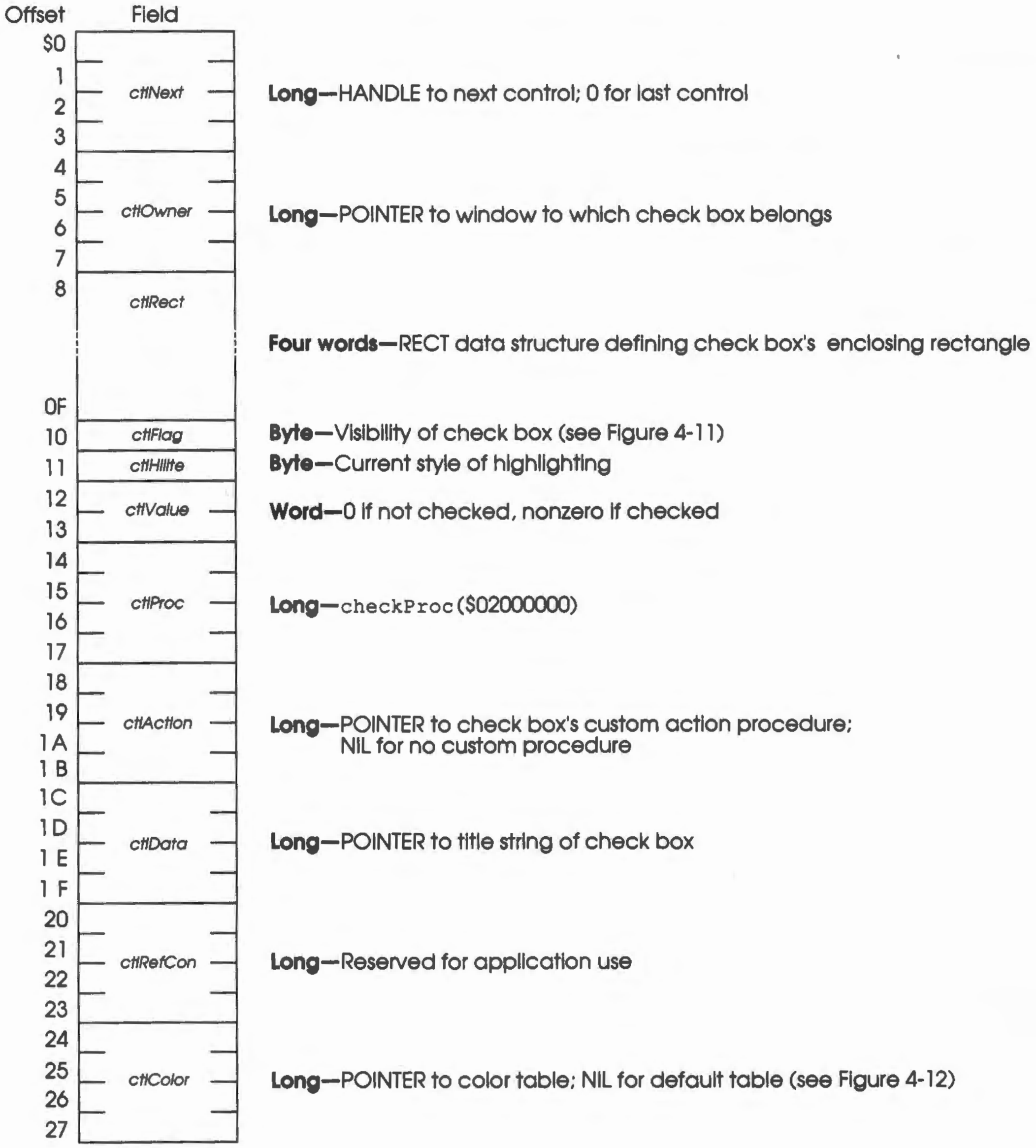
Offset	Field	
$0 1	*bttnOutline*	**Word**—Outline color Bits 15-8 = 0 7-4 = Outline color (bold outline and drop shadow, if used) 3-0 = 0
2 3	*bttnNorBack*	**Word**—Interior color when not highlighted Bits 15-8 = 0 7-4 = Background color 3-0 = 0
4 5	*bttnSelBack*	**Word**—Interior color when highlighted (selected) Bits 15-8 = 0 7-4 = Background color 3-0 = 0
6 7	*bttnNorText*	**Word**—Text color when not highlighted Bits 15-8 = 0 7-4 = Background color (only bits 5-4 used in 640 mode) 3-0 = Foreground color (only bits 1-0 used in 640 mode)
8 9	*bttnSelText*	**Word**—Text color when highlighted (selected) Bits 15-8 = 0 7-4 = Background color (only bits 5-4 used in 640 mode) 3-0 = Foreground color (only bits 1-0 used in 640 mode)

Figure 4-9
Simple button color table

Check box control record

Check boxes retain and display a setting, either checked (on) or not checked (off); clicking with the mouse reverses the setting. On the screen, a check box appears as a small square with a title to the right of the square; the box is either filled in with an X (checked) or is empty (not checked). Check boxes are frequently used to control or modify some future action, instead of causing an immediate action of their own. More than one box may be checked at any one time.

Figure 4-10 shows a check box control record.



Offset	Field	
$0 1 2 3	*ctlNext*	**Long**—HANDLE to next control; 0 for last control
4 5 6 7	*ctlOwner*	**Long**—POINTER to window to which check box belongs
8 0F	*ctlRect*	**Four words**—RECT data structure defining check box's enclosing rectangle
10	*ctlFlag*	**Byte**—Visibility of check box (see Figure 4-11)
11	*ctlHilite*	**Byte**—Current style of highlighting
12 13	*ctlValue*	**Word**—0 if not checked, nonzero if checked
14 15 16 17	*ctlProc*	**Long**—`checkProc`($02000000)
18 19 1A 1B	*ctlAction*	**Long**—POINTER to check box's custom action procedure; NIL for no custom procedure
1C 1D 1E 1F	*ctlData*	**Long**—POINTER to title string of check box
20 21 22 23	*ctlRefCon*	**Long**—Reserved for application use
24 25 26 27	*ctlColor*	**Long**—POINTER to color table; NIL for default table (see Figure 4-12)

Figure 4-10
Check box control record

The *ctlFlag* value defines whether the check box is visible or invisible, as shown in Figure 4-11.

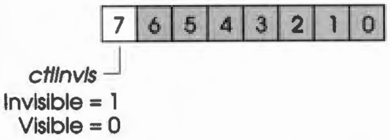
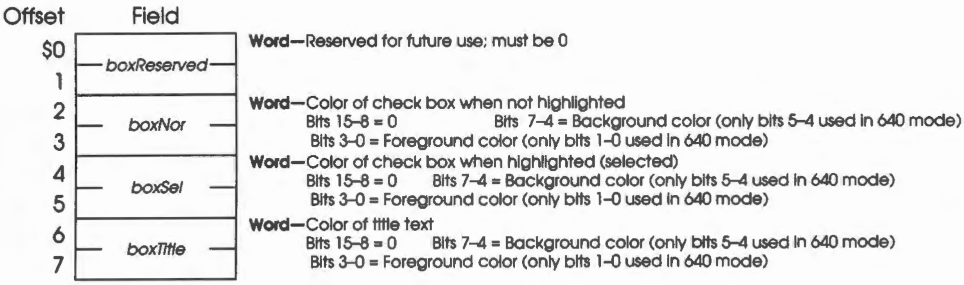


Figure 4-11
Check box control flag

The pointer in the *ctlColor* field points to the check box color table, which is defined as shown in Figure 4-12.



Offset	Field	
$0 1	*boxReserved*	**Word**—Reserved for future use; must be 0
2 3	*boxNor*	**Word**—Color of check box when not highlighted Bits 15–8 = 0 Bits 7–4 = Background color (only bits 5–4 used in 640 mode) Bits 3–0 = Foreground color (only bits 1–0 used in 640 mode)
4 5	*boxSel*	**Word**—Color of check box when highlighted (selected) Bits 15–8 = 0 Bits 7–4 = Background color (only bits 5–4 used in 640 mode) Bits 3–0 = Foreground color (only bits 1–0 used in 640 mode)
6 7	*boxTitle*	**Word**—Color of title text Bits 15–8 = 0 Bits 7–4 = Background color (only bits 5–4 used in 640 mode) Bits 3–0 = Foreground color (only bits 1–0 used in 640 mode)

Figure 4-12
Check box color table

Radio button control record

Radio buttons also retain and display an on-or-off setting. They're organized into families; only one button in a family can be on at a time. Clicking any button on turns off all the others in the family, like the buttons on a car radio. Radio buttons are used to offer a single choice among several alternatives; the radio button that's on is filled with a small black circle.

Figure 4-13 shows a radio button control record.

Offset	Field	
$0 1 2 3	*ctlNext*	**Long**—HANDLE to next control; 0 for last control
4 5 6 7	*ctlOwner*	**Long**—POINTER to window to which radio button belongs
8 0F	*ctlRect*	**Four words**—RECT data structure defining radio button's enclosing rectangle
10	*ctlFlag*	**Byte**—Visibility and family of button
11	*ctlHilite*	**Byte**—Current style of highlighting
12 13	*ctlValue*	**Word**—0 if off, nonzero if on
14 15 16 17	*ctlProc*	**Long**—`radioProc`($04000000)
18 19 1A 1B	*ctlAction*	**Long**—POINTER to radio button's custom action procedure; NIL for no custom procedure
1C 1D 1E 1F	*ctlData*	**Long**—POINTER to title string of radio button
20 21 22 23	*ctlRefCon*	**Long**—Reserved for application use
24 25 26 27	*ctlColor*	**Long**—POINTER to color table; NIL for default table (see Figure 4-15)

Figure 4-13
Radio button control record

The *ctlFlag* field is defined as shown in Figure 4-14.

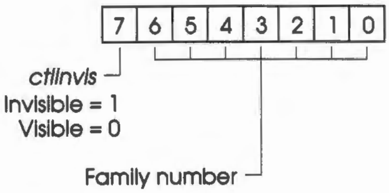


Figure 4-14
Radio button control flag

The pointer in *ctlColor* points to the radio button color table, which is defined as shown in Figure 4-15.

Offset	Field	
$0 1	*radReserved*	**Word**—Reserved for future use; must be 0
2 3	*radNor*	**Word**—Color of radio button when not highlighted Bits 15–8 = 0 Bits 7–4 = Background color (only bits 5-4 used in 640 mode) Bits 3–0 = Foreground color (only bits 1–0 used in 640 mode)
4 5	*radSel*	**Word**—Color of radio button when highlighted (selected) Bits 15–8 = 0 Bits 7–4 = Background color (only bits 5-4 used in 640 mode) Bits 3–0 = Foreground color (only bits 1–0 used in 640 mode)
6 7	*radTitle*	**Word**—Color of title text Bits 15–8 = 0 Bits 7–4 = Background color (only bits 5-4 used in 640 mode) Bits 3–0 = Foreground color (only bits 1–0 used in 640 mode)

Figure 4-15
Radio button color table

Scroll bar control record

Scroll bars are predefined dials. Arrows in the scroll bars scroll data one line at a time, paging regions scroll data a "page" at a time, and the thumb can be dragged to represent any position within the data area.

Figure 4-16 shows a scroll bar control record.

Offset	Field	
$0 1 2 3	*ctlNext*	**Long**—Handle to next control; NIL for last control
4 5 6 7	*ctlOwner*	**Long**—POINTER to window to which scroll bar belongs
8 0F	*ctlRect*	**Four words**—RECT data structure defining scroll bar's enclosing rectangle
10	*ctlFlag*	**Byte**—Style of scroll bar (see Figure 4–17)
11	*ctlHilite*	**Byte**—Current style of highlighting
12 13	*ctlValue*	**Word**—Number between 0 and data size minus view size
14 15 16 17	*ctlProc*	**Long**—`scrollProc`($06000000)
18 1B	*ctlAction*	**Long**—POINTER to scroll bar's custom action procedure; NIL for no custom procedure
1C 1F	*ctlData*	**Long**—High-order word is data size; low-order word is view size
20 23	*ctlRefCon*	**Long**—Reserved for application use
24 27	*ctlColor*	**Long**—POINTER to color table; NIL for default table (see Figure 4-18)
28 2F	*thumbRect*	**Four words**—RECT data structure defining thumb rectangle
30 38	*pageRegion*	**Four words**—RECT data structure defining page region, thumb bounds

Figure 4-16
Scroll bar control record

The *ctlFlag* field is defined as shown in Figure 4-17.

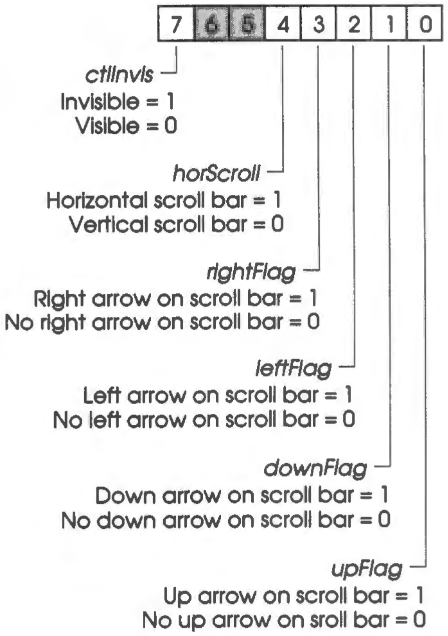

7
6
5
4
3
2
1
0
ctlInvis
Invisible = 1
Visible = 0
horScroll
Horizontal scroll bar = 1
Vertical scroll bar = 0
rightFlag
Right arrow on scroll bar = 1
No right arrow on scroll bar = 0
leftFlag
Left arrow on scroll bar = 1
No left arrow on scroll bar = 0
downFlag
Down arrow on scroll bar = 1
No down arrow on scroll bar = 0
upFlag
Up arrow on scroll bar = 1
No up arrow on sroll bar = 0

Figure 4-17
Scroll bar control flag

The pointer in *ctlColor* points to the scroll bar color table, which is defined as shown in Figure 4-18.

Offset	Field	
$0	*barOutline*	**Word**—Outline color Bits 15–8 = 0 Bits 7–4 = Outline color for arrow boxes, thumb, page region Bits 3–0 = 0
1		
2	*barNorArrow*	**Word**—Color of arrows when not highlighted Bits 15–8 = 0 Bits 7–4 = Background color (only bits 5-4 used in 640 mode) Bits 3–0 = Foreground color (only bits 1-0 used in 640 mode)
3		
4	*barSelArrow*	**Word**—Color of arrows when highlighted (selected) Bits 15–8 = 0 Bits 7–4 = Background color (only bits 5-4 used in 640 mode) Bits 3–0 = Foreground color (only bits 1-0 used in 640 mode)
5		
6	*barArrowBack*	**Word**—Color of arrow box interior background Bits 15–8 = 0 Bits 7–4 = Background color Bits 3–0 = 0
7		
8	*barNorThumb*	**Word**—Thumb's interior color when not highlighted Bits 15–8 = 0 Bits 7–4 = Background color Bits 3–0 = 0
9		
A	*barSelThumb*	**Word**—INTEGER; Reserved for future use; can be any value
B		
C	*barPageRgn*	**Word**—Page region's interior color Bits 15–9 = 0 Bit 8 = 1 for dotted pattern, 0 for solid pattern Bits 7–4 = Background of pattern, or color if solid Bits 3–0 = Foreground of pattern
D		
E	*barInactive*	**Word**—Color of scroll bar's interior when inactive Bits 15-8 = 0 7-4 = Background color 3-0 = 0
F		

Figure 4-18
Scroll bar color table

Size box control record

Size boxes provide a graphic symbol to represent the idea of resizing something. Figure 4-19 shows a size box control record.

Offset	Field	
$0	*ctlNext*	**Long**—HANDLE to next control; 0 for last control
1		
2		
3		
4	*ctlOwner*	**Long**—POINTER to window to which size box belongs
5		
6		
7		
8	*ctlRect*	**Four words**—RECT data structure defining size box's enclosing rectangle
0F		
10	*ctlFlag*	**Byte**—Visibility of size box (see Figure 4-20)
11	*ctlHilite*	**Byte**—Not used
12	*ctlValue*	**Word**—Not used
13		
14	*ctlProc*	**Long**—`growProc` ($08000000)
15		
16		
17		
18	*ctlAction*	**Long**—POINTER to control's custom action procedure; NIL for no custom procedure
19		
1A		
1B		
1C	*ctlData*	**Long**—Not used
1D		
1E		
1F		
20	*ctlRefCon*	**Long**—Reserved for application use
21		
22		
23		
24	*ctlColor*	**Long**—POINTER to color table; NIL for default table (see Figure 4-21)
25		
26		
27		

Figure 4-19
Size box control record

The *ctlFlag* value defines the visibility of the size box, as shown in Figure 4-20.

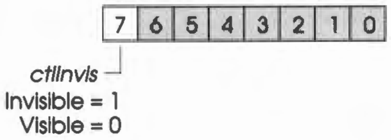


Figure 4-20
Size box control flag

The pointer in *ctlColor* points to the size box color table, which is defined as shown in Figure 4-21.

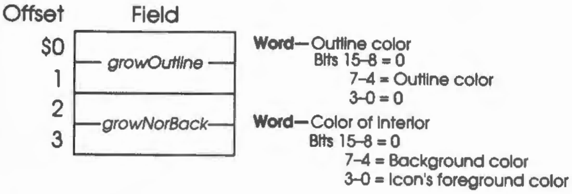


Figure 4-21
Size box color table

Defining your own controls

In addition to using the predefined controls, you can also define custom controls of your own. Perhaps you need a three-way selector switch, a memory-space indicator that looks like a thermometer, a thruster control for a spacecraft simulator, or some other special type of control. Controls and their indicators can occupy regions of any shape.

To define your own type of control, you place a **control definition procedure** in your application. The Control Manager stores the address of the procedure in the *ctlProc* field of the control record when you create the control with a NewControl routine. Later, when the Control Manager needs to perform a type-dependent action on the control, it calls the control definition procedure.

The definition procedure inputs and output are placed on the stack as follows:

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
ctlMessage	**Word**—Desired operation, as shown in Table 4-5
-- *ctlParam* --	**Long**—Value depends on operation
--*theControlHandle* --	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
-- *retValue* --	**Long**—Value depends on operation
	← **SP**

The *ctlMessage* parameter identifies the desired operation. It has one of the values shown in Table 4-5.

Table 4-5
Control Manager message parameters

Value	Message	Description
0	`drawCtl`	Draw the control (or control part)
1	`calcCRect`	Compute the rectangle to drag
2	`testCtl`	Test where mouse button was pressed
3	`initCtl`	Perform any additional control initialization
4	`dispCtl`	Take any additional disposal actions
5	`posCtl`	Move the control's indicator
6	`thumbCtl`	Compute the parameters for dragging an indicator
7	`dragCtl`	Drag either a control's indicator or the entire control
8	`autoTrack`	Called while dragging if –1 is passed to TrackControl
9	`newValue`	Called when the control gets a new value
10	`setParams`	Called when control gets a new additional parameters
11	`moveCtl`	Called when control moves; compute new position for parts
12	`recSize`	Return control record size (in bytes)

The value for *ctlParam* depends on the operation. Similarly, the control definition procedure returns a *retValue* only where indicated; in other cases, the routine can ignore *retValue,* since it is not used.

Draw routine

The message `drawCtl` asks the control definition procedure to draw all or part of the control within its enclosing rectangle. The low-order word of *ctlParam* is a part code specifying the part of the control to draw or 0 for the entire control. If the control is invisible, there's nothing to do; if it's visible, the definition procedure should draw the control (or the requested part), taking into account the current highlighting and value.

If *ctlParam* is the part code of the control's indicator, the draw routine can assume that the indicator hasn't moved; it might be called, for example, to highlight the indicator.

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
ctlMessage	**Word**—`drawCtl` message
-- *ctlParam* --	**Long**—(Low word only) If part code, draw part; if 0, draw control
--*theControlHandle* --	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
-- *retValue* --	**Long**—Undefined
	← **SP**

Test routine

The Control Manager routine TestControl sends the message `testCtl` to the control definition procedure when the mouse button is pressed in a visible control. This message asks in which part of the control, if any, a given point lies. The point is passed as the value of *ctlParam* in the local coordinates of the control's window; the vertical coordinate is in the low-order word of the long integer, and the horizontal coordinate is in the high-order word. The control definition procedure should return the part code for the part of the control that contains the point; it should return 0 if the point is outside the control or if the control is inactive.

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
ctlMessage	**Word**—`testCtl` message
-- *ctlParam* --	**Long**—Point in window's local coordinates; low word = Y, high word = X
--*theControlHandle* --	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
-- *retValue* --	**Long**—Part code if part contains point, 0 if it does not
	← **SP**

Calculate indicator rectangle routine

Just before the Control Manager starts to drag a control or its indicator, it calls the control's definition procedure to determine the coordinates of the control or its indicator. The highest bit of *ctlParam* must be 0 if the entire control is to be dragged, or 1 if the control's indicator is to be dragged.

Stack before call

previous contents	
— *longspace* —	**Long**—Space for result
ctlMessage	**Word**—`calcCRect` message
— *ctlParam* —	**Long**—RECT data structure address
—*theControlHandle* —	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
— *retValue* —	**Long**—0 for default RECT, nonzero if RECT is set
	← **SP**

If the definition procedure returns 0 and the entire control is to be dragged, the RECT pointed to by *ctlParam* is set to the control's enclosing rectangle.

If the definition procedure returns 0 and the control's indicator is to be dragged, the Control Manager assumes that the record is set up like a scroll bar record, and the RECT is set to the thumb rectangle. See the section "Scroll Bar Control Record" in this chapter.

Initialize routine

After allocating and initializing the control record as appropriate when creating a new control, the Control Manager sends the message `initCtl` to the control definition procedure. The definition procedure can then perform any type-specific initialization it requires. For example, the control definition procedure for scroll bars initializes the thumb and page RECTs and also stores *param1* and *param2* in the *ctlData* field. The initialize routine for standard buttons, check boxes, and radio buttons does nothing.

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
ctlMessage	**Word**—`initCtl` message
-- *ctlParam* --	**Long**—Low word = *param1,* high word = *param2;* passed to NewControl
-- *theControlHandle* --	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
-- *retValue* --	**Long**—Undefined
	← **SP**

Dispose routine

The Control Manager's DisposeControl routine sends the message `dispCtl` to the control definition procedure, telling it to perform any additional actions required when disposing of the control. If the control definition procedure returns 0 for *retValue,* the control will be erased and removed from the control list, and its record will be deallocated. The predefined standard controls always return 0.

If a custom control returns a nonzero *retValue,* the definition procedure has a chance to abort the disposal.

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
ctlMessage	**Word**—`dispCtl` message
-- *ctlParam* --	**Long**—Undefined
--*theControlHandle* --	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
-- *retValue* --	**Long**—0 to continue disposal, nonzero to abort disposal
	← **SP**

Position routine

When dragging a control's indicator, TrackControl calls the control definition procedure with the message `posCtl` to reposition the indicator and update the control's setting accordingly. The value of *ctlParam* is a point giving the vertical and horizontal offset, in pixels, by which the indicator is to be moved relative to its current position. (For example, this is often the offset between the points where the user pressed and released the mouse button while dragging the indicator.) The vertical offset is given in the low-order word of *ctlParam* and the horizontal offset in the high-order word. The definition procedure should calculate the control's new setting based on the given offset, update the *ctlValue* field in the control record, and redraw the control within its window to reflect the new setting.

❖ *Note:* The Control Manager routines SetCtlValue and SetCtlParams do not call the control definition procedure with this message; instead, they pass the `newValue` and `setParams` messages respectively. See the sections "New Value Routine" and "Set Parameters Routine" in this chapter.

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
ctlMessage	**Word**—`posCtl` message
-- *ctlParam* --	**Long**—High word is horizontal offset, low word is vertical offset
--*theControlHandle* --	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
-- *retValue* --	**Long**—0 = default reposition, nonzero = application did reposition
	← **SP**

❖ *Note:* If you're using the default reposition, your control record must look like a scroll bar control record. The Control Manager repositions the indicator as if it was a thumb on a scroll bar.

Thumb routine

Before the Control Manager begins to drag a control's indicator, it calls the control's definition procedure with the message `thumbCtl`. The control definition procedure should respond by calculating the limit rectangle, slop rectangle, axis constraint, and outline pattern to use for dragging the control's indicator. See the DragRect routine in this chapter for more information about these parameters. The parameters as they are defined in the limit block (shown in Figure 4-22) are passed to that routine.

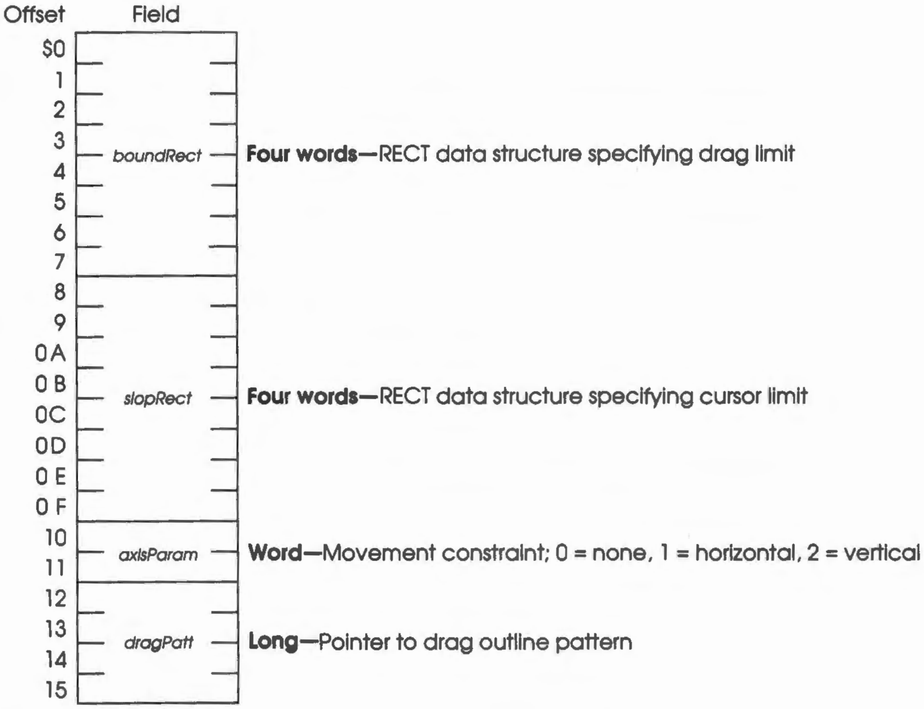
Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
ctlMessage	**Word**—`thumbCtl` message
-- *ctlParam* --	**Long**—POINTER to parameter block for dragging an indicator
--*theControlHandle* --	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
-- *retValue* --	**Long**—0 = default reposition, nonzero = application did reposition
	← **SP**

The *ctlParam* parameter is a pointer to the limit-block data structure, which is shown in Figure 4-22.



Offset	Field	
$0–7	*boundRect*	**Four words**—RECT data structure specifying drag limit
8–0F	*slopRect*	**Four words**—RECT data structure specifying cursor limit
10–11	*axisParam*	**Word**—Movement constraint; 0 = none, 1 = horizontal, 2 = vertical
12–15	*dragPatt*	**Long**—Pointer to drag outline pattern

Figure 4-22
Limit-block data

If the definition procedure returns 0, the default parameters shown in Figure 4-23 are used.

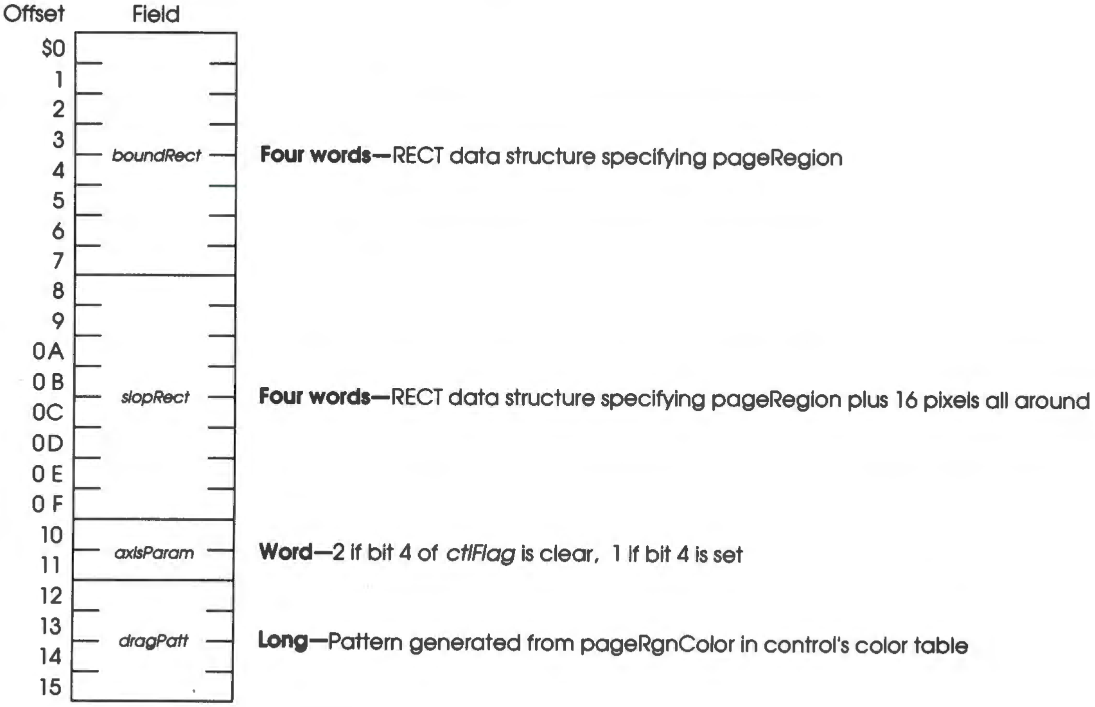

Offset
Field
$0
1
2
3
4
5
6
7
boundRect
Four words—RECT data structure specifying pageRegion
8
9
0A
0B
0C
0D
0E
0F
slopRect
Four words—RECT data structure specifying pageRegion plus 16 pixels all around
10
11
axisParam
Word—2 if bit 4 of *ctlFlag* is clear, 1 if bit 4 is set
12
13
14
15
dragPatt
Long—Pattern generated from pageRgnColor in control's color table

Figure 4-23
Default limit-block values

Drag routine

The message `dragCtl` asks the control definition procedure to drag the control or its indicator around the screen to follow the mouse until the user releases the mouse button. The *ctlParam* parameter specifies 0 to drag the entire control or a nonzero value as the part code of the part to drag.

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
ctlMessage	**Word**—`dragCtl` message
-- *ctlParam* --	**Long**—(Low-order word only) Part code to drag, 0 = drag entire control
--*theControlHandle* --	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
-- *retValue* --	**Long**—0 to use default dragging, nonzero if application did dragging
	← **SP**

The control definition procedure does not have to implement any form of custom dragging; if the procedure returns a *retValue* of 0, the Control Manager uses its own default method of dragging (calling DragControl to drag the control or DragRect to drag its indicator). Conversely, if the control definition procedure chooses to do its own custom dragging, it should signal the Control Manager not to use the default method by returning a nonzero *retValue.*

Track routine

You can design a control that has its action procedure in the control definition procedure. To do this, pass –1 for the *actionProc* parameter to the TrackControl routine and set the control's *ctlAction* field to –1. TrackControl responds by calling the control definition procedure with the `autoTrack` message. The definition procedure should respond like an action procedure, as discussed in the section "TrackControl" in this chapter.

The *ctlParam* parameter contains the part code, which defines in which part of the control the cursor was when the mouse button was pressed.

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
ctlMessage	**Word**—`autoTrack` message
-- *ctlParam* --	**Long**—(Low-order word only) Part code, 0 if not currently in part
-- *theControlHandle* --	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
-- *retValue* --	**Long**—Undefined
	← **SP**

New value routine

The Control Manager calls the control's definition procedure with the message `newValue` any time a control's value changes. First, the Control Manager stores the new value in the *ctlValue* field of the control's record. The definition should compute any new parameters affected by the change, such as a new thumb position for scroll bars, and then redraw the control (if it is visible). The definition procedure can assume that the control is already drawn in the window, so, in the case of scroll bars, only the thumb has to be erased and redrawn. Actually, the definition procedure for standard scroll bars erases only the part of the thumb that uncovered the page region, rather than the entire thumb.

In Version 1.2 and later of the Control Manager, *ctlParam* contains the previous and new values.

❖ *Note:* The *ctlParam* parameter is undefined in Control Manager Version 1.1 and earlier.

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
ctlMessage	**Word**—`newValue` message
-- *ctlParam* --	**Long**—High-order word is previous value, low-order word is new value
--*theControlHandle* --	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
-- *retValue* --	**Long**—Undefined
	← **SP**

Set parameters routine

The Control Manager calls the control's definition procedure with the message `setParams` any time a control's additional parameters change. The additional parameters are defined by the control. The values could be anything, even a pointer to more parameters. The definition should perform the actions the new parameters cause, including redrawing the control if necessary. The definition procedure can assume that the control is already drawn in the window, in contrast to the situation in which new parameters are sent with the message `initCtl` (see the section "Initialize Routine" in this chapter).

The only predefined control that uses additional parameters is the scroll bar. The low-order word is the view value, and the high-order word is the data size. Simple buttons, check boxes, and radio buttons do nothing with additional parameters. The standard scroll bar definiton procedure stores the values in the *ctlData* field of the control's record, computes a new thumb, and draws the new thumb in the scroll bar (if the scroll bar is visible).

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
ctlMessage	**Word**—`setParams` message
-- *ctlParam* --	**Long**—New parameters
--*theControlHandle* --	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
-- *retValue* --	**Long**—Undefined
	← **SP**

Move routine

The Control Manager calls the control's definition procedure with the message `moveCtl` from the MoveControl routine. The Control Manager first hides the control with HideControl, if the control was visible, and moves the control's enclosing rectangle (as defined by the *ctlRect* field of the control). The definition procedure should compute any other parameters necessary and return.

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
ctlMessage	**Word**—`moveCtl` message
-- *ctlParam* --	**Long**—High-order word = X-axis change, low-order = Y-axis change
--*theControlHandle* --	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
-- *retValue* --	**Long**—Undefined
	← **SP**

For example, the standard definition procedure for scroll bars also moves the *thumb* and *pageRegion* fields in the control record. Upon return from the definition procedure, the Control Manager makes a ShowControl call if the control was visible on entry in order to draw the control at its new position. The definition procedure should not redraw the control here, but should do everything necessary to ensure that the control will be drawn properly at its new position.

Record size routine

The Control Manager calls the control definition procedure with the message `recCtl` from the NewControl routine before it allocates memory for the control record. NewControl then allocates the number of bytes returned in *retValue* for the control's record.

Important
The *theControlHandle* parameter as passed to the definition procedure is not valid in this case. Because the control's record has not been allocated, no access to the record should be performed during the record size call. After the record has been allocated and initialized by the Control Manager, the definition procedure will be called again with the message `initCtl`. See the section "Initialize Routine" in this chapter.

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
ctlMessage	**Word**—`recCtl` message
-- *ctlParam* --	**Long**—Undefined
--*theControlHandle* --	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
-- *retValue* --	**Long**—Number of bytes needed for control record
	← **SP**

If your control needs only the standard control record, for example, button, check box, and radio button control records, return the size of the standard record (decimal 40). If your control needs additional data fields, for example, for a scroll bar record, return the size of the standard record, plus the additional size. You should never return a number less than the number of bytes in a standard record.

$0110 CtlBootInit

Initializes the Control Manager; called only by the Tool Locator.

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 Call must not be made by an application.

$0210 CtlStartUp

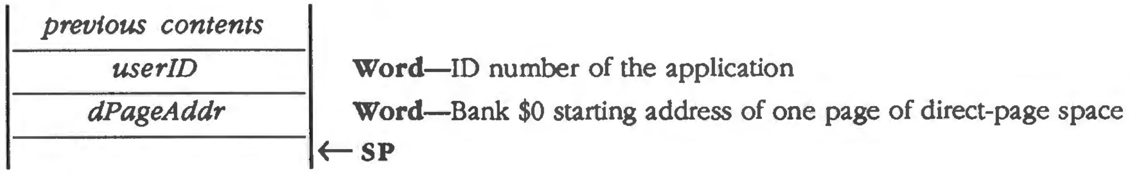
Starts up the Control Manager for use by an application.

Important

Your application must make this call before it makes any other Control Manager calls.

Parameters

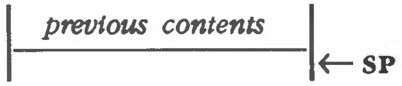
Stack before call



Word—ID number of the application

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

Stack after call



Errors

$1001 `wmNotStartedUp` Window Manager not initialized

C

```
extern pascal void CtlStartUp(userID,dPageAddr)

Word     userID;

Word     dPageAddr;
```

$0310 CtlShutDown

Shuts down the Control Manager.

Important

If your application has started up the Control Manager, the application must make this call before it quits.

CtlShutDown does not dispose of controls, because the Window Manager disposes of all controls in a window. Therefore, the Window Manager must be shut down before the Control Manager.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

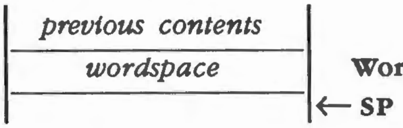
```
extern pascal void CtlShutDown()
```

$0410 CtlVersion

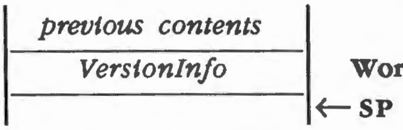
Returns the version number of the Control Manager.

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal Word CtlVersion()
```

$0510 CtlReset

Resets the Control Manager; called 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 Call must not be made by an application.

$0610 CtlStatus

Indicates whether the Control Manager is active.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
activeFlag	**Word**—BOOLEAN; TRUE if Control Manager active; FALSE if inactive
	← **SP**

Errors None

C

```
extern pascal Boolean CtlStatus()
```

$1210 CtlNewRes

Reinitializes resolution and mode. Call CtlNewRes after you have changed the video mode.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void CtlNewRes()
```

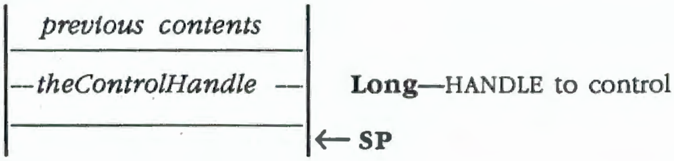
$0A10 DisposeControl

Deletes a specified control from its window's control list and releases the memory occupied by the control record and any data structures associated with the control.

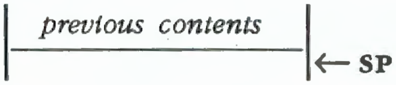
❖ *Note:* DisposeControl does not erase the control from the screen. If necessary, hide the control by calling HideControl before calling DisposeControl.

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal void DisposeControl(theControlHandle)
CtlRecHndl     theControlHandle;
```

$1710 DragControl

Pulls a dotted outline of the control around the screen, following the movements of the mouse until the button is released. When the mouse button is released, DragControl calls the MoveControl routine to move the control to the appropriate location. Called when the mouse button is pressed while the cursor is in a specified control.

❖ *Note:* Before beginning to follow the mouse, DragControl calls the control definition procedure to allow the application to perform its own custom dragging if it chooses. If the definition procedure doesn't choose to perform any custom dragging, DragControl uses the default dragging method.

The *startX, startY, limitRectPtr,* and *slopRectPtr* parameters have the same meaning as in the DragRect routine (see the section "DragRect" in this chapter). The *axis* parameter has the same meaning as bits 1–0 in the *dragFlag* parameter of the DragRect routine.

Parameters

Stack before call

previous contents	
startX	**Word**—X coordinate of starting point in local coordinates
startY	**Word**—Y coordinate of starting point in local coordinates
-- *limitRectPtr* --	**Long**—POINTER to bounds RECT data structure
-- *slopRectPtr* --	**Long**—POINTER to slop RECT data structure
axis	**Word**—Movement constraint: 0 = none, 1 = horizontal, 2 = vertical
--*theControlHandle* --	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void DragControl(startX,startY,limitRectPtr,slopRectPtr,
axis,theControlHandle)

Integer     startX;

Integer     startY;

Rect *limitRectPtr;

Rect *slopRectPtr;

Word     axis;

CtlRecHndl     theControlHandle;
```

You can also use the following alternate form of the call:

```
extern pascal void DragControl(startPoint,limitRectPtr,slopRectPtr,
axis,theControlHandle)

Point     startPoint;

Rect *limitRectPtr;

Rect *slopRectPtr;

Word     axis;

CtlRecHndl     theControlHandle;
```

$1D10 DragRect

Pulls a dotted outline of a specified rectangle around the screen, following the mouse movements until the mouse button is released.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *actionProcPtr* --	**Long**—POINTER to routine; NIL for default
-- *dragPatternPtr* --	**Long**—POINTER to pattern to use for drag outline
startX	**Word**—X coordinate of starting point in local coordinates
startY	**Word**—Y coordinate of starting point in local coordinates
-- *dragRectPtr* --	**Long**—POINTER to RECT data structure of rectangle to be dragged
-- *limitRectPtr* --	**Long**—POINTER to bounds RECT data structure
-- *slopRectPtr* --	**Long**—POINTER to slop RECT data structure
dragFlag	**Word**—Bit flag that customizes DragRect (see Figure 4-25)
	← **SP**

Stack after call

previous contents	
-- *moveDelta* --	**Long**—High word = amount X changed, low word = amount Y changed
	← **SP**

Errors

$1001	`wmNotStartedUp`	Window Manager not initialized

C

```
extern pascal LongWord DragRect(actionProcPtr,dragPatternPtr,
startX,startY,dragRectPtr,limitRectPtr,slopRectPtr,dragFlag)

VoidProcPtr     actionProcPtr;

Pattern     dragPatternPtr;

Integer     startX;

Integer     startY;

Rect *dragRectPtr;

Rect *limitRectPtr;

Rect *slopRectPtr;

Word     dragFlag;
```

You can also use the following alternate form of the call:

```
extern pascal LongWord DragRect(actionProcPtr,dragPatternPtr,
start,dragRectPtr,limitRectPtr,slopRectPtr,dragFlag)

VoidProcPtr     actionProcPtr;

Pattern     dragPatternPtr;

Point     start;

Rect *dragRectPtr;

Rect *limitRectPtr;

Rect *slopRectPtr;

Word     dragFlag;
```

(continued)

More about DragRect parameters

The *actionProcPtr* parameter is a pointer to a procedure that defines some action to be performed repeatedly for as long as the user holds down the mouse button; the procedure should have no parameters. If *actionProcPtr* is NIL, DragRect simply retains control until the mouse button is released.

The *dragPatternPtr* points to a QuickDraw II type of pattern. See Chapter 16, "QuickDraw II," in Volume 2 for more information.

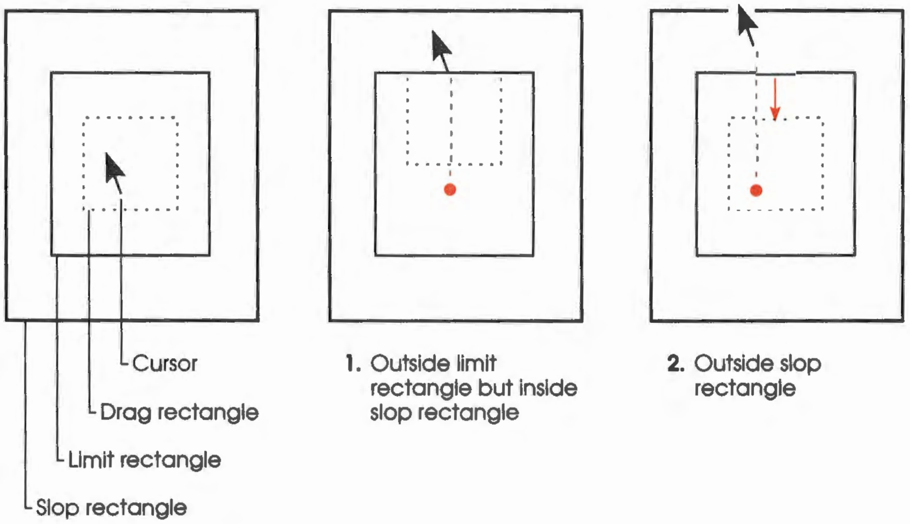
The *startX* and *startY* parameters are assumed to be the point where the cursor was when the mouse button was originally pressed in local coordinates. "Local coordinates," in this case, means coordinates that are local to the current GrafPort. However, if you want to drag an object anywhere on the screen, then the current GrafPort should be the size of the screen. When the GrafPort is the size of the screen, local and global coordinates are the same.

The RECT data structures pointed to by *limitRectPtr* and *slopRectPtr* are also in local coordinates (that is, local as explained for *startX* and *startY*). To understand these parameters, you must first understand the concept of **offset point.** The offset point is initially the point whose vertical and horizontal offsets from the top-left corner of the region's enclosing rectangle are the same as those of *startX* and *startY*. The offset point follows the mouse location, except that DragRect never moves the offset point outside of the **limit rectangle;** this limits the travel of the region's outline (but not the movements of the mouse). The **slop rectangle** (defined by the RECT data structure pointed to by *slopRectPtr*), which should completely enclose the limit rectangle, allows the user some margin for error (that is, the user doesn't have to be extremely precise) when moving the mouse. DragRect's behavior while tracking the mouse depends on the location of the mouse with respect to these two rectangles, as follows:

- When the mouse is inside the limit rectangle, the region's outline follows it normally. If the mouse button is released while the mouse is there, the region should be moved to the mouse location.
- When the mouse is outside the limit rectangle but inside the slop rectangle, DragRect "pins" the offset point to the edge of the limit rectangle. If the mouse button is released while the mouse is there, the region should be moved to this pinned location.
- When the mouse is outside the slop rectangle, the outline disappears from the screen, but the DragRect routine continues to follow the mouse; if the mouse moves back into the slop rectangle, the outline reappears. If the mouse button is released while the mouse is outside the slop rectangle, the region should not be moved from its original position.

Figure 4-24 shows what happens when

- The user moves the mouse past the boundaries of the limit rectangle but inside the boundaries of the slop rectangle
- The user moves the mouse outside the the boundaries of both the limit and the slop rectangles



1. Outside limit rectangle but inside slop rectangle

2. Outside slop rectangle

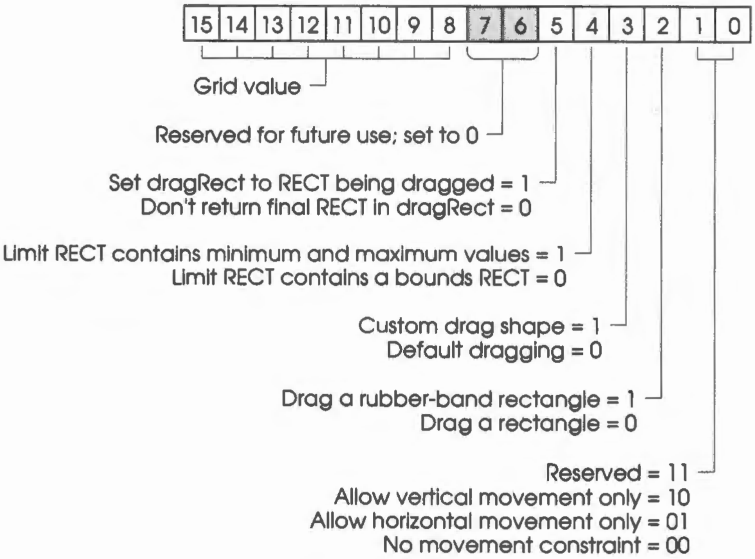
Figure 4-24
DragRect examples

The left diagram in Figure 4-24 shows the starting position. As the cursor is moved, an outline of the window is dragged with it. The outline will seem to be glued to the cursor at the offset point. However, if the cursor moves outside the limit rectangle, the outline will be left behind, as shown in the center diagram. As the cursor is moved outside the limit rectangle, but within the slop rectangle, the outline will get as close to the cursor as possible without letting the offset point leave the limit rectangle. Finally, if the cursor moves outside the slop rectangle, the outline will snap back to its starting position. If the cursor moves back into the slop rectangle, the outline will snap out to get as close to the cursor as it can.

If the mouse button is released within the slop rectangle, the high-order word of the value returned by DragRect contains the vertical coordinate of the ending mouse location minus that of *startX* and *startY*, and the low-order word contains the difference between the horizontal coordinates. If the mouse button is released outside the slop rectangle, both words are 0.

(continued)

The *dragFlag* parameter allows you to customize the DragRect routine so that it performs in different ways. The meanings of the bits in the flag are summarized in Figure 4-25, and each bit is discussed in greater detail after the figure.



15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Grid value
Reserved for future use; set to 0
Set dragRect to RECT being dragged = 1
Don't return final RECT in dragRect = 0
Limit RECT contains minimum and maximum values = 1
Limit RECT contains a bounds RECT = 0
Custom drag shape = 1
Default dragging = 0
Drag a rubber-band rectangle = 1
Drag a rectangle = 0
Reserved = 11
Allow vertical movement only = 10
Allow horizontal movement only = 01
No movement constraint = 00

Figure 4-25
DragRect routine *dragFlag* parameter

Bits 15–8 specify on what vertical columns the rectangle's horizontal position is to be bound. Values $00xx and $01xx both specify single horizontal movement; other values passsed must be a power of 2. For example, if the value $04xx is passed, the smallest amount of movement possible would be 4 pixels to the left or right of the starting position. From there, movements to the left or right of 8, 12, 16, or 20 pixels, and so on for other multiples of 4 would be possible.

The feature can be used to move dithered colors, which would otherwise be position dependent, and to speed up pixel copying by forcing a pattern to stay on byte or word boundaries. See Chapter 16, "QuickDraw II," in Volume 2 for more information about dithered colors.

Bit 5 can be set to 1 to make the DragRect routine store coordinates of the dragged rectangle in the RECT data structure pointed to by the *dragRectPtr* parameter. That RECT will then be updated whenever DragRect moves the rectangle. The RECT can be used as the final RECT when the DragRect routine returns, or by a custom draw routine for drawing the current RECT. See the description of bit 3.

Bit 4 can be set to 1 to pass the maximum and minimum amounts the cursor is allowed to move from its starting position (the delta values). This bit can be useful when you'd like to pin the cursor to a RECT other than the drag RECT. The values are signed and passed in the RECT data structure for the limit rectangle as follows:

- Top of limit rectangle = minimum Y delta (0 or less)
- Left of limit rectangle = minimum X delta (0 or less)
- Bottom of limit rectangle = maximum Y delta (0 or greater)
- Right of limit rectangle = maximum X delta (0 or greater)

Bit 3 can be set to 1 to use a custom draw routine for drawing the drag shape. In this case, the *actionProcPtr* parameter points to the custom routine that will be called to draw the drag outline. The custom routine will not be called in the same way or at the same time intervals as an action routine, but will be called whenever the DragRect routine determines it should be. The custom routine picks up the change in the cursor's position from the starting position passed in the *startX* and *startY* parameters.

Bit 2 defines the shape being dragged. If the bit is set to 1, the shape is a rectangle with a constant upper-left corner and a changing lower-right corner. The movement delta is added to the bottom and right sides of the drag rectangle to compute the drag rectangle. If bit 2 is 0, the shape is a rectangle with a constant width and height. The movement delta is added to all four sides of the drag rectangle to compute the drag rectangle.

Bits 1–0 allow you to restrict the region's motion to only one axis. They have one of the values shown in Table 4-6.

Table 4-6
Movement constraint values

Value	Name	Description
0	`noConstraint`	No constraint on movement
1	`hAxisOnly`	Movement constrained to horizontal axis only
2	`vAxisOnly`	Movement constrained to vertical axis only

If an axis constraint is in effect, the outline will follow the mouse's movements along the specified axis only, ignoring motion along the other axis.

$1010 DrawControls

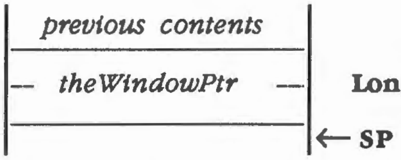
Draws all controls currently visible in a specified window. The controls are drawn in reverse order of creation; thus, in case of overlap, the controls created first appear frontmost in the window.

Important

Window Manager routines such as SelectWindow, ShowWindow, and BringToFront do not call DrawControls to display the window's controls. They just add the areas of the window that had not been visible to the window's update region, thus generating an update event. When your application receives an update event for a window that contains controls, the application should always call DrawControls explicitly between the BeginUpdate and EndUpdate calls.

Parameters

Stack before call



Long—POINTER to window whose controls are to be drawn

Stack after call

previous contents
← SP

Errors

None

C

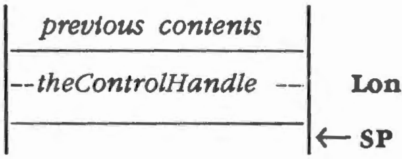
```
extern pascal void DrawControls(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$2510 DrawOneCtl

Draws a specified control. If you want to draw all of the controls in the control list, see the section "DrawControls" in this chapter.

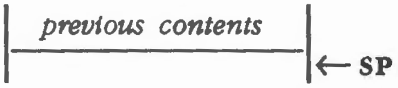
Parameters

Stack before call



Long—HANDLE to control to be drawn

Stack after call



Errors

None

C

```
extern pascal void DrawOneCtl(theControlHandle)

CtlRecHndl     theControlHandle;
```

$2410 EraseControl

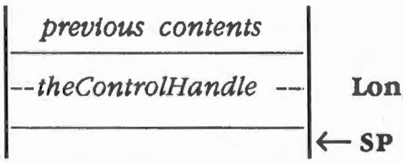
Makes a specified control invisible by filling the region the control occupies with the background pattern of the window's GrafPort. Unlike the HideControl routine, EraseControl does not add the control's enclosing rectangle to the window's update region.

The specified control's *ctlFlag* field is set to make the control invisible. If you need to make the control reappear, use the ShowControl routine.

If the control is already invisible, EraseControl has no effect.

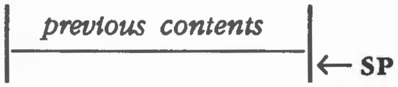
Parameters

Stack before call



Long—HANDLE to control to be erased

Stack after call



Errors

None

C

```
extern pascal void EraseControl(theControlHandle)

CtlRecHndl     theControlHandle;
```

$1310 FindControl

Tells in which of a specified window's controls, if any, the cursor was in when the user pressed the mouse button, as follows:

- If the mouse button was pressed while the cursor was in a visible, active control, FindControl sets the *foundCtlPtr* parameter to the control handle and returns a part code identifying the part of the control in which the cursor was located when the mouse button was pressed.
- If the mouse button was pressed while the cursor was in an invisible or inactive control or not in any control, FindControl returns 0 as *foundPart,* and *foundCtlPtr* is undefined.

❖ *Note:* FindControl also returns 0 as *foundPart,* and *foundCtlPtr* is undefined if the window is invisible or doesn't contain the specified point. However, the Window Manager shouldn't return this window in the first place, so this situation shouldn't arise.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *foundCtlPtr* --	**Long**—POINTER to address where handle of found control will be stored
pointX	**Word**—X coordinate to check, in global coordinates
pointY	**Word**—Y coordinate to check, in global coordinates
-- *theWindowPtr* --	**Long**—POINTER to window to check
	← **SP**

Stack after call

previous contents	
foundPart	**Word**—Part code of found part of control
	← **SP**

Errors

None

C

```
extern pascal Word FindControl(foundCtlPtr,pointX,pointY,theWindowPtr)
CtlRecHndl  *foundCtlPtr;
Integer     pointX;
Integer     pointY;
GrafPortPtr     theWindowPtr;
```

You can also use the following alternate form of the call:

```
extern pascal Word FindControl(foundCtlPtr,foundPoint,theWindowPtr)
CtlRecHndl  *foundCtlPtr;
Point     foundPoint;
GrafPortPtr     theWindowPtr;
```

About FindControl and Window Manager FindWindow routine

When the Window Manager routine FindWindow reports that the mouse button was pressed in the content region of a window and the window contains controls, your application should call FindControl with *theWindowPtr* equal to the window pointer and *pointX/pointY* equal to the point (in global coordinates) where the cursor was when the mouse button was pressed.

$2110 GetCtlAction

Returns the current value of a specified control's *ctlAction* field.

Parameters

Stack before call

previous contents	
— *longspace* —	**Long**—Space for result
—*theControlHandle* —	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
— *ctlActionValue* —	**Long**—Value in control's *ctlAction* field
	← **SP**

Errors

None

C

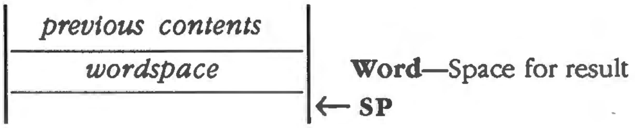
```
extern pascal LongProcPtr GetCtlAction(theControlHandle)
CtlRecHndl     theControlHandle;
```

$1F10 GetCtlDpage

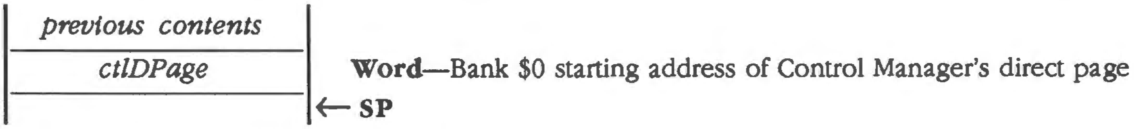
Returns the value of the Control Manager's direct page. This call is normally made only by the Dialog Manager.

Parameters

Stack before call



Stack after call


previous contents
ctlDPage
Word—Bank $0 starting address of Control Manager's direct page
← **SP**

Errors

None

C

```
extern pascal Word GetCtlDPage()
```

$1C10 GetCtlParams

Returns a specified control's additional parameter settings.

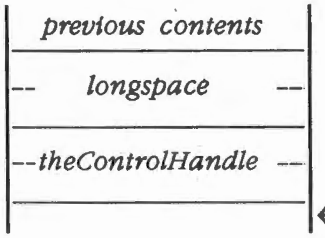
Scroll bars use *param1* as the scroll bar's view and *param2* as the data size.

Simple buttons, check boxes, radio buttons, and grow boxes do not use *param1* or *param2*.

❖ *Note:* Custom controls might or might not support this feature, depending upon the custom control.

Parameters

Stack before call

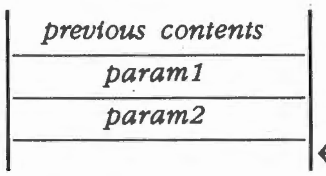


Long—Space for result

Long—HANDLE to control

← **SP**

Stack after call



Word—Value of additional control parameter

Word—Value of additional control parameter

← **SP**

Errors

None

C

```
extern pascal LongWord GetCtlParams(theControlHandle)
CtlRecHndl     theControlHandle;
```

$2310 GetCtlRefCon

Returns the current value of a specified control's *ctlRefCon* field.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
--*theControlHandle* --	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
-- *ctlRefConValue* --	**Long**—Value in control's *ctlRefCon* field
	← **SP**

Errors

None

C

```
extern pascal LongWord GetCtlRefCon(theControlHandle)
CtlRecHndl     theControlHandle;
```

$0D10 GetCtlTitle

Returns the value in a specified control's *ctlData* field. For controls with titles, the value is the pointer to the control's Pascal-type title string. For scroll bars, the *ctlData* field contains the view and data sizes. For custom controls, the *ctlData* field is defined by the control.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
--*theControlHandle* --	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
-- *ctlTitlePtr* --	**Long**—POINTER to control's *ctlData* field
	← **SP**

Errors

None

C

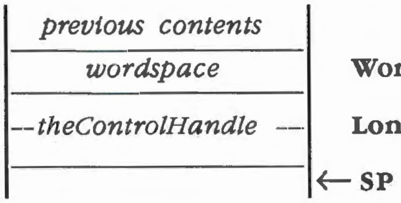
```
extern pascal Pointer GetCtlTitle(theControlHandle)

CtlRecHndl     theControlHandle;
```

$1A10 GetCtlValue

Returns a specified control's current *ctlValue* field.

Parameters

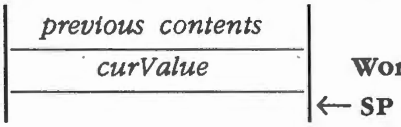
Stack before call



Word—Space for result

Long—HANDLE to control

Stack after call



Word—Control's current value

Errors

None

C

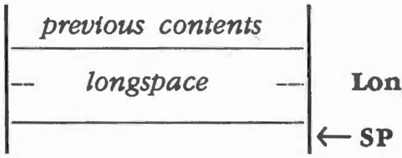
```
extern pascal Word GetCtlValue(theControlHandle)
CtlRecHndl     theControlHandle;
```

$1E10 GrowSize

Returns the height and width of the size box control, using the Control Manager's current icon font. You can use this value, for example, to help you compute the size of the scroll bar.

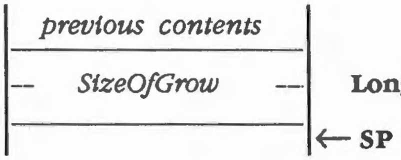
Parameters

Stack before call



Long—Space for result

Stack after call



Long—High-order word is width, low-order word is height

Errors

None

C

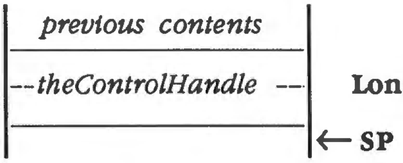
```
extern pascal LongWord GrowSize()
```

$0E10 HideControl

Makes a specified control invisible by filling the region the control occupies with the background pattern of the window's GrafPort. The routine also adds the control's enclosing rectangle to the window's update region, so that anything else previously obscured by the control reappears on the screen. If the control is already invisible, HideControl has no effect.

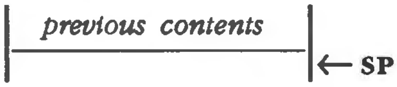
Parameters

Stack before call



Long—HANDLE to control to be hidden

Stack after call



Errors

None

C

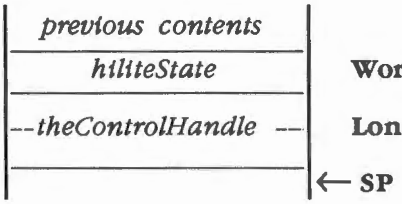
```
extern pascal void HideControl(theControlHandle)

CtlRecHndl     theControlHandle;
```

$1110 HiliteControl

Changes the way a specified control is highlighted. HiliteControl calls the control definition routine to redraw the control with its new highlighting.

Parameters

Stack before call



Word—Type of highlighting: 0 = none, 1–253 = part code, 255 = inactive

Long—HANDLE to control to be highlighted

Stack after call

previous contents
← SP

Errors

None

C

```
extern pascal void HiliteControl(hiliteState,theControlHandle)

Word      hiliteState;

CtlRecHndl     theControlHandle;
```

More about highlighting

The *hiliteState* parameter has one of the following values:

1. The value 0 (`noHilite`) indicates no highlighting and that the control is active. If any part of the control is highlighted, the highlighting is removed. If the control is inactive, it's changed to active and redrawn.
2. A value between 1 and 253 is interpreted as a part code designating the part of the (active) control to be highlighted. See Table 4-2 in the section "Part Codes" in this chapter.
3. The value 255 (`inactiveHilite`) means that the control is to be made inactive and redrawn accordingly.

❖ *Note:* Do not use the value 254; this value is reserved for future use.

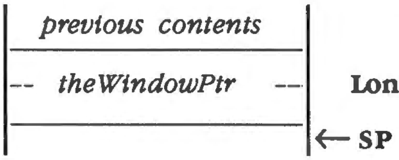
$0B10 KillControls

Disposes of all controls associated with a specified window by calling the DisposeControl routine for each control in the window's control list.

❖ *Note:* The Window Manager routine CloseWindow calls KillControls to dispose of all controls associated with the specified window.

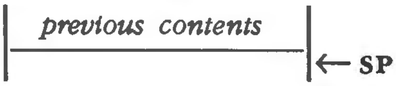
Parameters

Stack before call



Long—POINTER to window whose controls are to be disposed of

Stack after call



Errors

None

C

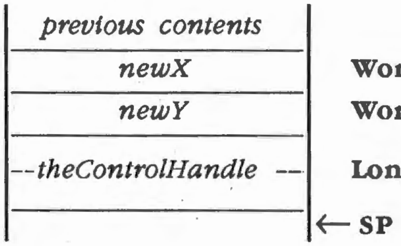
```
extern pascal void KillControls(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$1610 MoveControl

Moves a specified control to a new location within its window. The upper-left corner of the control's enclosing rectangle is moved to the horizontal and vertical coordinates *newX* and *newY* (given in the local coordinates of the control's window); the bottom-right corner is adjusted accordingly to keep the size of the rectangle the same as before. If the control is currently visible, it's hidden and then redrawn at its new location.

Parameters

Stack before call

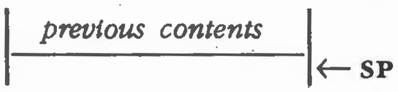


Word—New X origin of control, in local coordinates

Word—New Y origin of control, in local coordinates

Long—HANDLE to control

Stack after call



Errors

None

C

```
extern pascal void MoveControl(newX,newY,theControlHandle)

Integer     newX;

Integer     newY;

CtlRecHndl     theControlHandle;
```

You can also use the following alternate form of the call:

```
extern pascal void MoveControl(newPos,theControlHandle)

Point     newPos;

CtlRecHndl     theControlHandle;
```

$0910 NewControl

Creates a control, adds it to the beginning of a specified control list, and returns a handle to the new control. The field that determines highlighting is set to 0 (no highlighting). NewControl does not draw the control.

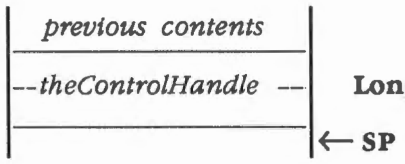
❖ *Note:* The control definition procedure may perform additional initialization, including changing any of the control record fields. The scroll bar is the only standard control for which additional initialization is performed; its control definition procedure computes the thumb and the page region from *boundsRectPtr* and *flag*.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *theWindowPtr* --	**Long**—POINTER to window owner
-- *boundsRectPtr* --	**Long**—POINTER to RECT data structure defining enclosing rectangle
-- *titlePtr* --	**Long**—POINTER to title string (*ctlData*)
flag	**Word**—Bit flag (see Figure 4-26)
value	**Word**—Control's starting value
param1	**Word**—Additional parameter (view size for scroll bars)
param2	**Word**—Additional parameter (data size for scroll bars)
-- *defProcPtr* --	**Long**—POINTER to definition procedure, or standard control value
-- *refCon* --	**Long**—Reserved for application use
-- *colorTablePtr* --	**Long**—POINTER to control's color table; NIL for default color table
	← **SP**

Stack after call



Long—HANDLE to control; NIL = error

Errors The *controlHandle* parameter will be NIL if the control record can't be allocated.

C

```
extern pascal CtlRecHndl NewControl(theWindowPtr,boundsRectPtr,titlePtr,
flag,value,param1,param2,defProcPtr,refCon,colorTablePtr)

GrafPortPtr     theWindowPtr;

Rect *boundsRectPtr;

Pointer     titlePtr;

Word     flag;

Word     value;

Word     param1;

Word     param2;

LongProcPtr     defProcPtr;

Longint     refCon;

CtlColorTablePtr     colorTablePtr;
```

More about NewControl parameters

The values passed as parameters are stored in the corresponding control record fields. The pointer in *theWindowPtr* parameter points to the window to which the new control will belong. All coordinates pertaining to the control are interpreted using this window's local coordinate system.

The RECT data structure pointed to by *boundsRectPtr* specifies the rectangle that encloses the control in the window's local coordinates and thus determines the control's size and location.

The *titlePtr* parameter points to the control's title, if any (if there is none, just pass a NIL pointer as the title). Be sure the title will fit in the control's enclosing rectangle; if the title doesn't fit, it may not be completely erased by the HideControl routine.

The *flag* parameter is a bit flag that further defines the control. The bit values and their functions vary according to the type of control being displayed, as shown in Figure 4-26.

(continued)

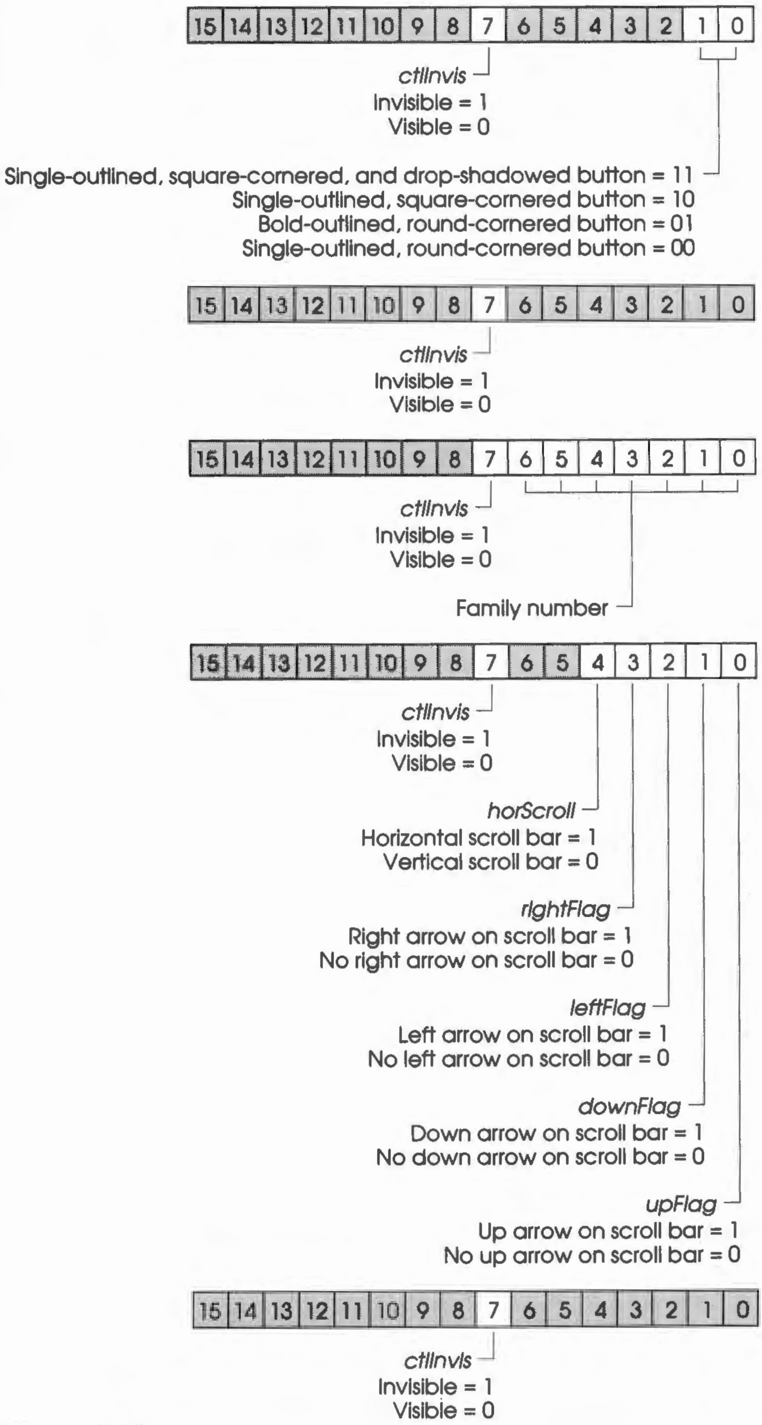

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
ctlInvis
Invisible = 1
Visible = 0
Single-outlined, square-cornered, and drop-shadowed button = 11
Single-outlined, square-cornered button = 10
Bold-outlined, round-cornered button = 01
Single-outlined, round-cornered button = 00
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
ctlInvis
Invisible = 1
Visible = 0
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
ctlInvis
Invisible = 1
Visible = 0
Family number
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
ctlInvis
Invisible = 1
Visible = 0
horScroll
Horizontal scroll bar = 1
Vertical scroll bar = 0
rightFlag
Right arrow on scroll bar = 1
No right arrow on scroll bar = 0
leftFlag
Left arrow on scroll bar = 1
No left arrow on scroll bar = 0
downFlag
Down arrow on scroll bar = 1
No down arrow on scroll bar = 0
upFlag
Up arrow on scroll bar = 1
No up arrow on scroll bar = 0
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
ctlInvis
Invisible = 1
Visible = 0

Figure 4-26
Control Manager flag bits

The *value* field gives the control's initial settings.

The *param1* and *param2* fields contain additional values that are defined by the control's definition procedure. For standard scroll bars, *param1* is the size of the view and *param2* is the total data size. The standard scroll bar definition procedure will store the value of *param1* in the *ctlData* field and *param2* in *ctlData* + 2 field. Any values can be passed when creating simple buttons, check boxes, radio buttons, or size boxes.

The *defProcPtr* points to the control's definition procedure. The definition procedures for custom control types are discussed in the section "Defining Your Own Controls" in this chapter. The values for the standard control types are as shown in Table 4-7.

Table 4-7
Standard control type values

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

The *refCon* field is reserved for application use.

The *colorTablePtr* points to the color table used to draw the control. NIL can be passed to use a default color table; other values access a color table defined by the control's definition procedure.

$2010 SetCtlAction

Sets a specified control's *ctlAction* field to a pointer for a custom control action procedure. See the section "TrackControl" in this chapter for more information.

Parameters

Stack before call

previous contents	
— *newActionPtr* —	**Long**—POINTER to custom control action procedure
—*theControlHandle* —	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

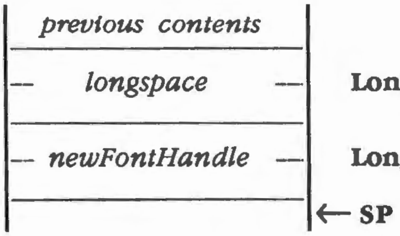
```
extern pascal void SetCtlAction(newActionPtr,theControlHandle)

LongProcPtr     newActionPtr;

CtlRecHndl     theControlHandle;
```

$1810 SetCtlIcons

Replaces the current icon font with a specified new font and returns the handle of the old font, or just returns the handle of the old font. See the section "Control Manager Icon Font" in this chapter for more information.

Parameters

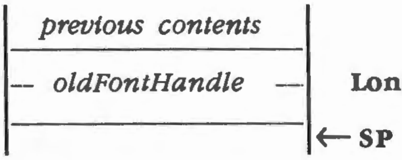
Stack before call



Long—Space for result

Long—HANDLE to new icon font; negative value to not set new font

Stack after call



Long—HANDLE to old icon font

Errors

None

C

```
extern pascal FontHndl SetCtlIcons(newFontHandle)

FontHndl      newFontHandle;
```

$1B10 SetCtlParams

Sets new parameters to the control's definition procedure, which will set the values and redraw the control if necessary.

Scroll bars use *param1* as the scroll bar's view and *param2* the data size. If –1 is passed for either *param1* or *param2,* that parameter will not be changed.

Simple buttons, check boxes, radio buttons, and size boxes do not use *param1* or *param2,* and no action is performed.

❖ *Note:* Custom controls might or might not not support this feature, depending upon the control.

Parameters

Stack before call

previous contents	
param2	**Word**—Additional control parameter; defined by control
param1	**Word**—Additional control parameter; defined by control
--*theControlHandle* --	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

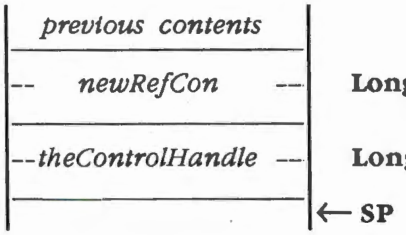
```
extern pascal void SetCtlParams(param2,param1,theControlHandle)

Word      param2;

Word      param1;

CtlRecHndl     theControlHandle;
```

$2210 SetCtlRefCon

Sets a specified control's *ctlRefCon* field to a new value. The *ctlRefCon* field is reserved for the application's use and is not changed (except by this call) by the Control Manager.

Parameters

Stack before call



Long—Value to store in the control's *ctlRefCon* field

Long—HANDLE to control

Stack after call

previous contents
← SP

Errors

None

C

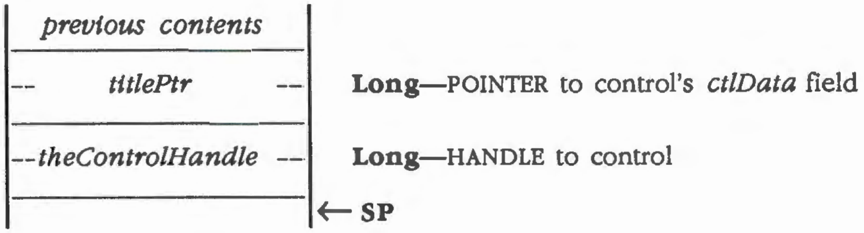
```
extern pascal void SetCtlRefCon(newRefCon,theControlHandle)

Longint     newRefCon;

CtlRecHndl     theControlHandle;
```

$0C10 SetCtlTitle

Sets a specified control's *ctlData* field to a specified title and redraws the control.

Parameters

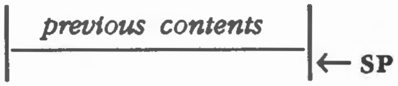
Stack before call



Long—POINTER to control's *ctlData* field

Long—HANDLE to control

Stack after call



Errors

None

C

```
void SetCtlTitle(titlePtr,theControlHandle)
Pointer titlePtr;
Handle theControlHandle;
```

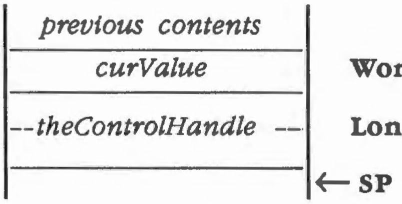
$1910 SetCtlValue

Sets a specified control's *ctlValue* field to a specified value and redraws the control to reflect the new setting. For check boxes and radio buttons, a value of 1 fills the control with the appropriate mark, and 0 clears it. For scroll bars, SetCtlValue redraws the thumb when appropriate.

If the specified value is out of range, the value is pinned to the nearest endpoint of the range, as specified by the control.

Parameters

Stack before call



previous contents

curValue **Word**—Current value of control

--*theControlHandle*-- **Long**—HANDLE to control

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

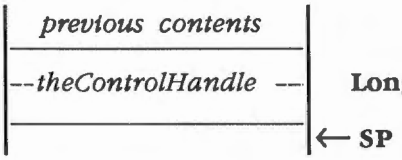
```
extern pascal void SetCtlValue(curValue,theControlHandle)

Word      curValue;

CtlRecHndl     theControlHandle;
```

$0F10 ShowControl

Makes a specified control visible. The control is drawn in its window but may be completely or partially obscured by overlapping windows or other objects. If the control is marked as visible as specified by the *ctlInvis* bit (bit 7) in the *ctlFlag*, ShowControl has no effect.

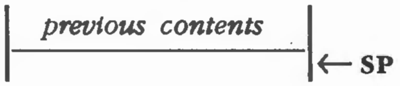
Parameters

Stack before call



Long—HANDLE to control to be shown

Stack after call



Errors

None

C

```
extern pascal void ShowControl(theControlHandle)

CtlRecHndl     theControlHandle;
```

$1410 TestControl

If a specified control is visible and active, TestControl tests which part of the control contains a specified point (using the local coordinates of the control's window); the routine returns the corresponding part code, or it returns 0 if the point is outside the control.

If the specified control is invisible or inactive, TestControl returns 0. TestControl is called by the FindControl and TrackControl routines; normally an application won't need to use TestControl.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
pointX	**Word**—X coordinate to check, in local coordinates
pointY	**Word**—Y coordinate to check, in local coordinates
--*theControlHandle* --	**Long**—HANDLE of control to be tested
	← **SP**

Stack after call

previous contents	
partCode	**Word**—Part code of the POINT; 0 if not in visible, active control
	← **SP**

Errors

None

C

```
extern pascal Word TestControl(pointX,pointY,theControlHandle)

Integer     pointX;

Integer     pointY;

CtlRecHndl      theControlHandle;
```

You can also use the following alternate form of the call:

```
extern pascal Word TestControl(testPoint,theControlHandle)

Point     testPoint;

CtlRecHndl      theControlHandle;
```

$1510 TrackControl

Follows the movements of the mouse and responds appropriately until the mouse button is released; the exact response depends on the type of control and the part of the control in which the mouse button was pressed.

If highlighting is appropriate, TrackControl performs the highlighting and then removes it before returning. When the mouse button is released, TrackControl returns with the part code if the mouse is in the same part of the control that it was originally in; otherwise, TrackControl returns 0 (in which case the application should do nothing).

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
startX	**Word**—X coordinate, in global coordinates, of starting point
startY	**Word**—Y coordinate, in global coordinates, of starting point
-- *actionProcPtr* --	**Long**—POINTER to routine; NIL, or negative (see the section "Additional Actions and the Action Procedure" in this chapter)
--*theControlHandle* --	**Long**—HANDLE to control
	← **SP**

Stack after call

previous contents	
partCode	**Word**—Selected part when button was released
	← **SP**

Errors

None

C

```
extern pascal Word TrackControl(startX,startY,actionProcPtr,theControlHandle)

Integer      startX;

Integer      startY;

LongProcPtr      actionProcPtr;

CtlRecHndl     theControlHandle;
```

You can also use the following alternate form of the call:

```
extern pascal Word TrackControl(start,actionProcPtr,theControlHandle)
Point     start;
LongProcPtr     actionProcPtr;
CtlRecHndl     theControlHandle;
```

About TrackControl and indicators

If the part code indicates that the user pressed the mouse button while the cursor was in an indicator, TrackControl drags a dotted outline of the indicator to follow the mouse. When the mouse button is released, TrackControl calls the control definition procedure to reposition the indicator. The control definition function for scroll bars responds by redrawing the thumb, calculating the control's current setting based on the new relative position of the thumb, and storing the current setting in the control record. The application must then scroll to the corresponding relative position in the document.

Additional actions and the action procedure

TrackControl may take additional actions beyond highlighting the control or dragging the indicator, depending on the value passed in the *actionProcPtr* parameter, as described below. For a custom control, what you pass will depend on how the control is defined. What to pass for the standard control types is as follows:

- The *actionProcPtr* parameter may be a pointer to an action procedure that defines some action to be performed repeatedly for as long as the user holds down the mouse button.
- If *actionProcPtr* is NIL, TrackControl performs no additional actions. This is appropriate for simple buttons, check boxes, radio buttons, and the thumb of a scroll bar.
- If *actionProcPtr* is a negative number, TrackControl checks the *ctlAction* field of the control record. No additional actions are performed if *ctlAction* is 0. If *ctlAction* is negative, the control's definition procedure is called with an `autoTrack` message. If *ctlAction* is neither 0 nor negative, it is considered a valid pointer to an action routine and is called.

(continued)

The action procedure in the control definition procedure is described in the section "Defining Your Own Controls" in this chapter. The input to the action procedure must be as follows:

partCode	**Word**—Selected part
--*theControlHandle* --	**Long**—HANDLE to control
RTL | *RTL* *RTL* |← **SP**	**3 bytes**—RTL address

In this case, TrackControl passes the control handle and the part code to the action procedure. (It passes 0 in the *partCode* parameter if the mouse has moved outside the original control part.) As an example of this type of action procedure, consider what should happen when the mouse button is pressed in a scroll arrow or paging region of a scroll bar. For these cases, your action procedure should examine the part code to determine exactly where the mouse button was pressed, scroll up or down a line or page as appropriate, and call the SetCtlValue routine to change the control's setting and redraw the thumb.

Control Manager summary

This section briefly summarizes the constants, data structures, and tool set error codes contained in the Control Manager.

Important

These definitions are provided in the appropriate interface file.

Table 4-8
Control Manager constants

Name	Value	Description
Invisible control flag value		
`ctlInVis`	$0080	Invisible mask for any type of control
Simple button control flag values		
`simpRound`	$0000	Single-outlined, round-cornered button
`simpBRound`	$0001	Bold-outlined, round-cornered button
`simpSquare`	$0002	Single-outlined, square-cornered button
`simpDropSquare`	$0003	Single-outlined, square-cornered, drop-shadowed button
Radio button control flag values		
`family`	$007F	Radio button family number
Scroll bar control flag values		
`upFlag`	$0001	Up arrow on scroll bar
`downFlag`	$0002	Down arrow on scroll bar
`leftFlag`	$0004	Left arrow on scroll bar
`rightFlag`	$0008	Right arrow on scroll bar
`horScroll`	$0010	Horizontal scroll bar
CtlProc		
`simpleProc`	$00000000	Simple button standard control
`checkProc`	$02000000	Simple button standard control
`radioProc`	$04000000	Radio button standard control
`scrollProc`	$06000000	Scroll bar standard control
`growProc`	$08000000	Size box standard control

(continued)

Table 4-8 (continued)
Control Manager constants

Name	Value	Description
DefProc (message parameters)		
`drawCtl`	$00	Draw the control
`calcCRect`	$01	Compute the rectangle to drag
`testCtl`	$02	Test where mouse button was pressed
`initCtl`	$03	Perform any additional control initialization
`dispCtl`	$04	Take any additional disposal actions
`posCtl`	$05	Move the control's indicator
`thumbCtl`	$06	Compute the parameters for dragging an indicator
`dragCtl`	$07	Drag either a control's indicator or the entire control
`autoTrack`	$08	Called while dragging if –1 is passed to TrackControl
`newValue`	$09	Called when the control gets a new value
`setParams`	$0A	Called when the control gets new additional parameters
`moveCtl`	$0B	Called when control moves
`recSize`	$0C	Return control record size in bytes
Axis parameters		
`noConstraint`	$0000	No constraint on movement
`hAxisOnly`	$0001	Movement constrained to horizontal axis only
`vAxisOnly`	$0002	Movement constrained to vertical axis only
Part codes		
`noPart`	$00	No part
`simpleButton`	$02	Simple button
`checkBox`	$03	Check box
`radioButton`	$04	Radio button
`upArrow`	$05	Up arrow on scroll bar
`downArrow`	$06	Down arrow on scroll bar
`pageUp`	$07	Page up
`pageDown`	$08	Page down
`growBox`	$0A	Size box
`thumb`	$81	Thumb
HiliteControl parameters		
`noHilite`	$0000	Highlight control
`inactiveHilite`	$00FF	Remove hightlighting from control

Table 4-9
Control Manager data structures

Name	Offset	Type	Definition
CtlRec (control record)			
ctlNext	$00	CtlRecHndl	Handle to next control
ctlOwner	$04	GrafPortPtr	Pointer to control's window
ctlRect	$08	Rect	Enclosing rectangle
ctlFlag	$10	Byte	Bit flags
ctlHilite	$11	Byte	Highlighted part
ctlValue	$12	Integer	Control's value
ctlProc	$14	LongProcPtr	Control's definition procedure
ctlAction	$18	LongProcPtr	Control's action procedure
ctlData	$1C	Longint	Reserved for CtlProc's use
ctlRefCon	$20	Longint	Reserved for application use
ctlColor	$24	Pointer	Control's color table
BarColors (scroll bar color table)			
barOutline	$00	Word	Color for outlining bar, arrows, and thumb
barNorArrow	$02	Word	Color of arrows when not highlighted
barSelArrow	$04	Word	Color of arrows when highlighted
barArrowBack	$06	Word	Color of arrow box's background
barNorThumb	$08	Word	Color of thumb's background when not highlighted
barSelThumb	$0A	Word	Color of thumb's background when highlighted
barPageRgn	$0C	Word	Color and pattern page region; high-order byte: 1 = dither, 0 = solid
barInactive	$0E	Word	Color of scroll bar's interior when inactive
BoxColors (check box color table)			
boxReserved	$00	Word	Reserved for future use
boxNor	$02	Word	Color of box when not checked
boxSel	$04	Word	Color of box when checked
boxTitle	$06	Word	Color of check box's title
BttnColors (button color table)			
bttnOutline	$00	Word	Color of outline
bttnNorBack	$02	Word	Color of background when not selected
bttnSelBack	$04	Word	Color of background when selected
bttnNorText	$06	Word	Color of title's text when not selected
bttnSelText	$08	Word	Color of title's text when selected

(continued)

Table 4-9 (continued)
Control Manager data structures

Name	Offset	Type	Definition
LimitBlk (limit block)			
boundRect	$00	Rect	Drag bounds
slopRect	$08	Rect	Cursor bounds
axisParam	$10	Word	Movement constraints
dragPatt	$12	Pointer	Pattern for drag outline
RadioColors (radio button color table)			
radReserved	$00	Word	Reserved for future use
radNor	$02	Word	Color of radio button when off
radSel	$04	Word	Color of radio button when on
radTitle	$06	Word	Color of radio button's title text

Note: The actual assembly-language equates have a lowercase *o* (the letter) in front of all of the names given in this table.

Table 4-10
Control Manager error codes

Code	Name	Description
$1001	`wmNotStartedUp`	Window Manager not initialized

Chapter 5

Desk Manager

The **Desk Manager** is the tool set that enables your application to support **desk accessories.** Desk accessories are mini-applications that can be run at the same time as an Apple IIGS application. There are two types of desk accessories in the Apple IIGS environment: **classic desk accessories** and **new desk accessories.**

Classic desk accessories (CDAs) are desk accessories designed to function in a nondesktop, nonevent-based environment. Unlike new desk accessories, a CDA has full control of the computer during what is basically an interrupt state (generated by a key press). The desk accessory is responsible for saving and restoring any of the application's memory that it uses as well as handling all I/O.

New desk accessories (NDAs) are desk accessories designed to execute in a desktop, event-driven environment. NDAs run in a window and have control when that window is the frontmost window. The control executed by an NDA is described in this chapter.

❖ *Macintosh programmers:* New desk accessories are the kind of desk accessories available on the Macintosh.

A preview of the Desk Manager routines

To introduce you to the capabilities of the Desk Manager, all Desk Manager routines are grouped by function and briefly described in Table 5-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the Desk Manager routines (discussed in alphabetical order).

Table 5-1
Desk Manager routines and their functions

Routine	Description
Housekeeping routines	
DeskBootInit	Initializes the Desk Manager; called only by the Tool Locater—must not be called by an application
DeskStartUp	Starts up the Desk Manager for use by an application
DeskShutDown	Shuts down the Desk Manager when an application quits
DeskVersion	Returns the version number of the Desk Manager
DeskReset	Resets the Desk Manager; called only when the system is reset—must not be called by an application
DeskStatus	Indicates whether the Desk Manager is active
Installation routines	
InstallNDA	Installs a specified NDA in the system
InstallCDA	Installs a specified CDA in the system
Classic desk accessory routines	
ChooseCDA	Activates the Desk Manager and displays the CDA menu—must not be called by an application
SetDAStrPtr	Changes the names of the built-in CDAs
GetDAStrPtr	Returns the pointer to the names of the built-in CDAs
New desk accessory routines	
OpenNDA	Opens a specified NDA
CloseNDA	Closes a specified NDA
CloseNDAbyWinPtr	Closes an NDA with a specified window pointer
CloseAllNDAs	Closes all open NDAs
FixAppleMenu	Adds the names of the NDAs to a specified menu
GetNumNDAs	Returns the total number of NDAs currently installed
SystemClick	Handles mouse-down events in a system (that is, desk accessory) window
SystemEdit	Passes standard menu edits to system windows
SystemTask	Causes a desk accessory to perform its periodic action
SystemEvent	Previews all events returned to an application and indicates whether the event has been processed by a desk accessory
State-saving routines	
SaveScrn	Saves the 80-column text screens in banks $00, $01, $E0, and $E1—must not be called by an application
RestScrn	Restores the screen area saved by the Desk Manager—must not be called by an application
SaveAll	Saves all the variables that the Desk Manager preserves when the CDA menu is activated—must not be called by an application
RestAll	Restores all the variables that the Desk Manager preserves when the CDA menu is activated—must not be called by an application

Using classic desk accessories

A user activates a classic desk accessory from the CDA menu. The **CDA menu** is displayed by pressing Apple-Control-Escape. Two CDAs are built into the system:

- Control Panel
- Alternate Display Mode

Any others (up to 11) are loaded from disk. From the CDA menu, a user can select any of the CDAs currently in the system. The desk accessory selected is activated and retains control until it shuts down. When it shuts down, the Desk Manager redisplays the CDA menu. Only when the user selects Quit from the CDA menu does the original application resume operation.

When the CDA menu can be displayed

The Desk Manager obtains control in two ways. If the Event Manager is active, the Desk manager is called in conjunction with GetNextEvent. When the user presses Apple-Control-Escape, a desk accessory event is posted. Later, when the GetNextEvent routine is called, the Desk Manager previews the event and, if the event is a desk accessory event (among others), retrieves the event for processing.

If the Event Manager is not active, the Desk Manager gets control whenever the user presses Apple-Control-Escape. Before the manager displays the CDA menu, it checks the system **busy flag.** If something in the system is busy, the Desk Manager schedules a wake-up with the Scheduler. The next time the system flag is free, the Scheduler wakes up the Desk Manager, which then can display the CDA menu. This guarantees that CDAs have all system resources available to them when they are called.

See Chapter 19, "Scheduler," in Volume 2 for more information.

Writing classic desk accessories

Classic desk accessories are load files that have a file type of $B9 and must be placed on the system disk in the DESK.ACCS subdirectory of the SYSTEM directory.

A CDA must start with a header section as follows:

```
StartOfDA      str'Name of DA'              ; Name of the DA

               dc i4'StartOfDACode'         ; Pointer to start of code

               dc i4'ShutDownRoutine'       ; Pointer to shutdown routine
```

❖ *Note:* The *str* in the preceding example is an APW-specific macro that generates a **Pascal-type string** (a string beginning with a length byte and followed by ASCII characters).

The header section contains the name of the desk accessory and two pointers. The first pointer is the address of the primary entry point—the activation entry point—to the CDA. The CDA gets control through this entry point, with the processor in full native mode.

The second pointer points to the entry point used whenever the DeskShutDown routine is called. DeskShutDown is called by all applications that have issued a DeskStartUp call, and also by ProDOS when it switches from ProDOS 16 to ProDOS 8 or vice versa.

❖ *Note:* The second entry point is necessary because CDAs can spawn background tasks that rely on the availability of the current ProDOS. The shutdown routine allows the CDA to stop the tasks.

When the Desk Manager displays the CDA menu, it saves the text pages in banks $0, $1, $E0, and $E1, along with pages 0 and 1 of bank $0.

Important

Only the screen holes used by the Desk Manager are preserved.

These parts of memory, which contain the system direct page and stack, are restored by the Desk Manager when the user selects Quit from the CDA menu. Thus, a CDA can use almost all of this memory, except for the stack, as it sees fit. Since the Desk Manager's return address is on the stack (along with other Desk Manager variables), the CDA cannot cut the stack back any farther than it is when it gets control.

A CDA must be careful when using any other memory in the system that it does not already own. A CDA can ask the Memory Manager for additional memory, but there is no guarantee that the memory will be available. For example, ProDOS 8 applications already have all special memory reserved for them.

❖ *Note:* The CDA can obtain a user ID by using the User ID Manager routines in the Miscellaneous Tool Set. See Chapter 14, "Miscellaneous Tool Set," for more information.

Important

The CDA must be able to respond appropriately when no additional memory is available.

Supporting new desk accessories

New desk accessories are loaded automatically by the operating system at boot time. An application that wants to make NDAs available to the user does not have to do a lot of work, particularly if the application is using the Window Manager routine TaskMaster. The application must, however, make sure that all the tool sets listed in Table 5-2 are loaded and started up.

Table 5-2
Tool sets required to support new desk accessories

Tool set number		Tool set name	Minimum version needed
$01	#01	Tool Locator	1.2
$02	#02	Memory Manager	1.2
$03	#03	Miscellaneous Tool Set	1.2
$04	#04	QuickDraw II	1.2
$06	#06	Event Manager	1.0
$0E	#14	Window Manager	1.3
$0F	#15	Menu Manager	1.3
$10	#16	Control Manager	1.3
$14	#20	LineEdit Tool Set	1.0
$15	#21	Dialog Manager	1.0
$16	#22	Scrap Manager	1.0

Supporting new desk accessories with TaskMaster

If an application uses TaskMaster, it needs to make only three calls to support new desk accessories after it has loaded and started up the proper tool sets:

- **DeskStartUp:** to start up the Desk Manager
- **FixAppleMenu:** to add the list of NDAs to the Apple menu
- **DeskShutDown:** to shut down the Desk Manager before the other tool sets are shut down

After the FixAppleMenu call has been made, TaskMaster automatically handles opening NDAs in response to menu selections, calling SystemTask and SystemClick when appropriate. If the application sets up the menu items correctly, TaskMaster can even call SystemEdit when the user selects an item from the Edit menu or close a desk accessory in response to the user selecting Close from the File menu (if the appropriate task mask has been set up).

Supporting new desk accessories without TaskMaster

Applications that do not use TaskMaster must take the following steps to support new desk accessories:

1. Call DeskStartUp to initialize the Desk Manager.
2. Call FixAppleMenu to add the list of NDAs to the Apple menu.
3. Call OpenNDA when the user selects an NDA from the Apple menu.
4. Call SystemTask frequently (at least every time through the event loop).
5. Call SystemClick when a mouse-down event occurs in a system window.
6. Call SystemEdit when a desk accessory is active and the user selects Undo, Cut, Copy, Paste, or Clear from the Edit menu.
7. Close an NDA when the user selects Close from the File menu. You can use CloseNDA or CloseNDAbyWinPtr to do this.
8. Call DeskShutDown to shut down the Desk Manager.

Writing new desk accessories

New desk accessories are load files with file type $B8. An NDA must be placed on the system disk in the DESK.ACCS subdirectory of the SYSTEM directory. NDAs have four entry points: open, close, action, and init. For each of these entry points the processor is in full native mode. No direct page is assigned to an NDA, so it must obtain space from the stack or by asking the Memory Manager.

A new desk accessory can assume that the tool sets shown in Table 5-2 have been loaded and started up. A new desk accessory can also assume that the Print Manager is available but not necessarily loaded.

The NDA is responsible for saving and restoring important globals such as the current GrafPort.

The NDA must start with a header section as follows:

```
StartOfDA      dc i4'PtrToOpen'      ; Pointer to the open routine, which should RTL
               dc i4'PtrToClose'     ; Pointer to the close routine, which should RTL
               dc i4'PtrToAction'    ; Pointer to the action routine, which should RTL
               dc i4'PtrToInit'      ; Pointer to the init routine, which should RTL
               dc i2'Period'         ; How often the NDA gets run codes
               dc i2'EventMask'      ; Describes what events it wants
               dc c'  MenuLine \H**' ; The text that describes the menu item
               dc i1'0'              ; Terminator for the menu line
```

When the Desk Manager calls the open routine, it puts 4 bytes on the stack before it pushes the RTL address. The open routine must replace those 4 bytes with a pointer to its window and leave the RTL address intact.

The close routine has no inputs or outputs. However, it must be able to work even if the desk accessory is not open when the routine is called.

The action code is passed to the action routine in the A register; other information is passed to the action routine in the X and Y registers. The possible action codes are shown in Table 5-3.

Table 5-3
New desk accessory action codes

Code	Action	Description
1	`eventAction`	Passed when an event must be handled by the desk accessory. The X and Y registers contain a pointer to the event record (low-order word in X, high-order word in Y). The only events that can be passed to an NDA are ButtonDown, ButtonUp, KeyDown, AutoKeyDown, Update, and Activate. Update and Activate events for a desk accessory are always passed to it. The other four are passed to it only if the *eventMask* indicates that they should be passed.
2	`runAction`	Passed when the specified time period has elapsed.
3	`cursorAction`	Passed to a desk accessory if it is the frontmost window when SystemTask is called. This allows the desk accessory to change the cursor when it is over the NDAs window.
4	Not used	
5	`undoAction`	These codes are passed to a desk accessory when the application determines that the user has selected one of these edit commands from the Edit menu. The action call should return a Boolean value in the A register indicating whether the command was handled.
6	`cutAction`	
7	`copyAction`	
8	`pasteAction`	
9	`clearAction`	

The InitRoutine is called for each installed NDA every time DeskStartUp or DeskShutDown is called. The Desk Manager passes a variable in the A register to indicate whether a DeskStartUp or DeskShutDown call is being made. A 0 indicates a shutdown call; a nonzero value indicates a startup call.

❖ *Note:* As part of its cleanup process, ProDOS 16 shuts down all new desk accessories every time it starts an application.

The Period field of the header section describes how often the DA should be called with the `runAction` code, as shown in Table 5-4.

Table 5-4
New desk accessory
Period field values

Period	Interval
0	As often as possible
1	Every sixtieth of a second
2	Every thirtieth of a second
60	Every second
$FFFF	Never

The action routine is called with the `runAction` code from SystemTask. The application should call SystemTask every time through its event loop.

The MenuLine is a line of text that will be passed to the Menu Manager to appear in the Apple menu. As shown in the preceding assembly-language header section, the line must start with two place-holding characters because the Menu Manager puts something in those positions. The line must also have a backslash (\) in it and an *H* followed by two place-holding characters. The place-holding characters are replaced with the menu item ID for the desk accessory when the FixAppleMenu routine is called.

$0105 DeskBootInit

Initializes the Desk Manager; called only by the Tool Locator.

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 Call must not be made by an application.

$0205 DeskStartUp

Starts up the Desk Manager for use by an application. An application must make this call if it wishes to support desk accessories.

Important
Your application must make this call before it makes any other Desk Manager calls. In addition, all of the tools required by NDAs must be started up before this call is made. It is also important that applications not make this call unless they completely support NDAs.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void DeskStartUp()
```

$0305 DeskShutDown

Shuts down the Desk Manager.

Important

If your application has started up the Desk Manager, the application must make this call before it quits. In addition, this call must be made before any of the required tools are shut down.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

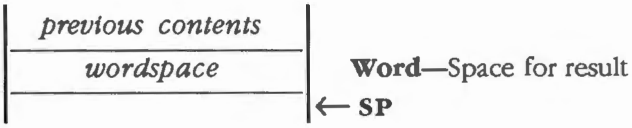
```
extern pascal void DeskShutDown()
```

$0405 DeskVersion

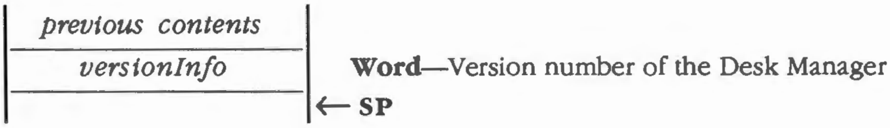
Returns the version number of the Desk Manager.

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal Word DeskVersion()
```

$0505 DeskReset

Resets the Desk Manager; called 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 Call must not be made by an application.

$0605 DeskStatus

Indicates whether the Desk Manager is active.

Parameters

Stack before call

previous contents
wordspace — **Word**—Space for result
← **SP**

Stack after call

previous contents
activeFlag — **Word**—BOOLEAN; TRUE if Desk Manager active, FALSE if inactive
← **SP**

Errors None

C

```
extern pascal Boolean DeskStatus()
```

$1105 ChooseCDA

Activates the Desk Manager and displays the CDA menu. ChooseCDA causes the Desk Manager to display the CDA menu as if the user pressed the appropriate keys.

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 Call must not be made by an application.

$1D05 CloseAllNDAs

Closes all open NDAs.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

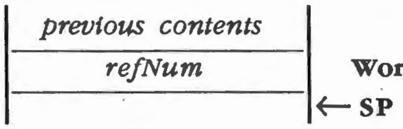
```
extern pascal void CloseAllNDAs()
```

$1605 CloseNDA

Closes a specified NDA. You normally won't use the routine in an application, because NDAs are closed automatically when the user presses the mouse button with the cursor in the close box (the SystemClick routine handles that situation).

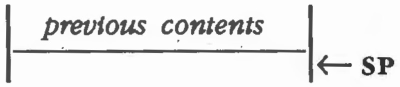
Parameters

Stack before call



Word—As returned by the OpenNDA routine

Stack after call



Errors

None

C

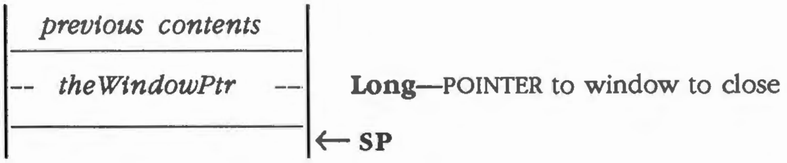
```
extern pascal void CloseNDA(refNum)

Word     refNum;
```

$1C05 CloseNDAbyWinPtr

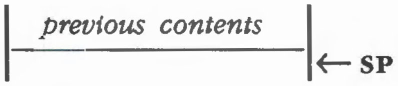
Closes the NDA whose window pointer is equal to the one that is passed. This call is handy when the system is trying to close a desk accessory because the user chose Close from the File menu. When the user chooses Close, your application should use the Window Manager FrontWindow routine to determine which window is to be closed. If the front window is not an application window, the application can pass the pointer to the CloseNDAbyWinPtr routine.

Parameters

Stack before call



Stack after call



Errors

$0510	`daNotFound`	Specified DA not available
$0511	`notSysWindow`	Window pointer is not a pointer to a window owned by an NDA

C

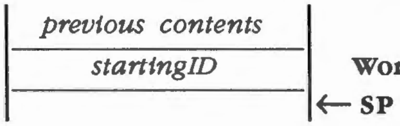
```
extern pascal void CloseNDAByWinPtr(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$1E05 FixAppleMenu

Adds the names of the NDAs to a specified menu. This call is used to add the names of the currently installed NDAs to a menu (usually the Apple menu). The first NDA appended to the menu is given an ID of 1, the second NDA an ID of 2, and so on.

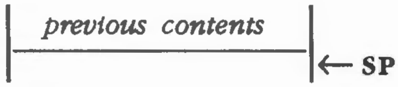
Parameters

Stack before call



Word—ID of menu that will display the NDAs

Stack after call



Errors

None

C

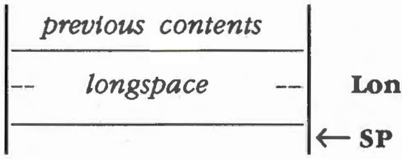
```
extern pascal void FixAppleMenu(startingID)

Word     startingID;
```

$1405 GetDAStrPtr

Returns the pointer to the table of strings containing the built-in CDA names.

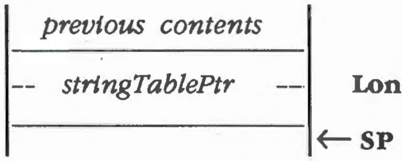
Parameters

Stack before call



Long—Space for result

Stack after call



Long—POINTER to table of strings for the built-in CDA names

Errors

None

C

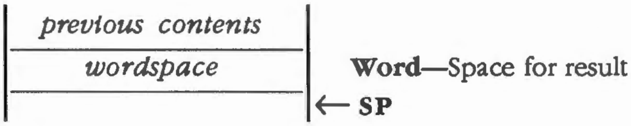
```
extern pascal Pointer GetDAStrPtr()
```

$1B05 GetNumNDAs

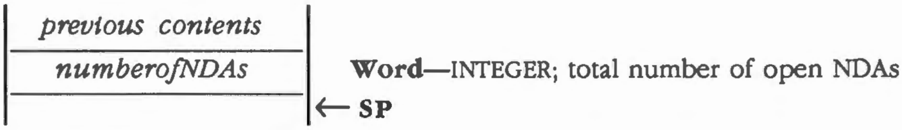
Returns the total number of NDAs currently installed.

Parameters

Stack before call



Stack after call



Errors

None

C

```
extern pascal unsigned int GetNumNDAs()
```

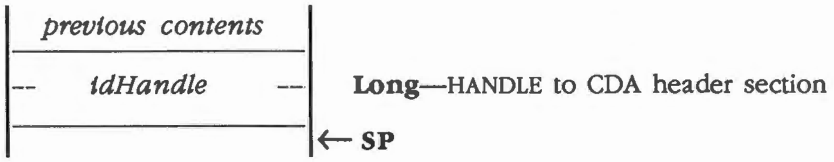
$0F05 InstallCDA

Installs a specified CDA in the system. This routine is normally called only by ProDOS 16 when the machine is booted.

The CDA header section is described in the section "Writing Classic Desk Accessories" in this chapter.

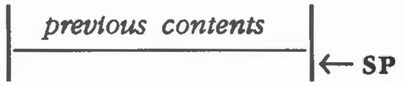
Parameters

Stack before call



Long—HANDLE to CDA header section

Stack after call



Errors

None

C

```
extern pascal void InstallCDA(idHandle)

Handle     idHandle;
```

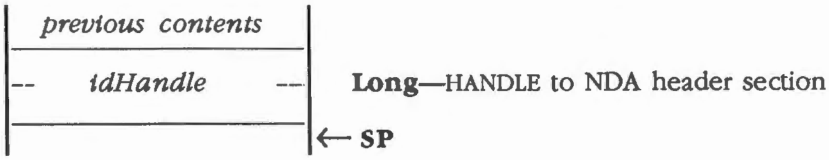
$0E05 InstallNDA

Installs a specified new desk accessory in the system. This routine is normally called only by ProDOS 16 when the machine is booted.

The NDA header section is described in the section "Writing New Desk Accessories" in this chapter.

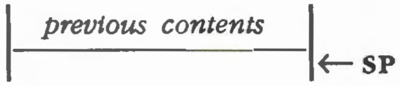
Parameters

Stack before call



Long—HANDLE to NDA header section

Stack after call



Errors

None

C

```
extern pascal void InstallNDA(idHandle)

Handle      idHandle;
```

$1505 OpenNDA

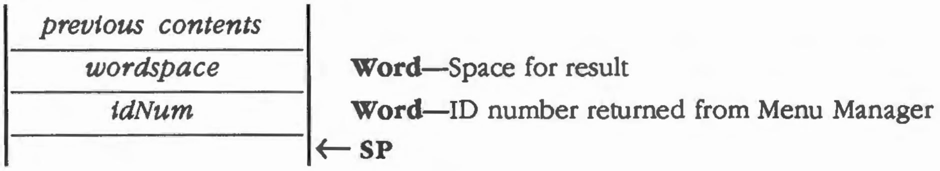
Opens a specified NDA. The *idNum* passed is the same ID returned by the Menu Manager and set up by the FixAppleMenu call.

❖ *Note:* If your application is using the Window Manager routine TaskMaster, the application doesn't need to make the OpenNDA call.

An application should make this call when the user selects an NDA from the Apple menu.

Parameters

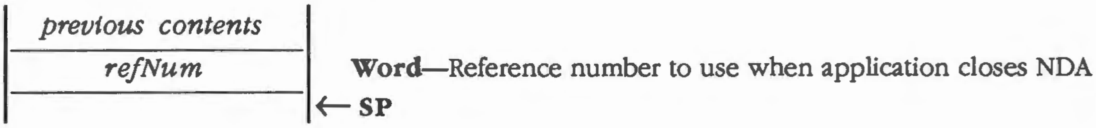
Stack before call



Word—Space for result

Word—ID number returned from Menu Manager

Stack after call



Word—Reference number to use when application closes NDA

Errors

$0510	daNotFound	Specified DA not available

C

```
extern pascal Word OpenNDA(idNum)
Word     idNum;
```

$0C05 RestAll

Restores all the variables that the Desk Manager preserves when the CDA menu is activated.

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 Call must not be made by an application.

$0A05 RestScrn

Restores the screen area saved by the Desk Manager. The entire screen and all of the the screen holes are restored.

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 Call must not be made by an application.

$0B05 SaveAll

Saves all the variables that the Desk Manager preserves when the CDA menu is activated. The routine also sets the display to text mode in either 640 or 320 mode, depending upon the settings in the Control Panel.

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 Call must not be made by an application.

$0905 SaveScrn

Saves the 80-column text screens in banks $00, $01, $E0, and $E1. This new image of the screen is used for subsequent calls to the RestScrn routine. The entire screen and all of the screen holes are preserved.

Warning
An application must never make this call. The screen is saved in only one place, so a subsequent desk accessory call could destroy the screen.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

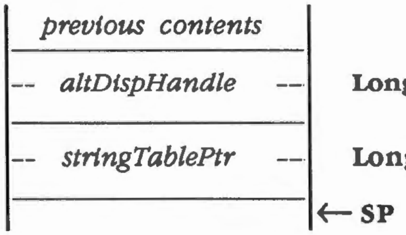
C Call must not be made by an application.

$1305 SetDAStrPtr

Changes the names of the built-in CDAs. This routine can be used to localize the built-in desk accessories.

Parameters

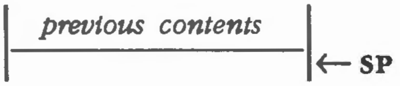
Stack before call



Long—HANDLE to new alternate display desk accessory

Long—POINTER to table of strings

Stack after call



Errors

None

C

```
extern pascal void SetDAStrPtr(altDispHandle,stringTablePtr)
Handle     altDispHandle;
Pointer     stringTablePtr;
```

(continued)

Alternate-display-mode desk accessory

Your alternate-display-mode desk accessory should contain code similar to the following:

```
AltDispDA      anop
               dc      il'StrEnd-StrStart'         ; Length of string
StrStart       dc      c'Alternate Display Mode'   ; Name you want
StrEnd         dc      i4'Open'
               dc      i4'ShutDown'

Open           sep #$30                            ; 8-bit m and x
               longa off
               longi off
               phb                                 ; Save data bank register
               lda     #$00                        ; Set db reg to $00
               pha
               plb
               jsl     $E100A4                     ; Alternate display mode vector
               plb                                 ; Restore data bank register
               rep #$30                            ; 16-bit m and x
               longa on
               longi on
ShutDown       rtl
```

Table of strings

The table of strings pointed to by *stringTablePtr* must look like this:

```
StringTable    dc      i4'titlestr'                ; Title line
               dc      i4'controlstr'              ; Control Panel
               dc      i4'quitstr'                 ; Quit
               dc      i4'selectstr'               ; Select string
```

The strings currently used are discussed in the following sections.

Title string

The title string must be exactly 39 characters long and must be built in a manner similar to the following:

```
titlestr       dc      h'5A A0 41 A0'               ; Bar-space-Apple-space characters
               dc      c'Desk Accessories '         ; Note space after string
               dc      18h'20'                      ; Inverse
               dc      h'5F 00'                     ; Bar and terminator
```

If you change the title (*Desk Accessories* in the preceding example), you should also change the number (*18* here) appropriately. The number specifies the number of inverse spaces to display.

❖ *Note:* The strings used for titles by the Desk Manager are **C strings** (strings terminated by $00); thus, there is no length byte.

Control string

The control string must be less than 34 characters long.

```
controlstr     dc      c'Control Panel'
               dc      h'00'
```

Quit string

The quit string must be less than 34 characters long.

```
quitstr        dc      c'Quit'
               dc      h'00'
```

Select string

This string must be exactly 39 characters long and must be built similar to the following:

```
selectstr      dc      h'5A'
               dc      c' Select '                  ; Note space before and after string
               dc      h'4A A0 4B'
               dc      17h'A0'
               dc      c'Open: '                    ; Note space after string
               dc      h'4D A0 A0 5F 00'
```

If you change the length of the words (*Select* or *Open*), you should alter the number *17* in the fourth line of the example. This alters the number of spaces between *Select* and *Open*.

$1705 SystemClick

SystemClick handles mouse-down events in the size, drag, zoom, and close boxes. This routine should be called when the application detects a mouse-down event in a system window.

❖ *Note:* If an application is using TaskMaster, it never needs to make this call. TaskMaster does the work for it.

If the window is inactive and the event is in the content area, information area, window frame, or vertical or horizontal scroll bar, SystemClick will make the window active. If the window is already active, SystemClick passes the event to the desk accessory.

Parameters

Stack before call

previous contents	
-- *eventRecPtr* --	**Long**—POINTER to event record
-- *theWindowPtr* --	**Long**—POINTER to system window
findWndwResult	**Word**—Result of the FindWindow call
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SystemClick(eventRecPtr,theWindowPtr,findWndwResult)

EventRecordPtr     eventRecPtr;

GrafPortPtr     theWindowPtr;

Word     findWndwResult;
```

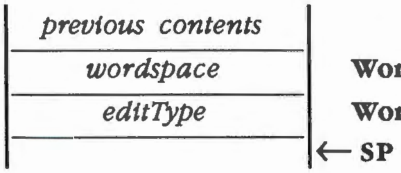
$1805 SystemEdit

Passes standard menu edits to system windows. The valid edit types are

1	`undo`
2	`cut`
3	`copy`
4	`paste`
5	`clear`

Parameters

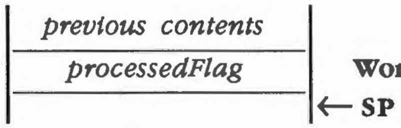
Stack before call



Word—Space for result

Word—Edit type

Stack after call



Word—BOOLEAN; TRUE if frontmost window system window and desk accessory has to handle call, otherwise FALSE

Errors

None

C

```
extern pascal Boolean SystemEdit(editType)

Word     editType;
```

$1A05 SystemEvent

Entry point the Event Manager uses into the Desk Manager. Every event returned to an application is first processed by SystemEvent, which returns TRUE if the event has been processed by a desk accessory and FALSE if it is to be sent to the application. The CDA activation keystroke is processed in this way.

Warning
An application must never make this call.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
eventWhat	**Word**—From event record
-- *eventMessage* --	**Long**—From event record
-- *eventWhen* --	**Long**—From event record
-- *eventWhere* --	**Long**—From event record
eventMods	**Word**—From event record
	← **SP**

Stack after call

previous contents	
processFlag	**Word**—BOOLEAN; TRUE if event is to be processed by DA, FALSE if not
	← **SP**

Errors None

C Call must not be made by an application.

$1905 SystemTask

For each open desk accessory, SystemTask causes the accessory to perform the periodic action defined for it, if any such action was defined and if the proper time period has elapsed since the action was last performed. The routine should be called periodically by an application to support desk accessories that perform periodic actions.

❖ *Note:* If the application is using the Window Manager routine TaskMaster, it doesn't need to make this call. TaskMaster does the work for it.

For example, a clock accessory can be defined such that the second hand moves once every second; the periodic action for the accessory is to move the second hand to the next position, and SystemTask will alert the accessory every second to perform that action.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

None

C

```
extern pascal void SystemTask()
```

Desk Manager summary

This section briefly summarizes the constants and tool set error codes contained in the Desk Manager. There are no predefined data structures for the Desk Manager.

Important

These definitions are provided in the appropriate interface file.

Table 5-5
Desk Manager constants

Name	Value	Description
NDA action codes		
eventAction	$0001	Code passed if event is to be handled by desk accessory
runAction	$0002	Code passed when specified time period has elapsed
cursorAction	$0003	Code passed if desk accessory is frontmost window when SystemTask called
undoAction	$0005	Code passed when user selects Undo from Edit menu
cutAction	$0006	Code passed when user selects Cut from Edit menu
copyAction	$0007	Code passed when user selects Copy from Edit menu
pasteAction	$0008	Code passed when user selects Paste from Edit menu
clearAction	$0009	Code passed when user selects Clear from Edit menu
Edit types		
undo	$0001	Undo edit type
cut	$0002	Cut edit type
copy	$0003	Copy edit type
paste	$0004	Paste edit type
clear	$0005	Clear edit type

Table 5-6
Desk Manager error codes

Code	Name	Description
$0510	daNotFound	Specified DA not available
$0511	notSysWindow	Window pointer is not a pointer to a window owned by an NDA

Chapter 6

Dialog Manager

The **Dialog Manager** allows you to implement dialog boxes and the alert mechanism, two means of communication between the application and the user. The Dialog Manager provides these capabilities in a way consistent with the Apple *Human Interface Guidelines: The Apple Desktop Interface.*

You should already be familiar with the following:

- The basic concepts and structures behind QuickDraw, particularly rectangles, GrafPorts, and pictures
- The Event Manager, the Window Manager, and the Control Manager
- The LineEdit Tool Set, to understand text editing in dialog boxes

A preview of the Dialog Manager routines

To introduce you to the capabilities of the Dialog Manager, all Dialog Manager routines are grouped by function and briefly described in Table 6-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the Dialog Manager routines (discussed in alphabetical order).

Table 6-1
Dialog Manager routines and their functions

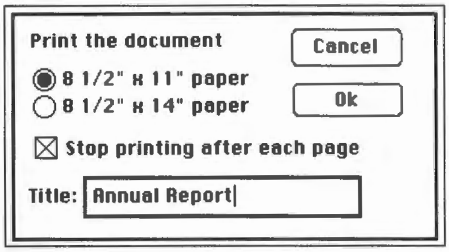
Routine	Description
Housekeeping routines	
DialogBootInit	Initilializes the Dialog Manager; called only by the Tool Locator—must not be called by an application
DialogStartUp	Starts up the Dialog Manager for use by an application
DialogShutDown	Shuts down the Dialog Manager and frees any memory allocated by the Dialog Manager
DialogVersion	Returns the version number of the Dialog Manager
DialogReset	Resets the Dialog Manager; called only when the system is reset—must not be called by an application
DialogStatus	Indicates whether the Dialog Manager is active
Dialog creation and disposal routines	
NewModalDialog	Creates a specified modal dialog and returns a pointer to the GrafPort of the new dialog
NewModelessDialog	Creates a specified modeless dialog and returns a pointer to the GrafPort of the new dialog
GetNewModalDialog	Creates a modal dialog and returns a pointer to the port of the new dialog
CloseDialog	Removes a specified dialog window from the screen and deletes it from the window list
Item creation and removal routines	
NewDItem	Adds a new item to the dialog's item list
GetNewDItem	Adds a new item to a specified dialog's item list using a template
RemoveDItem	Removes a specified item from a specified dialog and erases it from the screen
Dialog event-handling routines	
ModalDialog	If the frontmost window is a modal dialog box, ModalDialog repeatedly gets and handles events in the dialog's window
ModalDialog2	If the frontmost window is a modal dialog, ModalDialog2 repeatedly gets and handles events in the dialog's window; after handling an event involving an enabled dialog item, it returns with the part code and the item ID in *itemHitInfo*
IsDialogEvent	Determines whether a specified event needs to be handled as part of a modeless dialog
DialogSelect	Handles an event as part of a specified modeless dialog
DlgCut	Checks whether a specified dialog has any editLine items and, if so, applies the LineEdit procedure LECut to the currently selected editLine item
DlgCopy	Checks whether a specified dialog has any editLine items and, if so, applies the LineEdit routine LECopy to the current editLine item
DlgPaste	Checks whether a specified dialog has any editLine items and, if so, applies the LineEdit routine LEPaste to the current editLine item
DlgDelete	Checks whether a specified dialog has any editLine items and, if so, applies the LineEdit routine LEDelete to the current editLine item
DrawDialog	Draws the contents of a specified dialog box

Table 6-1 (continued)
Dialog Manager routines and their functions

Routine	Description
Alert routines	
Alert	Invokes an alert defined by a specified alert template
StopAlert	Invokes an alert defined by a specified alert template and draws the stop icon in the upper-left corner of the alert box
NoteAlert	Invokes an alert defined by a specified alert template and draws the note icon in the upper-left corner of the alert box
CautionAlert	Invokes an alert defined by a specified alert template and draws the caution icon in the upper-left corner of the alert box
Item manipulation routines	
ParamText	Specifies text for 1–4 special strings in statText, longStatText, and longStatText2 items
GetControlDItem	Returns a handle to the control for a specified item
GetIText	Returns the text of a specified statText or editLine item in a specified dialog box
SetIText	Provides the text for a specified statText or editLine item in a specified dialog box and draws the item
SelectIText	Sets the selection range or insertion point for a specified editLine item in a specified dialog box
GetDItemType	Returns the type of a specified item (buttonItem, radioItem, statText, etc.)
SetDItemType	Changes a specified item to a new specified item type
GetDItemBox	Returns the display rectangle of a specified item
SetDItemBox	Changes the display rectangle of a specified item to a new display rectangle
GetFirstDItem	Returns the ID of the first item in a specified dialog
GetNextDItem	Returns the ID of the next item in a specified dialog after a specified item
GetDefButton	Returns the ID of the default button item in a specified dialog
SetDefButton	Sets the ID of the default button to a specified ID
GetDItemValue	Returns the current value of a specified item
SetDItemValue	Sets the value of a specified item to a new desired value and redraws the item
GetAlertStage	Returns the stage of the last occurrence of an alert as a number from 0 to 3
ResetAlertStage	Resets the stage of the last occurrence of an alert so that the next occurrence of that same alert will be treated as its first stage
DefaultFilter	Calls the standard default filter used by ModalDialog or Alert when no user filter procedure is specified
HideDItem	Erases a specified item from a specified dialog
ShowDItem	Makes visible a specified item from a specified dialog
FindDItem	Returns the ID of the item located at a specified point in a specified dialog
UpdateDialog	Redraws the part of a specified dialog that is in a specified update region
DisableDItem	Disables a specified item in a specified dialog
EnableDItem	Enables a specified item in a specified dialog
Miscellaneous routines	
ErrorSound	Establishes the sound procedure for alerts
SetDAFont	Specifies the font for the dialog or alert window's GrafPort

Dialog boxes

A **dialog box** appears on the screen when an application needs more information to carry out a command. As shown in Figure 6-1, a dialog box typically resembles a form on which the user checks boxes and fills in blanks.



Print the document

● 8 1/2" x 11" paper

○ 8 1/2" x 14" paper

☒ Stop printing after each page

Title: Annual Report

Figure 6-1
Typical dialog box

By convention, a dialog box appears slightly below the menu bar, is somewhat narrower than the screen, and is centered between the left and right edges of the screen. It may contain any or all of the following:

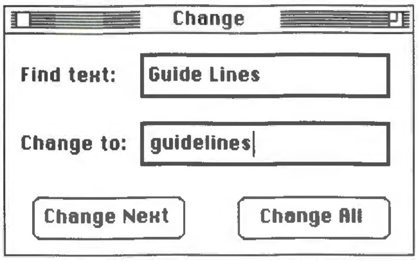
- Informative or instructional text
- Rectangles in which text may be entered (initially blank or containing default text that can be edited)
- Controls of any kind
- Graphics (icons or QuickDraw II pictures)
- Anything else, as defined by the application

The user supplies any necessary information in the dialog box; for example, by entering text or clicking a check box. The dialog box usually contains a button labeled *OK* to tell the application to accept the information provided and perform the command, and a button labeled *Cancel* to cancel the command as though it had never been given (retracting all actions since its invocation). Some dialog boxes may use a more descriptive word than *OK;* for simplicity, this chapter refers to the button as the *OK button*. There may even be more than one button that will perform the OK command, each in a different way.

Most dialog boxes require the user to respond before doing anything else. Clicking a button to perform or cancel the command makes the box disappear; clicking outside the dialog box causes only a beep from the speaker. This type of box is called a **modal dialog box** because it puts the user in the state, or *mode,* of being able to work only inside the dialog box. A modal dialog box usually has the same general appearance as the box shown in Figure 6-1. One of the buttons in the dialog box may be boldly outlined. Pressing the Return key has the same effect as clicking the outlined button or, if no outlined button exists, the OK button; the particular button whose effect occurs is called the dialog's default button and is the preferred ("safest") button to use in the current situation. If no boldly outlined or OK button appears, pressing the Return key should not, by convention, have any effect.

Other dialog boxes do not require the user to respond before doing anything else; these are called **modeless dialog boxes.** The user can, for example, work in document windows on the desktop before clicking a button in the dialog box, and modeless dialog boxes can be set up to respond to the standard editing commands in the Edit menu. Clicking a button in a modeless dialog box will not make the box disappear: The box will remain visible so that the user can perform the command again. A Cancel button, if present, will simply stop the action currently being performed by the command; this is useful for long printing or searching operations.

As shown in Figure 6-2, a modeless dialog box looks like a document window. It can be moved, made inactive and active again, or closed like any document window. When you're done with the command and want the box to disappear, you can click its close box or choose Close from the File menu when the dialog box is the active window.



Find text: Guide Lines

Change to: guidelines

Figure 6-2
Modeless dialog box

A dialog box can't be modal and modeless at the same time; different routines are used to create the two types.

Some dialog boxes may not require any response at all. For example, when an application is performing a time-consuming process, it can display a dialog box that contains only a message telling what it's doing; then, when the process is complete, the application can simply remove the dialog box.

The alert mechanism provides applications with a means of reporting errors or giving warnings. An **alert box** is similar to a modal dialog box, but an alert box appears only when something has gone wrong or must be brought to the user's attention. The alert box is usually placed slightly farther below the menu bar than a dialog box. To help the user who isn't sure how to proceed when an alert box appears, the preferred button to use in the current situation is outlined in bold so that it stands out from the other buttons in the alert box. The outlined button is also the alert box's default button; if the user presses the Return key, the effect is the same as clicking this button. See Figure 6-3.

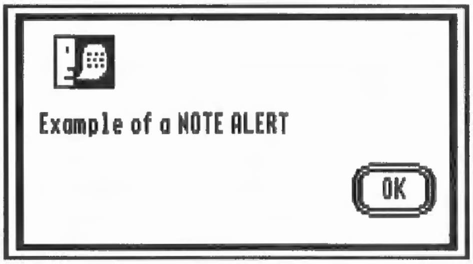

Example of a NOTE ALERT

Figure 6-3
Typical alert box

There are three standard kinds of alerts—Stop, Note, and Caution—each indicated by a particular icon in the upper-left corner of the alert box. Figure 6-3 illustrates a Caution alert. The icons identifying Stop and Note alerts are similar; instead of a danger sign, they show a stop sign and a talking face, respectively. Other alerts can have anything in the the upper-left corner, including blank space if desired.

The alert mechanism also provides another type of signal: sound from the speaker. The application can base its response on the number of consecutive times an alert occurs; the first time, it might simply beep, and thereafter it may present an alert box. The sound isn't limited to a single beep but may be any sequence of tones and may occur either alone or along with an alert box. As an error is repeated, the default button can also change (perhaps from OK to Cancel). You can specify different responses for up to four occurrences of the same alert.

Dialog and alert windows

A dialog box appears in a **dialog window.** When you call a Dialog Manager routine to create a dialog, you supply the same kind of information as when you create a window with a Window Manager routine. For example, you usually take the following steps:

1. Call GetNewModalDialog, NewModalDialog, or NewModelessDialog to determine how the window looks and behaves.
2. Supply a rectangle that becomes the port rectangle of the window's GrafPort.
3. Specify whether the window is visible or invisible.

For modeless dialog boxes, you also specify the window's plane (which, by convention, should initially be the frontmost). The dialog window is created as specified.

You can manipulate a dialog window just like any other window with Window Manager or QuickDraw routines—you can show it, hide it, move it, or change its size or plane, for example—all, of course, in conformance with the Apple *Human Interface Guidelines*. The Dialog Manager can use the clip region of the dialog window's GrafPort, so if you want clipping to occur, you can set this region with the QuickDraw II routine SetClip or ClipRect.

In the same fashion as a dialog box, an alert box appears in an **alert window.** You don't have the same flexibility in defining and manipulating an alert window, however. The Dialog Manager chooses the window definition procedure, so that all alert windows have much the same appearance and behavior. The size and location of the box are supplied as part of the alert box definition. You don't specify the alert window's plane; it always appears in front of all other windows. Since an alert box requires the user to respond before doing anything else and the response makes the box disappear, the application doesn't manipulate the alert window.

Item templates

To create a dialog or an alert box, the Dialog Manager needs to know what items the dialog or alert box contains. Your application passes this information as a list of pointers to **item templates.**

An item template contains the following information:

- An ID number uniquely identifying the item. All subsequent Dialog Manager calls referring to that item will be made using the ID number.
- The type of item. This includes not only whether the item is a standard control, editable text, or some other type, but also whether the Dialog Manager should return to the application when the item is clicked.
- An item descriptor such as a title for a control, a procedure pointer for a user-defined item, or text for an editable or noneditable text item.
- A display rectangle, which determines the location of the item within the dialog or alert box.
- The initial value of a standard control, the word length of a longStatText item, the maximum string length of an editLine item, or any value you want for a userItem.
- A flag determining whether the item should originally be visible or invisible and including item-specific information; for example, the family number of a radio button, or whether a scroll bar is horizontal or vertical.
- A pointer to a color table used to draw items (custom color tables are used only for standard controls or controls you define yourself). Take care that your use of color conforms to the Apple *Human Interface Guidelines*.

There are several Dialog Manager procedures that, given a pointer to a dialog port and an item ID, set or return that item's text, type, display rectangle, appearance, and value.

The structure of the item template is shown in Figure 6-4.

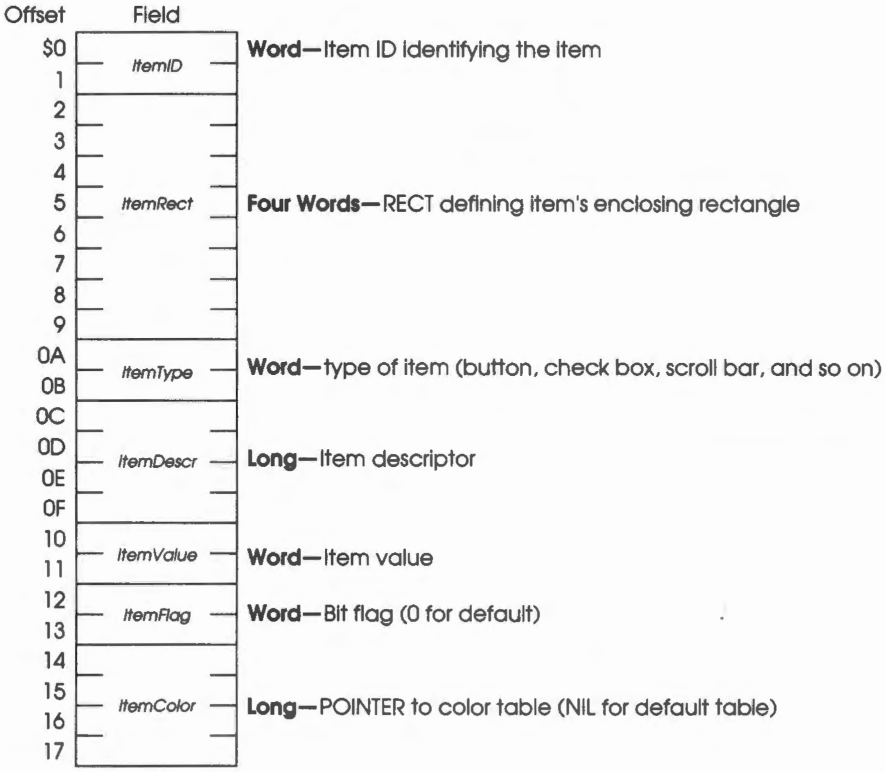
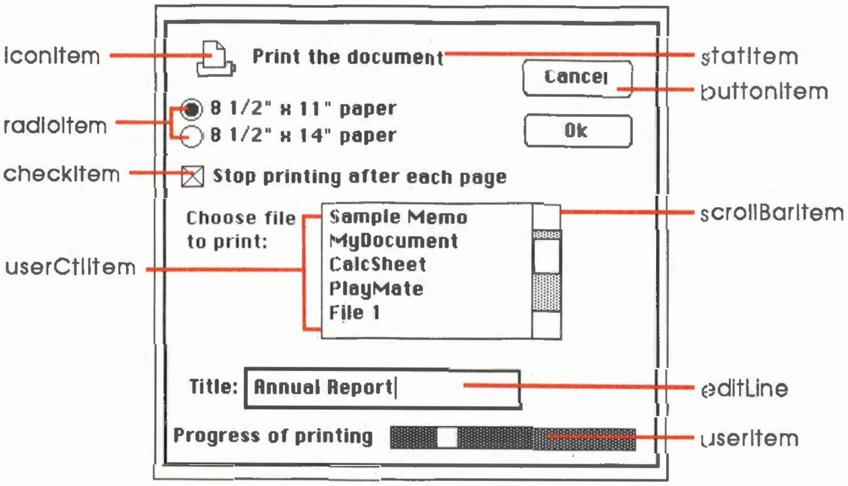

Offset
Field
$0
1
2
3
4
5
6
7
8
9
0A
0B
0C
0D
0E
0F
10
11
12
13
14
15
16
17
ItemID
ItemRect
ItemType
ItemDescr
ItemValue
ItemFlag
ItemColor
Word—Item ID identifying the item
Four Words—RECT defining item's enclosing rectangle
Word—type of item (button, check box, scroll bar, and so on)
Long—Item descriptor
Word—Item value
Word—Bit flag (0 for default)
Long—POINTER to color table (NIL for default table)

Figure 6-4
Item template

The following sections further describe the information contained in the item template.

Item types

The **item type** identifies the type of a dialog item. The type is specified by a predefined constant or combination of constants, as listed in Table 6-2. Figure 6-5 illustrates some of these item types.



Print the document

● 8 1/2" x 11" paper
○ 8 1/2" x 14" paper

☒ Stop printing after each page

Choose file to print:

Title: Annual Report

Progress of printing

Figure 6-5
Item types

Table 6-2
Dialog item types

Item type	Description
`buttonItem`	Standard button control
`checkItem`	Standard check box control
`radioItem`	Standard radio button control
`scrollBarItem`	Special scroll bar for dialog boxes
`userCtlItem`	Application-defined control
`userCtlItem2`	Application-defined control
`statText`	Static text; text cannot be edited (several lines allowed)
`longStatText`	Static text; text cannot be edited (several lines allowed)
`longStatText2`	Static text; text cannot be edited and can contain embedded commands (several lines allowed)
`editLine`	Text that can be edited (dialog boxes only); the Dialog Manager accepts text typed by the user and allows editing (one line only)
`iconItem`	Icon
`picItem`	QuickDraw II picture

Table 6-2 (continued)
Dialog item types

Item type	Description
`userItem` •	Application-defined item, such as a picture whose appearance changes (dialogs only)
`itemDisable` + *anyItem*	If itemDisable is specified for an item, the Dialog Manager doesn't let the application know about events involving that item. For example, your a pplication may not need to know every time the user types a character or clicks in an editLine item; you may need to look at the text only when the OK button is clicked. In this case, you would disable the editLine item. Standard buttons and check boxes should always be enabled so that your application will know when they've been clicked.

Important

Don't confuse *disabling* a control with making one *inactive* with the Control Manager procedure HiliteControl. When you want a control not to respond at all to being clicked, you make it inactive. An inactive control is dimmed to show that it's inactive, while a disabled control doesn't change its appearance. See Chapter 4, "Control Manager," for more information.

An editLine item may initially either show default text or be empty. Text entry and editing are handled in the conventional way, as with the LineEdit Tool Set; in fact, the Dialog Manager usually calls LineEdit routines to handle text operations.

The Apple-X, Apple-C, and Apple-V commands, respectively, cut, copy, and paste the current selection in the active editLine item, allowing you to copy and paste text between different editLine items (the cut/copy/paste mechanism preserves the space between words).

The Tab key advances the cursor to the next editLine item in the item list, wrapping around to the first item if no other items remain. In an alert box or a modal dialog box (regardless of whether it contains an editLine item), pressing the Return key has the same effect as clicking the default button; in an alert box, the default button is identified in the alert template, whereas in a modal dialog box the default item is the item in the item list whose ID number is 1 (unless specified otherwise).

Item descriptor and item value

The **item descriptor** and **item value** provide additional information about a specific dialog item, as shown in Table 6-3.

Table 6-3
Dialog item descriptors

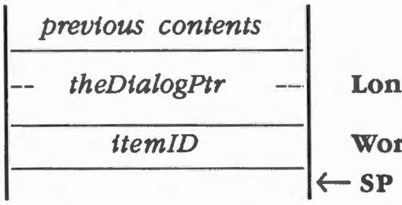
Item type	Item descriptor	Item value
`buttonItem`	Pointer to the title string	Initial value of the control
`checkItem`	Pointer to the title string	Initial value of the control
`radioItem`	Pointer to the title string	Initial value of the control
`scrollBarItem`	Pointer to dialog scroll bar action procedure	0 or default value if *itemDescr* = 0
`userCtlItem`	Pointer to control definition procedure	Initial value of the control
`userCtlItem2`	Pointer to parameter block	Initial value of the control
`statText`	Pointer to the static string	For application use
`longStatText`	Pointer to the beginning of the text	Length of the text (0 to 32767 characters)
`longStatText2`	Pointer to the beginning of the text	Length of the text (0 to 32767 characters)
`editLine`	Pointer to the default string	Maximum length of default text (0 to 255 characters)
`iconItem`	Handle to the icon	For application use
`picItem`	Handle to the picture	For application use
`userItem`	Pointer to item definition procedure	For application use

❖ *Note:* Whenever "For application use" is specified under "Item value," the value parameter is not accessed by the Dialog Manager and can be used by the application for its own purpose (use the GetDItemValue and SetDItemValue routines to change this field). For example, the application might want to store the indicator position of the userItem in Figure 6-5. Note that SetDItemValue redraws the item to display its new value.

The procedure for a userItem draws the item; for example, if the item is a clock, it draws the clock with the current time displayed. When this procedure is called, the current port will have been set by the Dialog Manager to the dialog window's `GrafPort`.

The procedure must have a dialog pointer and an item ID as input, as follows:

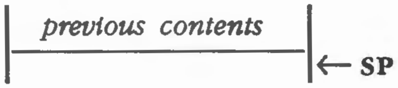
Stack before call



Long—POINTER to the dialog's GrafPort

Word—ID of item to draw

Stack after call



The *theDialogPtr* parameter is a pointer to the dialog window; if the procedure draws in more than one dialog window, this parameter tells the procedure the window in which to draw.

The *itemID* parameter is the item ID; if the procedure draws more than one item, this parameter tells it which one to draw.

For a button item, check item, or radio item, *itemDescr* is a pointer to the title of the button, check box, or radio button, and *itemValue* is the initial value of the control (this is useful with check boxes and radio buttons).

For a statText item, *itemDescr* is a pointer to a string containing the static text, and *itemValue* is not used. You can have several lines of text in the same item by inserting carriage returns (ASCII 13 = $0D) inside the string. An example of a typical string you could use for a statText item is as follows:

```
StaticStr     dc     i1'EndStaticStr-StaticStr-1'
              dc     c'Do you want to save',h'0D'
              dc     c'before quitting?',h'0D'
EndStaticStr  anop
```

If you're using APW, you can use the macro STR for a one-line static text item, as follows:

```
StaticStr     str    'File not found'
```

For a longStatText item, *itemDescr* is a pointer to the beginning of the text, and *itemValue* is the word length of the text (0 to 32767 characters). An example of typical *itemDescr* and *itemValue* parameters you would use for a longStatText item is as follows, where *itemDescr* is a pointer to the text shown in the code fragment

```
myLongText     dc     c'This is a really very...',h'0D'

               dc     c'very... very...',h'0D'

               ...

               dc     c'long text, that contains',h'0D'

               dc     c'more than 255 characters',h'0D'

               dc     c'so that I need a LongStatText',h'0D'

               dc     c'item to print it in a single item',h'0D'

EndLongText anop
```

and *itemValue* is

```
EndLongText-myLongText
```

For an editLine item, *itemDescr* is a pointer to the default string containing the default text that first appears in the item when the dialog comes up, and *itemValue* is the maximum allowed length of the editable string (0 to 255 characters). An example of typical *itemDescr* and *itemValue* parameters you would use for an editLine item is as follows, where *itemDescr* is a pointer to the string

```
EditLStr       str    'Untitled'    ; default string
```

and *itemValue* is 15 characters (the maximum length for a ProDOS filename).

If you pass 0 for *itemDescr*, the line will not contain any default text.

If the item is the first editLine item to be created, it will be the current active editLine item, and the default text (if there is any) will be selected.

For a scrollBarItem, *itemDescr* is a pointer to a special action procedure that is called during initialization and scrolling. This procedure can, for example, change the appearance of different items in the dialog in real time while the user is scrolling the scroll bar, and it will do so without reporting anything to the application. In fact, if the scrollBarItem is disabled, the application will not even know that the user clicked in it.

The definition of a dialog scroll bar action procedure is as follows:

Stack before call

previous contents	
wordspace	**Word**—Space for result
command	**Word**—See list of possible commands in Table 6-4
— *theDialogPtr* —	**Long**—POINTER to dialog the scroll bar is in
scrollBarID	**Word**—Item ID of scroll bar
	← **SP**

Stack after call

previous contents	
result	**Word**—Depends on command, as shown in Table 6-4
	← **SP**

Table 6-4
Dialog scroll bar action procedure commands

Command	Result	Comments
`getInitView`	Initial view	View size at creation (called before control is allocated)
`getInitTotal`	Initial total	Total size at creation (called before control is allocated)
`getInitValue`	Starting value	Value at creation (called before control is allocated)
`scrollLineUp`	New value	Scroll one line up and return new scroll bar value
`scrollLineDown`	New value	Scroll one line down and return new scroll bar value
`scrollPageUp`	New value	Scroll one page up and return new scroll bar value
`scrollPageDown`	New value	Scroll one page down and return new scroll bar value
`scrollThumb`	New value	Get thumb position; scroll to that position and return new correct value (usually the same)

For the commands `getInitView`, `getInitTotal`, and `getInitValue`, do not make any reference to the scroll bar control because these calls are made before the control is allocated.

The commands `scrollLineUp and scrollPageDown` should first call the GetDItemValue routine with *scrollBarID* to get the previous value of the scroll bar, then make some changes (such as changing an icon or the text of a statText item or adding or removing items from the dialog), and finally return the new value of the scroll bar.

For the `scrollThumb` command, you should first call the GetDItemValue routine with *scrollBarID*. GetDItemValue returns the new thumb position. You can then make whatever changes you want and return either the value obtained from GetDItemValue or any other suitable value.

Your scroll bar action procedure is called by NewDItem to create a scrollBarItem and by ModalDialog when the user clicks in a scrollBarItem. ModalDialog sets the new scroll bar value according to the result returned by your procedure.

For an iconItem, *itemDescr* is a handle to an icon, and *itemValue* is not used. The icon record contains the following fields:

```
iconrect       equu                     ; boundsRect (width is multiple of 8)

iconImage      equ iconRect + 8         ; pixel image of icon
```

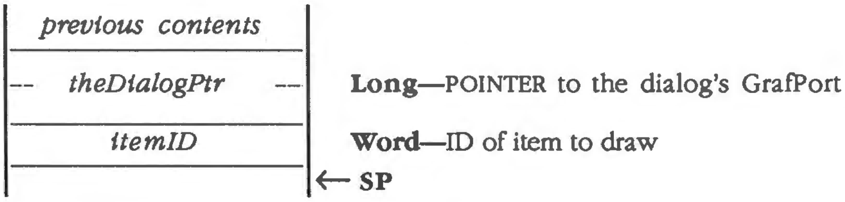
Picture items were not yet implemented at the time of publication.

For a userCtlItem, *itemDescr* is a pointer to a control definition procedure, as defined in Chapter 4, "Control Manager," and *itemValue* is the initial value of the control.

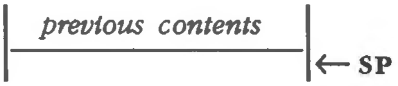
For a userItem, *itemDescr* is a pointer to an item definition procedure, and *itemValue* is not used. The definition of an item definition procedure is as follows:

MyItem

Stack before call



Stack after call



The procedure for a userItem draws the item; for example, if the item is a clock, it will draw the clock with the current time displayed. When this procedure is called, the current port will have been set by the Dialog Manager to the dialog window's GrafPort.

Display rectangle

The **display rectangle** in the item template controls how the item is displayed. You must specify the values for the upper-left corner of the display rectangle; however, if you specify (0,0) as the coordinates for the lower-right corner for simple buttons, radio buttons, or check boxes, the Dialog Manager will calculate the display rectangle for you.

- For standard controls, scroll bars, and user controls, the display rectangle becomes the control's enclosing rectangle.
- For an editLine item, the display rectangle becomes LineEdit's view rectangle. The text is clipped if there are more characters than will fit in the rectangle. In addition, the Dialog Manager uses the QuickDraw II routine FrameRect to draw a rectangle outside the display rectangle.
- Any statText, longStatText, or longStatText2 items are displayed in exactly the same way as editLine items, except that a rectangle isn't drawn outside the display rectangle, and you can display more than one line of text by inserting carriage-return characters in the text.
- The rectangle for a statText item must always be at least as wide as the first character of the text; a good rule of thumb is to make it at least 20 pixels wide. See the section "StringWidth" in Chapter 16, "QuickDraw II," for more information.
- Icons are clipped to fit the display rectangle.
- If the procedure for a userItem draws outside the item's display rectangle, the drawing is clipped to the display rectangle.

❖ *Note:* Clicking anywhere within the display rectangle is considered a click in that item. If display rectangles overlap, a click in the overlapping area is considered a click in whichever item comes first in the item list.

Item ID

Each item in an item list is identified by an **item ID,** a unique number in the list allowing you to further reference this item. In a modal dialog's item list, the item whose ID is 1 is assumed to be the dialog's **default button,** unless specified otherwise by the SetDefButton routine. If the user presses the Return key, the Dialog Manager normally returns the ID of the default button, just as when that item is actually clicked. By convention, the OK button in an alert's item list should have an ID of 1 and the Cancel button should have an ID of 2 (in fact, those numbers are given to OK and Cancel constants).

To conform with the Apple *Human Interface Guidelines,* the Dialog Manager automatically outlines the default button in bold, unless there is no default button (that is, no button item with ID 1).

❖ *Note:* If you don't want any default button, don't create an item with an ID of 1.

An item ID of 0 is invalid.

Item flag

The *itemFlag* parameter usually contains the same value as the *flag* parameter given to the Control Manager routine NewControl. The *itemFlag* parameter may also contain the family number of a radio button or information as to whether a scroll bar is horizontal or vertical. For more details, refer to Chapter 4, "Control Manager."

❖ *Note:* Don't use *itemFlag* to outline a default button in bold, because the Dialog Manager handles the concept of a default button automatically.

Item color tables

If you specify NIL for the color table in an item template, the item is drawn using the default color table. Otherwise, you can specify a custom color table for the item, as described in Chapter 16, "QuickDraw II." If you do decide to use a special color table, take care that your use of color conforms to the Apple *Human Interface Guidelines*.

Dialog records

To create a dialog, you pass information to the Dialog Manager in parameters or in a template; the Dialog Manager then incorporates that information into a **dialog record.** The dialog record contains the window record for the dialog window, a handle to the dialog's item list, and some additional fields. The Dialog Manager creates the dialog window by calling the Window Manager routine NewWindow and then setting the dialog type in the dialog record to indicate whether the dialog is modal or modeless.

The routine that creates the dialog returns a **dialog pointer** to the dialog's GrafPort; thereafter, you use that pointer to refer to the dialog in Dialog Manager routines or even in Window Manager or QuickDraw II routines. The dialog pointer is equivalent to the window pointer for the dialog box. It is not a pointer to the dialog record or even to the window record.

The Dialog Manager provides routines for handling events in the dialog window and disposing of the dialog when you're done.

The structure of a dialog record is private. You can perform all necessary operations on a dialog without accessing the fields of the dialog record directly. To get or change information about a dialog, you pass the dialog pointer to a Dialog Manager routine. Similarly, to get or change information about an item in a dialog, you pass the dialog pointer and the item ID to a Dialog Manager routine. You'll never access information directly through the handle to the item.

Alerts

When you call a Dialog Manager routine to invoke an alert, you pass the routine a pointer to the **alert template,** which contains the following:

- An alert ID used by the Dialog Manager to manage the stages between the different alerts.
- A rectangle, given in global coordinates, which determines the alert window's size and location. This becomes the portRect of the window's GrafPort. To allow for the normal menu bar and the border around the portRect, the top coordinate of the rectangle should be at least 25 pixels below the top of the screen.
- Information about exactly what should happen at each stage of the alert.
- A list of pointers to the item templates.

An alert gets its parameters from an alert template. The structure of an alert template is shown in Figure 6-6.

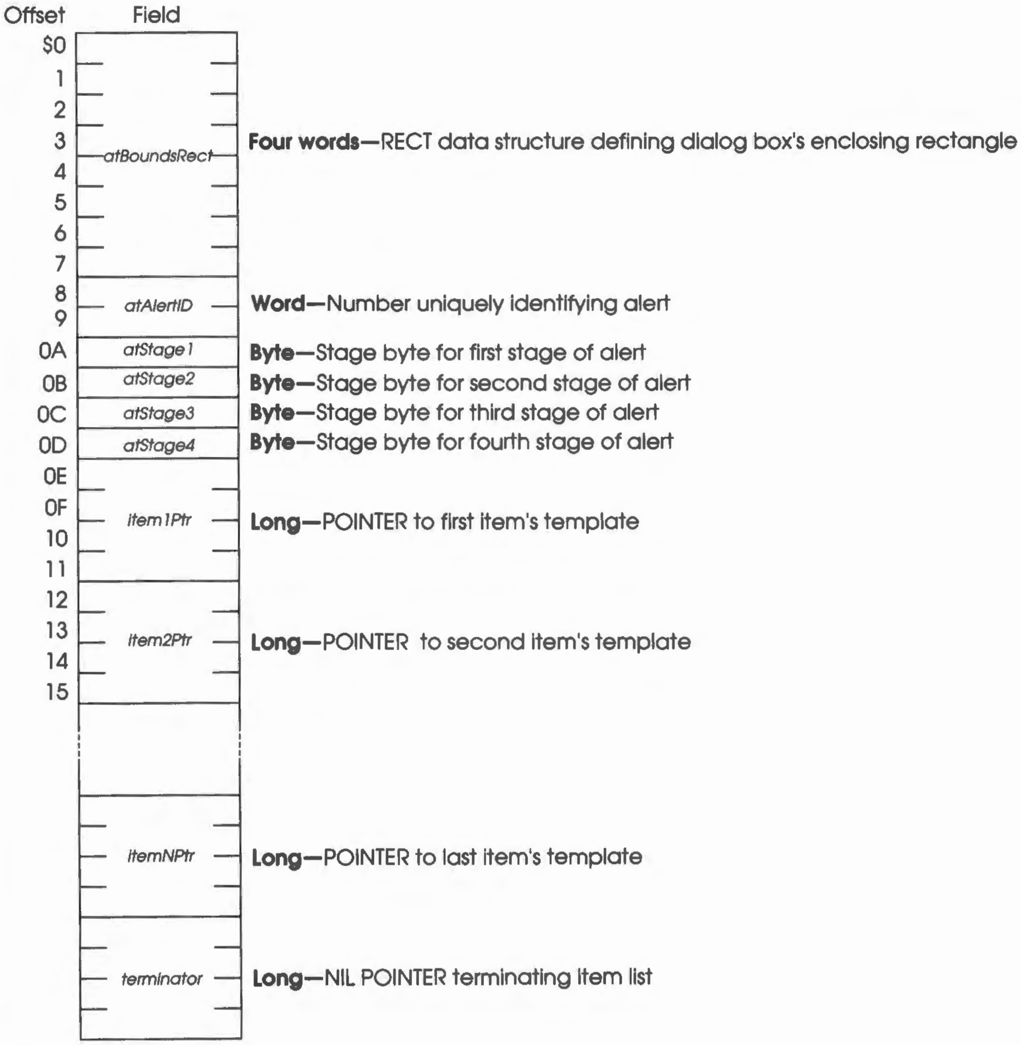

Offset
Field
$0
1
2
3
4
5
6
7
8
9
0A
0B
0C
0D
0E
0F
10
11
12
13
14
15
atBoundsRect
atAlertID
atStage1
atStage2
atStage3
atStage4
item1Ptr
item2Ptr
itemNPtr
terminator
Four words—RECT data structure defining dialog box's enclosing rectangle
Word—Number uniquely identifying alert
Byte—Stage byte for first stage of alert
Byte—Stage byte for second stage of alert
Byte—Stage byte for third stage of alert
Byte—Stage byte for fourth stage of alert
Long—POINTER to first item's template
Long—POINTER to second item's template
Long—POINTER to last item's template
Long—NIL POINTER terminating item list

Figure 6-6
Alert template

Every alert has four **alert stages,** which correspond to consecutive occurrences of the alert. The first three stages correspond to the first three occurrences, and the fourth stage includes the fourth and subsequent occurrences. (The Dialog Manager compares the current alert's ID to the last alert's ID to determine whether these indicate that it is the same alert.)

The actions for each stage are specified by the following three pieces of information:

1. Is the alert box is to be drawn?
2. Which is the default button—the OK button (or, if none, a button that will perform the command) or the Cancel button? For an alert, the OK button should have an ID 1 and the Cancel button an ID 2.
3. Which of four sounds should be emitted at this stage of the alert?

The actions taken are determined by a **stage byte,** which contains the bit fields shown in Figure 6-7.

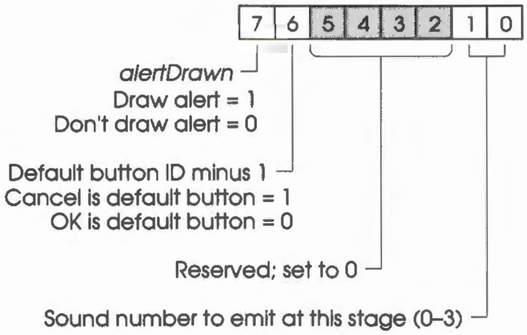


Figure 6-7
Stage byte

The **alert sounds** are determined by a sound procedure that emits one of up to four tones or sequences of tones. The sound procedure has one parameter: an integer from 0 to 3. The procedure can emit any sound for each of these numbers, which identify the sounds in the alert template. The volume of each beep depends on the current speaker volume setting, which the user can adjust with the Control Panel desk accessory.

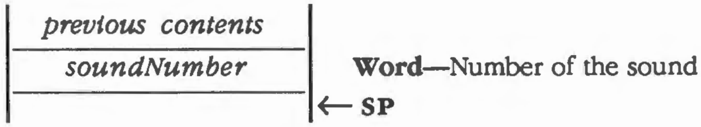
The standard sound procedure is as follows:

0	No sound
1	One short beep of a preset pitch and duration
2	Two short beeps, each of the same pitch and duration as sound number 1
3	Three short beeps, each of the same pitch and duration as sound number 1

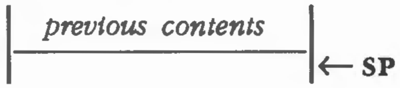
If you want other sounds besides the standard ones, write your own sound procedure and call ErrorSound to make it the current sound procedure. For example, you might declare a sound procedure named MySound as follows:

MySound

Stack before call



Stack after call



If you want two successive beeps of different pitch, you need to write a procedure that will emit that sound for a particular sound number, and you need to specify that number in the alert template. The Apple IIGS Miscellaneous Tool Set routine FWEntry allows you to be in the 16-bit environment and still call the Apple II firmware, which has routines for emitting sound (the standard sound procedure calls the BELL routine at $FBDD); for more complex sounds, you can use the Sound Tool Set. See Chapter 14, "Miscellaneous Tool Set," and Chapter 21, "Sound Tool Set," for more information.

❖ *Note:* When the Dialog Manager detects a click outside an alert box or a modal dialog box, it emits sound number 1; thus, for consistency with the Apple *Human Interface Guidelines,* sound number 1 should always be a single beep.

Internally, alerts are treated as special modal dialogs. The alert routine creates the alert window by calling NewModalDialog and every item with GetNewDItem. The Dialog Manager works with the dialog created by NewModalDialog, just as when it operates on a dialog window, but it disposes of the dialog before returning to the application. Normally your application won't change the dialog record for an alert; however, there is a way that this can happen: For any alert, you can specify a filter procedure that will be executed repeatedly during the alert, and this procedure may change the dialog. For details, see the section "Filter Procedures" in this chapter.

Using the Dialog Manager

This section discusses how the Dialog Manager routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

The Dialog Manager depends upon the presence of the tool sets shown in Table 6-5 and requires that at least the minimum version of the tool set be present.

Table 6-5
Dialog Manager—other tool sets required

Tool set number		Tool set name	Minimum version needed
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0
$03	#03	Miscellaneous Tool Set	1.0
$04	#04	QuickDraw II	1.0
$05	#05	Desk Manager	1.0
$06	#06	Event Manager	1.0
$0E	#14	Window Manager	1.3
$10	#16	Control Manager	1.3
$14	#20	LineEdit Tool Set	1.0

Your application must make the DialogStartUp call before it makes any other Dialog Manager calls. Conversely, when your application quits, it must make the DialogShutDown call.

Where appropriate in your program, call NewModalDialog, NewModelessDialog, or GetNewModalDialog to create any dialogs you need. Then call NewDItem or GetNewDItem for each new item you want to add to the dialog. When you no longer need a dialog, you'll usually call CloseDialog.

In most cases, you won't have to change the dialogs from the way they're defined at their creation. However, if you want to modify an item in a dialog, you can use one of the GetDItem*XXX* calls to get information about the item and SetDItem*XXX* to change it. In some cases it may be appropriate to call some other routine to change the item; for example, to move a control in a dialog, you would get its handle from GetControlDItem and then call the appropriate Control Manager routine. There are also two routines specifically for accessing or setting the content of a text item in a dialog box: GetIText and SetIText.

To handle events in a modal dialog, just call the ModalDialog routine after putting up the dialog box. If your application includes any modeless dialog boxes, you'll pass events to IsDialogEvent to learn whether they need to be handled as part of a dialog. If those events do need to be handled, you'll then usually call DialogSelect. Before calling DialogSelect, however, you should check whether the user has given the keyboard equivalent of a command, and you may want to check for other special cases, depending on your application. For more information about event handling, see Chapter 7, "Event Manager."

You can support the use of the standard editing commands in a modeless dialog's editText items with DlgCut, DlgCopy, DlgPaste, and DlgDelete.

A dialog box that contains editLine items normally is displayed with the insertion point in the first such item in its item list. You may instead want to display a dialog box with text selected in an editLine item or cause an insertion point or text selection to reappear after the user has made an error in entering text. For example, the user who accidentally types nonnumeric characters when a number is required can be given the opportunity to type the entry again. The SelectIText routine makes this possible.

To invoke a particular alert, call one of the alert routines: StopAlert, NoteAlert, or CautionAlert for one of the standard kinds of alert, or Alert for an alert with something other than a standard icon (or nothing at all) in its upper-left corner.

You can find out what the current default button is by calling the GetDefButton routine on the dialog pointer for the alert passed to your filter procedure.

You can substitute text in statText items with text that you specify in the ParamText routine. This means, for example, that a document name supplied by the user can appear in an error message.

By calling the HideDItem routine, you can make an item invisible. This technique can be useful, for example, if your application needs to display a number of similar dialog boxes with one item missing or different in some of them. You can use a single dialog box in which the item or items that aren't currently relevant are invisible. To hide an item or make one reappear, use the HideDItem or ShowDItem routines. Note the following, however:

- When you want to change text in a statText item, you will find the Dialog Manager routine ParamText (described later in this chapter) easier to use than hiding the item and showing it again.
- Instead of making an item invisible and visible, you can use the RemoveDItem routine to completely remove the item from the item list.

If you want the font in your dialog and alert windows to be other than the system font, call SetDAFont to change the font.

Filter procedures

Filter procedures allow you to control the type of events handled by Dialog Manager routines. Many Dialog Manager routines allow you to specify a custom filter procedure by including a *filterProcPtr* parameter to that procedure.

If you specify NIL as the *filterProcPtr*, the standard filter procedure is executed. The standard filter procedure performs the following actions:

- Causes the Return key to have the same effect as clicking the default button
- Supports the Apple-X/C/V commands for cut/copy/paste operations inside the dialog

If *filterProcPtr* isn't NIL, the calling routine filters events by executing the procedure it points to.

❖ *Note:* If you set bit 31 of the *filterProcPtr* parameter to 1 before passing it to the calling routine, the standard filter procedure will also be called after your filter procedure. This allows you to define a custom filter procedure and still get the benefits of the default Return key and the cut/copy/paste feature for consistency with the Apple *Human Interface Guidelines*.

Your filterProc procedure should have three parameters and return a Boolean value. For example, this is how it would be declared if it were named MyFilter:

MyFilter

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *theDialogPtr* --	**Long**—POINTER to the dialog's GrafPort
-- *eventPtr* --	**Long**—POINTER to the event
-- *itemHitPtr* --	**Long**—POINTER to the item hit
	← **SP**

Stack after call

previous contents	
ignoreFlag	**Word**—BOOLEAN; TRUE if event is to be ignored,
	← **SP** FALSE if calling routine should handle event

An *ignoreFlag* value of FALSE tells the calling routine to handle the event, which either can be changed to simulate a different event, or sent through unchanged.

You could use the filterProc procedure, for example, to treat a typed character in a special way (such as to ignore it or make it have the same effect as another character or as clicking a button); in this case, the function would test for a key event with that character. As another example, suppose the dialog box contains a userItem whose procedure draws a clock with the current time displayed. The filterProc procedure can call that procedure and return FALSE without altering the current event.

$0115 DialogBootInit

Initializes the Dialog Manager; called only by the Tool Locator.

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 Call must not be made by an application.

$0215 DialogStartUp

Starts up the Dialog Manager for use by an application.

Important

Your application must make this call before it makes any other Dialog Manager calls.

DialogStartup performs the following initialization:

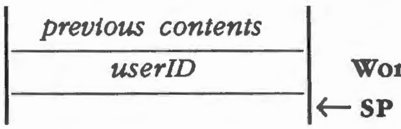
- Installs the standard sound procedure
- Passes empty strings to ParamText
- Sets the dialog font to the System font
- Sets the alert stage to 1

Important

The Dialog Manager shares its direct page with the Control Manager, so it does not need a special direct page. However, the Control Manager must be present for the Dialog Manager to run, even if the application does not use any standard control items in its dialogs.

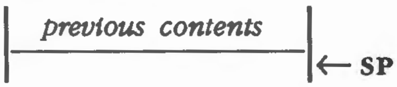
Parameters

Stack before call



Word—ID number of the application

Stack after call



Errors

None

C

```
extern pascal void DialogStartUp(userID)

Word     userID;
```

$0315 DialogShutDown

Shuts down the Dialog Manager and frees any memory allocated by the Dialog Manager.

Important

If your application has started up the Dialog Manager, the application must make this call before it quits.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

None

C

```
extern pascal void DialogShutDown()
```

$0415 DialogVersion

Returns the version number of the Dialog Manager.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
versionInfo	**Word**—Version number of the Dialog Manager
	← **SP**

Errors

None

C

```
extern pascal Word DialogVersion()
```

$0515 DialogReset

Resets the Dialog Manager; called only when the system is reset.

Warning

An application must never make this call.

DialogReset resets the dialog font to the system font, clears the strings set by the ParamText routine, resets the sound procedure to the standard sound procedure, and resets the alert stage to 1.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

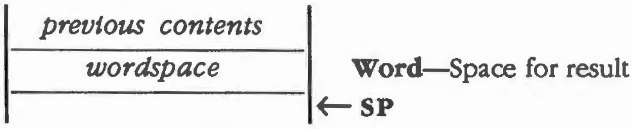
C Call must not be made by an application.

$0615 DialogStatus

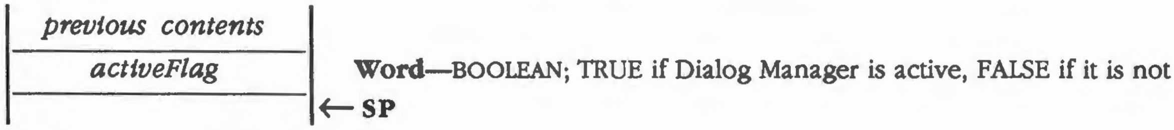
Indicates whether the Dialog Manager is active.

Parameters

Stack before call



Stack after call


previous contents
activeFlag
Word—BOOLEAN; TRUE if Dialog Manager is active, FALSE if it is not
← **SP**

Errors None

C `extern pascal Boolean DialogStatus()`

$1715 Alert

Invokes an alert defined by a specified alert template. The routine calls the current sound procedure, if any, passing it the sound number specified in the alert template for this stage of the alert. If no alert box is to be drawn at this stage, Alert returns a function result of –1; otherwise, it takes the following actions:

1. Creates and displays the alert window for this alert
2. Draws the alert box
3. Waits for the user to select an item
4. Removes the alert box
5. Returns with the ID of the item hit

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
— *alertTemplatePtr* —	**Long**—POINTER to an alert template
— *filterProcPtr* —	**Long**—POINTER to filter procedure; NIL for standard filter
	← **SP**

Stack after call

previous contents	
itemHit	**Word**—ID of item hit (minus 1 if not drawn)
	← **SP**

Errors

None

C

```
extern pascal Word Alert(alertTemplatePtr,filterProcPtr)

AlertTempPtr     alertTemplatePtr;

WordProcPtr     filterProcPtr;
```

(continued)

Alert template

Alert gets its parameters from an alert template. The definition of an alert template is shown in Figure 6-8.

Offset	Field	
$0 1 2 3 4 5 6 7	*atBoundsRect*	**Four words**—RECT data structure defining dialog box's enclosing rectangle
8 9	*atAlertID*	**Word**—Number uniquely identifying alert
0A	*atStage1*	**Byte**—Stage byte for first stage of alert (see Figure 6-9)
0B	*atStage2*	**Byte**—Stage byte for second stage of alert (see Figure 6-9)
0C	*atStage3*	**Byte**—Stage byte for third stage of alert (see Figure 6-9)
0D	*atStage4*	**Byte**—Stage byte for fourth stage of alert (see Figure 6-9)
0E 0F 10 11	*item1Ptr*	**Long**—POINTER to first item's template; item template is defined in the section "Item Template" in this chapter
12 13 14 15	*item2Ptr*	**Long**—POINTER to second item's template
	itemPtr	**Long**—POINTER to last item's template
	terminator	**Long**—NIL POINTER terminating item list

Figure 6-8
Alert template

A stage byte is a bit flag containing the bit fields shown in Figure 6-9.

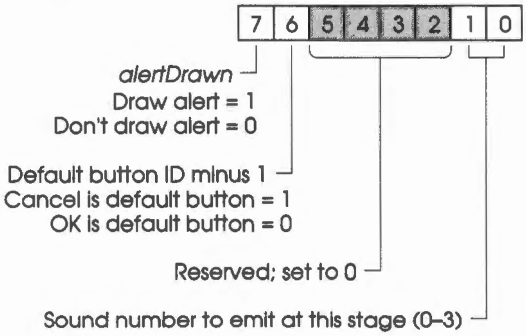


Figure 6-9
Stage byte

❖ *Note:* The Alert routine creates the alert window by calling NewModalDialog and GetNewDItem for each item in the alert, and it performs the rest of its processing by calling ModalDialog.

The Alert routine repeatedly gets and handles events in the alert window until an enabled item is clicked, at which time it returns the item ID. Normally you'll then do whatever is appropriate in response to a click of that item.

Alert gets each event by calling the Event Manager routine GetNextEvent. If the event is a mouse-down event outside the content region of the alert window, Alert emits sound number 1 (which should be a single beep) and gets the next event; otherwise, it filters and handles the event as described next.

If you specify NIL for *filterProcPtr,* the standard filter procedure is executed. If *filterProcPtr* isn't NIL, the calling routine filters events by executing the procedure it points to.

❖ *Note:* If you set bit 31 of the *filterProcPtr* parameter to 1 before passing it to the calling routine, the standard filter procedure will also be called after your filter procedure. This allows you to define a custom filter procedure and still get the benefits of the default Return key and the cut/copy/paste feature for consistency with the Apple *Human Interface Guidelines*.

(continued)

Alert handles the events for which the filter procedure returns FALSE as follows:

- If the mouse button is pressed in a control, Alert calls the Control Manager routine TrackControl. If the mouse button is released inside the control and the control is enabled, Alert returns; otherwise, it does nothing.
- If the mouse button is pressed in any other enabled item, Alert simply returns. If it's pressed in any other disabled item or in no item, or if any other event occurs, Alert does nothing.

Before returning to the application with the item number, Alert removes the alert box from the screen. (Alert disposes of the alert window and its associated data structures, the item list, and the items.)

❖ *Note:* The Alert routine's removal of the alert box would not be the desired result if the user clicked a check box or radio button; however, alerts normally contain only static text, icons, pictures, and buttons that are supposed to make the alert box go away. If your proposed alert box contains other items besides these, consider whether it might be more appropriate as a dialog box.

$1A15 CautionAlert

Invokes an alert defined by a specified alert template and draws the caution icon in the upper-left corner of the box. The caution icon is shown in Figure 6-10.

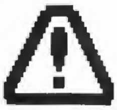


Figure 6-10
Caution icon

The alert template is defined in Figure 6-8 under the Alert routine.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *alertTemplatePtr* --	**Long**—POINTER to an alert template
-- *filterProcPtr* --	**Long**—POINTER to filter procedure; NIL for standard filter
	← **SP**

Stack after call

previous contents	
itemHit	**Word**—ID of item hit
	← **SP**

Errors

None

C

```
extern pascal Word CautionAlert(alertTemplatePtr,filterProcPtr)
AlertTempPtr    alertTemplatePtr;
WordProcPtr     filterProcPtr;
```

$0C15 CloseDialog

Removes a specified dialog window from the screen and deletes it from the window list.

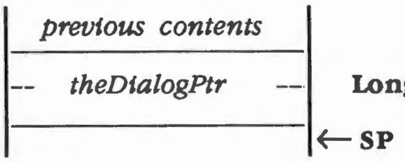
The routine releases the memory occupied by the following:

- The data structures associated with the dialog window (such as the window's structure, content, and update regions)
- All of the items in the dialog box (except for pictures and icons) and any data structures associated with them

❖ *Note:* CloseDialog does not affect the memory space allocated for your own structures, such as dialog, alert, and item templates

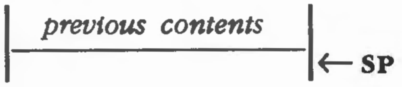
Parameters

Stack before call



Long—POINTER to the dialog's GrafPort

Stack after call



Errors

Window Manager errors	Returned unchanged

C

```
extern pascal void CloseDialog(theDialogPtr)
GrafPortPtr     theDialogPtr;
```

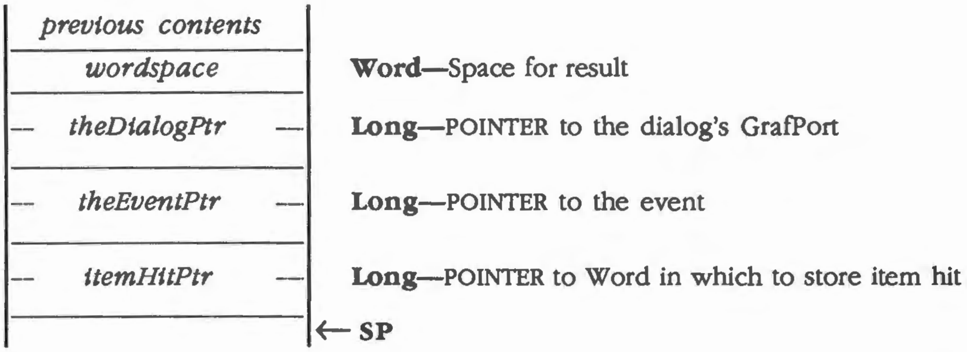
$3615 DefaultFilter

Calls the standard default filter used by the ModalDialog or Alert routines when no user filter procedure is specified. Given a pointer to an event involving dialog items, DefaultFilter filters the Apple-X, Apple-C, and Apple-V keys to make them cut, copy, and paste. The routine also interprets the Return key as a click in the default button.

DefaultFilter returns a TRUE result if the default button has been clicked and the item hit (pointed to by *itemHitPtr*) contains the button's ID number, or it returns TRUE if a cut/copy/paste operation has been performed on an enabled editLine item.

Parameters

Stack before call



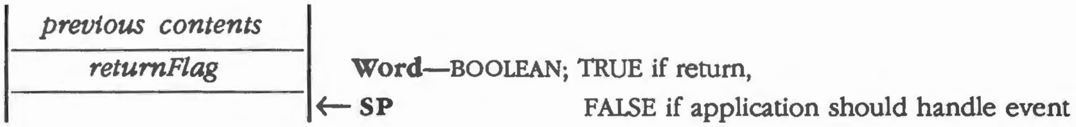
Word—Space for result

Long—POINTER to the dialog's GrafPort

Long—POINTER to the event

Long—POINTER to Word in which to store item hit

Stack after call



Word—BOOLEAN; TRUE if return,
FALSE if application should handle event

Errors

None

C

```
extern pascal Boolean DefaultFilter(theDialogPtr,theEventPtr,itemHitPtr)

GrafPortPtr     theDialogPtr;

EventRecordPtr     theEventPtr;

Word   *itemHitPtr;
```

$1115 DialogSelect

Handles an event as part of a specified modeless dialog. You'll normally call DialogSelect when the IsDialogEvent routine returns TRUE, passing the event in the event record pointed to by *theEventPtr*.

If the event involves an enabled dialog item, DialogSelect returns a result of TRUE with the dialog pointer stored at the location pointed to by *resultPtr* and the item ID stored at the location pointed to by *itemHitPtr*. Otherwise, the routine returns FALSE with *resultPtr* and *itemHitPtr* undefined. Normally when DialogSelect returns TRUE, you'll do whatever is appropriate as a response to the event, and when it returns FALSE you'll do nothing.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
— *theEventPtr* —	**Long**—POINTER to the event record
— *resultPtr* —	**Long**—POINTER to Long in which to store POINTER to dialog's GrafPort
— *itemHitPtr* —	**Long**—POINTER to Word in which to store ID of item hit
	← **SP**

Stack after call

previous contents	
enabledFlag	**Word**—BOOLEAN; TRUE if event involved an enabled item, FALSE if not
	← **SP**

Errors

None

C

```
extern pascal Boolean DialogSelect(theEventPtr,resultPtr,itemHitPtr)

EventRecordPtr     theEventPtr;

GrafPortPtr *resultPtr;

Word  *itemHitPtr;
```

More about DialogSelect and events

The actions that DialogSelect takes in response to events are as follows:

- For an activate or update event for a dialog window, DialogSelect activates or updates the window and returns FALSE.
- For a key-down or auto-key event with the Apple key held down, DialogSelect returns FALSE.
- For a mouse-down event in an editLine item, DialogSelect responds as appropriate (displaying a caret at the insertion point or selecting text), and it returns TRUE if the editLine item is enabled or FALSE if it's disabled.
- For a key-down or auto-key event occurring without the Apple key being held down and with an editLine item present, text entry and editing are handled in the standard way. DialogSelect returns TRUE if the editLine item is enabled or FALSE if it's disabled.
- For a key-down or auto-key event with no editLine item present, DialogSelect returns FALSE.

❖ *Note:* To treat a typed character in a special way (such as to ignore it or to give it the same effect as another character or as clicking a button), you need to check for a key-down event with that character before calling DialogSelect.

- For a mouse-down event in a control, DialogSelect calls the Control Manager routine TrackControl. If the mouse button is released inside the control and the control is enabled, DialogSelect returns TRUE; otherwise, it returns FALSE.
- For a mouse-down event in any other enabled item, DialogSelect returns TRUE.
- For a mouse-down event in any other disabled item or in no item, or for any other event, DialogSelect returns FALSE.

❖ *Note:* If the event isn't one that DialogSelect specifically checks for (if it's a null event, for example), and if an editLine item is present in the dialog, DialogSelect calls the LineEdit Tool Set routine LEIdle to make the cursor blink.

$3915 DisableDItem

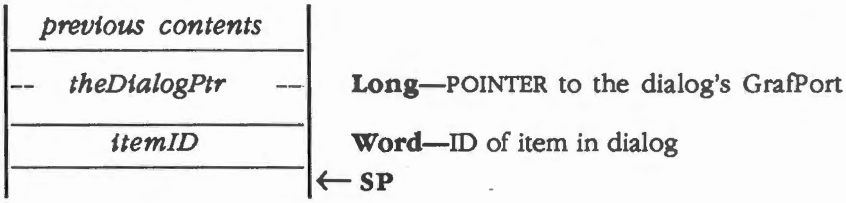
Disables a specified item in a specified dialog. If the item is already disabled, DisableDItem does nothing.

Important

Disabled is different from *deactivated*. If you do not want a control to respond at all when the user clicks it, deactivate the control with the Control Manager routine HiliteControl.

Parameters

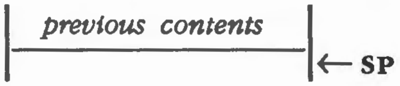
Stack before call



Long—POINTER to the dialog's GrafPort

Word—ID of item in dialog

Stack after call



Errors

$150C	`itemNotFound`	No such item

C

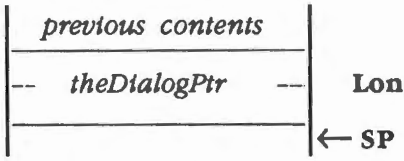
```
extern pascal void DisableDItem(theDialogPtr,itemID)

GrafPortPtr     theDialogPtr;

Word      itemID;
```

$1315 DlgCopy

Checks whether a specified dialog has any editLine items and, if so, applies the LineEdit routine LECopy to the current editLine item. You can call DlgCopy to handle the Copy editing command when a modeless dialog window is active.

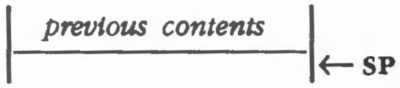
Parameters

Stack before call



Long—POINTER to the dialog's GrafPort

Stack after call



Errors

None

C

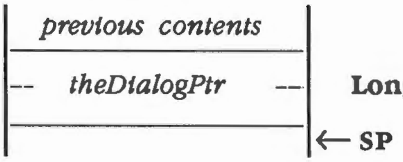
```
extern pascal void DlgCopy(theDialogPtr)

Pointer     theDialogPtr;
```

$1215 DlgCut

Checks whether a specified dialog has any editLine items and, if so, applies the LineEdit routine LECut to the current editLine item. You can call DlgCut to handle the Cut editing command when a modeless dialog window is active.

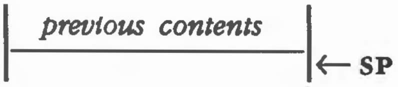
Parameters

Stack before call



Long—POINTER to the dialog's GrafPort

Stack after call



Errors

None

C

```
extern pascal void DlgCut(theDialogPtr)

GrafPortPtr     theDialogPtr;
```

$1515 DlgDelete

Checks whether a specified dialog has any editLine items and, if so, applies the LineEdit routine LEDelete to the current editLine item. You can call DlgDelete to handle the Clear editing command when a modeless dialog window is active.

Parameters

Stack before call

```
|  previous contents  |
|---------------------|
|-- theDialogPtr   -- |   Long—POINTER to the dialog's GrafPort
|---------------------|
|                     |<— SP
```

Stack after call

```
|  previous contents  |
|---------------------|<— SP
```

Errors

None

C

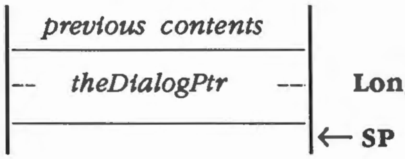
```
extern pascal void DlgDelete(theDialogPtr)

GrafPortPtr    theDialogPtr;
```

$1415 DlgPaste

Checks whether a specified dialog has any editLine items and, if so, applies the LineEdit routine LEPaste to the current editLine item. You can call DlgPaste to handle the Paste editing command when a modeless dialog window is active.

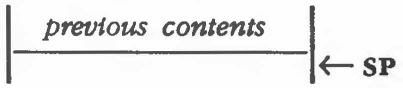
Parameters

Stack before call



Long—POINTER to the dialog's GrafPort

Stack after call



Errors

None

C

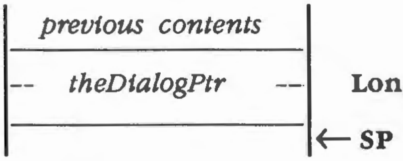
```
extern pascal void DlgPaste(theDialogPtr)

GrafPortPtr     theDialogPtr;
```

$1615 DrawDialog

Draws the contents of a specified dialog box. Since the normal sequence of IsDialogEvent and DialogSelect handles dialog window updating, this procedure is useful only in unusual situations. You would call it, for example, to display a dialog box that doesn't require any response but merely tells the user what's going on during a time-consuming process.

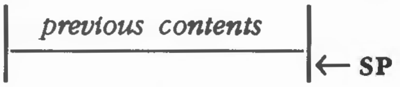
Parameters

Stack before call



Long—POINTER to the dialog's GrafPort

Stack after call



Errors

None

C

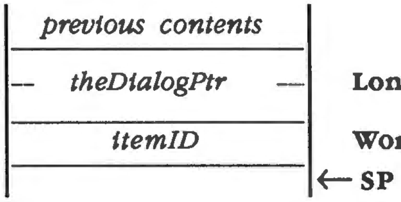
```
extern pascal void DrawDialog(theDialogPtr)

GrafPortPtr     theDialogPtr;
```

$3A15 EnableDItem

Enables a specified item in a specified dialog. If the item is already enabled, EnableDItem does nothing.

Parameters

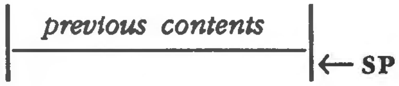
Stack before call



Long—POINTER to the dialog's GrafPort

Word—ID of item in dialog

Stack after call



Errors

$150C	`itemNotFound`	No such item

C

```
extern pascal void EnableDItem(theDialogPtr,itemID)

GrafPortPtr     theDialogPtr;

Word     itemID;
```

$0915 ErrorSound

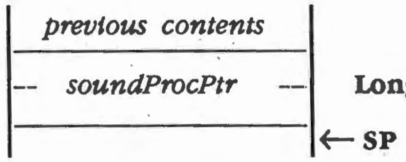
Establishes the sound procedure for alerts.

If you don't call ErrorSound, or if you pass NIL for *soundProcPtr,* the Dialog Manager uses the standard sound procedure. (For details, see the section "Alerts" in this chapter.)

❖ *Note:* The sound procedure is also called by the ModalDialog routine with a sound number of 1 when the user clicks outside the dialog box.

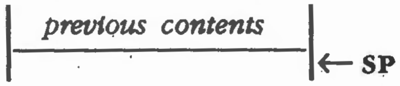
Parameters

Stack before call



Long—POINTER to sound procedure; NIL for standard sound

Stack after call



Errors

None

C

```
extern pascal void ErrorSound(soundProcPtr)

VoidProcPtr     soundProcPtr;
```

$2415 FindDItem

Returns the ID of the item located at a specified point in a specified dialog. The point must be expressed in global coordinates.

If there is no item at the location or if the specified point is outside of the specified dialog, FindDItem returns 0.

Important
The *thePoint* parameter must be expressed in global coordinates.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *theDialogPtr* --	**Long**—POINTER to the dialog's GrafPort
-- *thePoint* --	**Long**—POINT in global coordinates
	← **SP**

Stack after call

previous contents	
itemHit	**Word**—ID of item at *thePoint;* 0 if no item or point not in dialog
	← **SP**

Errors

None

C

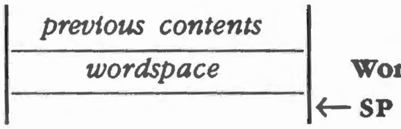
```
extern pascal Word FindDItem(theDialogPtr,thePoint)

GrafPortPtr     theDialogPtr;

Point     thePoint;
```

$3415 GetAlertStage

Returns the stage of the last occurrence of an alert as a number from 0 to 3.

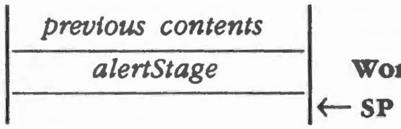
Parameters

Stack before call



Word—Space for result

Stack after call



Word—Current stage of the alert

Errors

None

C

```
extern pascal Word GetAlertStage()
```

$1E15 GetControlDItem

Returns a handle to the control record for a specified item. You can then make calls to the Control Manager to change the behavior of this item.

❖ *Note:* Dialog Manager calls are provided to change the attributes of items. Whenever possible, use those calls instead of Control Manager calls.

Important
Be very careful when you use GetControlDItem. By using the Control Manager, you bypass the Dialog Manager and can destroy data used by the Dialog Manager. However, it is safe to use GetControlDItem on standard controls (such as buttons, check boxes, and radio buttons). It is less safe to use it with dialog scroll bars, and it is definitely unsafe to use it with text items. Do not use it to change the *ctlRefCon* field in the control record of any control.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *theDialogPtr* --	**Long**—POINTER to the dialog's GrafPort
itemID	**Word**—Unique number identifying the item
	← **SP**

Stack after call

previous contents	
--*theControlHandle* --	**Long**—HANDLE to the item's control record
	← **SP**

Errors

$150C `itemNotFound` No such item

C

```
extern pascal CtlRecHndl GetControlDItem(theDialogPtr,itemID)

GrafPortPtr     theDialogPtr;

Word      itemID;
```

$3715 GetDefButton

Returns the ID of the default button item in a specified dialog. If the dialog does not contain any default button, GetDefButton returns 0.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *theDialogPtr* --	**Long**—POINTER to the dialog's GrafPort
	← **SP**

Stack after call

previous contents	
defButtonID	**Word**—ID of dialog default button, or 0
	← **SP**

Errors

None

C

```
extern pascal Word GetDefButton(theDialogPtr)
GrafPortPtr     theDialogPtr;
```

$2815 GetDItemBox

Returns the display rectangle of a specified item.

Parameters

Stack before call

previous contents	
-- *theDialogPtr* --	**Long**—POINTER to the dialog
itemID	**Word**—ID of item in dialog
-- *itemBoxPtr* --	**Long**—POINTER to 8 bytes in which to store the rectangle
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$150C	`itemNotFound`	No such item

C

```
extern pascal void GetDItemBox(theDialogPtr,itemID,itemBoxPtr)
GrafPortPtr     theDialogPtr;
Word      itemID;
Rect *itemBoxPtr;
```

$2615 GetDItemType

Returns the type of a specified item (buttonItem, radioItem, statText, and so on). If the item is currently disabled, the returned value is the type plus `itemDisable`. See the section "Item Types" in this chapter.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *theDialogPtr* --	**Long**—POINTER to the dialog
itemID	**Word**—ID of item in dialog
	← **SP**

Stack after call

previous contents	
itemType	**Word**—Type of item, including `itemDisable`
	← **SP**

Errors

$150C	`itemNotFound`	No such item

C

```
extern pascal Word GetDItemType(theDialogPtr,itemID)

GrafPortPtr     theDialogPtr;

Word      itemID;
```

$2E15 GetDItemValue

Returns the current value of a specified item.

For standard controls, *itemValue* is the current value of the control. For other types of items, *itemValue* may have special meaning, as follows:

- For a longStatText or longStatText2 item, the value is the length of the text.
- For a userItem, iconItem, or statText item, the value is reserved for the application's use.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
— *theDialogPtr* —	**Long**—POINTER to the dialog's GrafPort
itemID	**Word**—ID of item in dialog
	← **SP**

Stack after call

previous contents	
itemValue	**Word**—Current value of item
	← **SP**

Errors

$150C	`itemNotFound`	No such item

C

```
extern pascal Word GetDItemValue(theDialogPtr,itemID)
GrafPortPtr    theDialogPtr;
Word     itemID;
```

$2A15 GetFirstDItem

Returns the ID of the first item in a specified dialog. If there is no item in the dialog (for example, immediately following a NewModalDialog or NewModelessDialog call), GetFirstDItem returns 0.

Warning

You must not have any item with an ID of 0.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *theDialogPtr* --	**Long**—POINTER to the dialog's GrafPort
	← **SP**

Stack after call

previous contents	
firstItem	**Word**—ID of first item in dialog, or 0 if none
	← **SP**

Errors

None

C

```
extern pascal Word GetFirstDItem(theDialogPtr)
GrafPortPtr     theDialogPtr;
```

$1F15 GetIText

Returns the text of a specified statText or editLine item in a specified dialog box.

Important
Sufficient space for the returned text must be allocated before you call GetIText.

Parameters

Stack before call

previous contents	
-- *theDialogPtr* --	**Long**—POINTER to the dialog's GrafPort
itemID	**Word**—ID of item in dialog
-- *resultPtr* --	**Long**—POINTER to space in which to place the text
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$150A	`badItemType`	Inappropriate item type; only statText and editLine allowed
$150C	`itemNotFound`	No such item

C

```
extern pascal void GetIText(theDialogPtr,itemID,resultPtr)

GrafPortPtr     theDialogPtr;

Word     itemID;

Pointer     resultPtr;
```

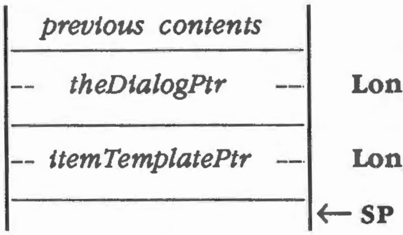
$3315 GetNewDItem

Adds a new item to a specified dialog's item list using a template.

Important
You must not have any item with an ID of 0.

Parameters

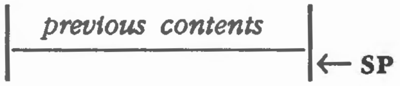
Stack before call



Long—POINTER to the dialog's GrafPort

Long—POINTER to an item template (see Figure 6-11)

Stack after call



Errors

$150A	`badItemType`	Inappropriate item type
$150B	`newItemFailed`	Item creation failed

C

```
extern pascal void GetNewDItem(theDialogPtr,itemTemplatePtr)

GrafPortPtr     theDialogPtr;

ItemTempPtr     itemTemplatePtr;
```

(continued)

Item template

Like the NewDItem routine, GetNewDItem adds a new item to a specified dialog's item list. However, instead of getting its parameters from the stack, GetNewDItem retrieves its parameters from a template whose definition is shown in Figure 6-11.

Most of the item template fields are the same as those you would pass to NewDItem, except as follows:

- The *itemRect* field contains the actual RECT definition of the display rectangle, not a pointer to it.
- The dialog that will contain the item is not specified in the template. This allows you to use dialog-independent items (such as OK and Cancel buttons) and repeat them during several dialogs.

See the section "Item Templates" in this chapter for a description of the parameters.

Offset	Field	
$0	*itemID*	**Word**—Item ID identifying item
1		
2	*itemRect*	**Four Words**—RECT defining item's enclosing rectangle
3		
4		
5		
6		
7		
8		
9		
0A	*itemType*	**Word**—Type of item (button, check box, scroll bar, and so on)
0B		
0C	*itemDescr*	**Long**—Item descriptor
0D		
0E		
0F		
10	*itemValue*	**Word**—Item value
11		
12	*itemFlag*	**Word**—Bit flag (0 for default)
13		
14	*itemColor*	**Long**—POINTER to color table; NIL for default table
15		
16		
17		

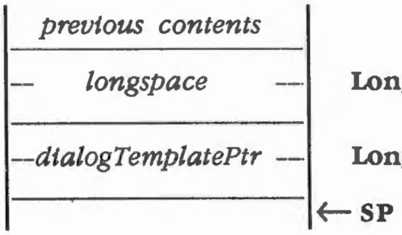
Figure 6-11
Item template

$3215 GetNewModalDialog

Creates a modal dialog and returns a pointer to the port of the new dialog. However, instead of getting its parameters from the stack, the routine gets them from a dialog template.

Parameters

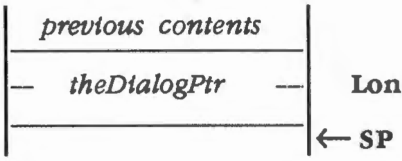
Stack before call



Long—Space for result

Long—POINTER to a dialog template (see Figure 6-12)

Stack after call



Long—POINTER to the dialog's GrafPort; NIL if error

Errors

Memory Manager errors Returned unchanged

C

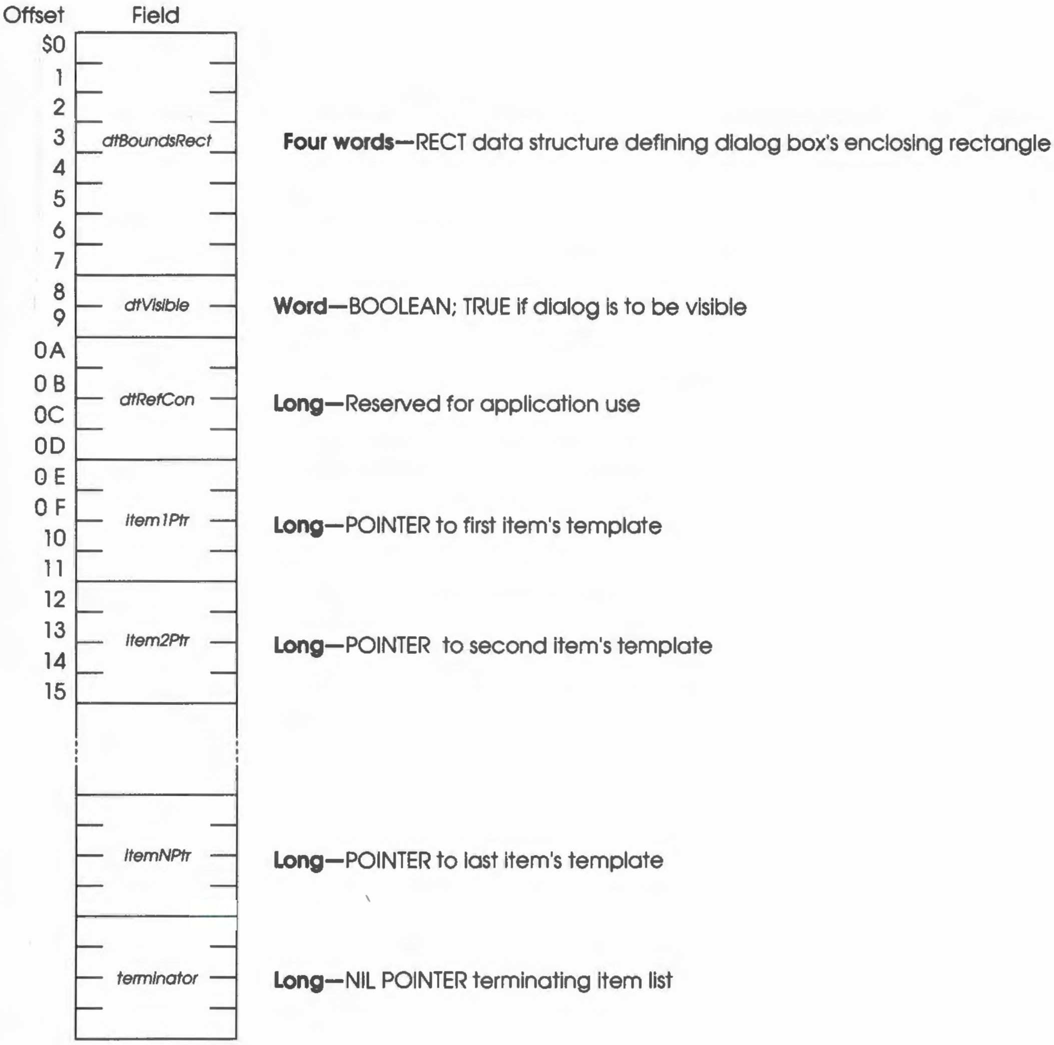
```
extern pascal GrafPortPtr GetNewModalDialog(dialogTemplatePtr)

DlgTempPtr     dialogTemplatePtr;
```

(continued)

Dialog template definition

The beginning of a dialog template contains the same values you would pass to NewModalDialog, except that *boundsRect* is the actual rectangle, not a pointer.

The *item1, item2,... itemN* fields are pointers to item templates for each of the items to include in the dialog. The last pointer must be 0 to signal the end of the list. See the section "Item Templates" in this chapter for description of the item template.



Offset	Field	
$0–7	*dtBoundsRect*	**Four words**—RECT data structure defining dialog box's enclosing rectangle
8–9	*dtVisible*	**Word**—BOOLEAN; TRUE if dialog is to be visible
0A–0D	*dtRefCon*	**Long**—Reserved for application use
0E–11	*item1Ptr*	**Long**—POINTER to first item's template
12–15	*item2Ptr*	**Long**—POINTER to second item's template
	itemNPtr	**Long**—POINTER to last item's template
	terminator	**Long**—NIL POINTER terminating item list

Figure 6-12
Dialog template

$2B15 GetNextDItem

Returns the ID of the next item in a specified dialog after a specified item. If the item is the last item in the dialog, GetNextDItem returns 0.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *theDialogPtr* --	**Long**—POINTER to the dialog's GrafPort
itemID	**Word**—ID of item in dialog
	← **SP**

Stack after call

previous contents	
nextItem	**Word**—ID of next item in dialog, 0 if no more items
	← **SP**

Errors

None

C

```
extern pascal Word GetNextDItem(theDialogPtr,itemID)

GrafPortPtr     theDialogPtr;

Word     itemID;
```

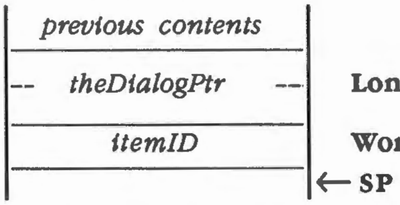
$2215 HideDItem

Erases a specified item from a specified dialog. The item is not removed from the item list and can be displayed again by calling the ShowDItem routine.

If the item is already invisible, HideDItem does nothing.

Parameters

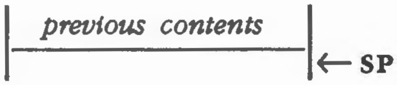
Stack before call



Long—POINTER to the dialog's GrafPort

Word—ID of item in dialog

Stack after call



Errors

$150C	`itemNotFound`	No such item

C

```
extern pascal void HideDItem(theDialogPtr,itemID)

GrafPortPtr     theDialogPtr;

Word     itemID;
```

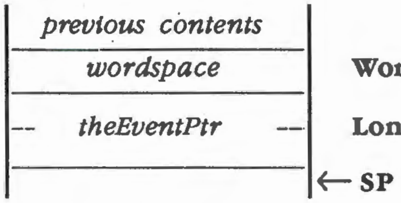
$1015 IsDialogEvent

Determines whether a specified event needs to be handled as part of a modeless dialog. If your application includes any modeless dialogs, call IsDialogEvent after calling the Event Manager routine GetNextEvent or the Window Manager routine TaskMaster. For more information about events, see Chapter 7, "Event Manager."

Important
If your modeless dialog contains any editLine items, you must call IsDialogEvent (and then DialogSelect), even if GetNextEvent returns FALSE; otherwise, your dialog won't receive null events and the cursor won't blink.

Parameters

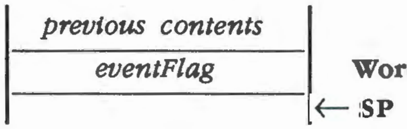
Stack before call



Word—Space for result

Long—POINTER to the event record

Stack after call



Word—BOOLEAN; TRUE if the event is a Dialog Event, FALSE if not

Errors

None

C

```
extern pascal Boolean IsDialogEvent(theEventPtr)

EventRecordPtr     theEventPtr;
```

(continued)

More about IsDialogEvent and events

If the event is an activate or update event for a dialog window, a mouse-down event in the content region of an active dialog window, or any other type of event when a dialog window is active, IsDialogEvent returns TRUE; otherwise, it returns FALSE.

When FALSE is returned, handle the event yourself like any other event that's not dialog related. When TRUE is returned, you'll generally pass the event to DialogSelect for it to handle (as described in the section "DialogSelect" in this chapter), but in some special cases, you may want to bypass DialogSelect or to perform some preprocessing before calling it. If so, check for those events and respond accordingly.

For cases other than these, pass the event to DialogSelect for that routine to handle.

$0F15 ModalDialog

If the frontmost window is a modal dialog box, ModalDialog repeatedly gets and handles events in the dialog's window. After the routine handles an event involving an enabled dialog item, it returns with the item ID in *itemHit*. Normally you'll then do whatever is appropriate in response to an event in that item.

Call ModalDialog after creating a modal dialog box and making its window the frontmost window.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *filterProcPtr* --	**Long**—POINTER to a filter procedure; NIL for standard filter
	← **SP**

Stack after call

previous contents	
itemHit	**Word**—ID of item hit; 0 if window not a modal dialog
	← **SP**

Errors

$150D `notModalDialog` Frontmost window not a modal dialog window

C

```
extern pascal Word ModalDialog(filterProcPtr)

WordProcPtr     filterProcPtr;
```

(continued)

More about ModalDialog and events

ModalDialog gets each event by calling the Event Manager routine GetNextEvent. If the event is a mouse-down event outside the content region of the dialog window, ModalDialog emits sound number 1 (which is preset to sound a single beep) and gets the next event; otherwise, it filters and handles the event as described next.

❖ *Note:* Once before getting each event, ModalDialog calls SystemTask, a Desk Manager routine that must be called regularly so that desk accessories will work properly.

If the filter procedure pointed to by *filterProcPtr* returns TRUE, ModalDialog will return immediately rather than handle the event; in this case, the filterProc procedure sets *itemHit* to the item ID that ModalDialog should return.

If you want the filter procedure to handle a special event and prevent ModalDialog from handling it, but don't want to leave ModalDialog, change the *what* field of the Event Record to nullEvent and return FALSE.

ModalDialog handles events for which the filterProc procedure returns FALSE as follows:

- For an activate or update event for the dialog window, ModalDialog activates or updates the window.
- For a mouse-down event in an editLine item, ModalDialog responds to the mouse activity as appropriate (displaying an insertion point or selecting text).
- For a key-down event with an editLine item present, text entry and editing are handled in the standard way, with the following exception: If the Apple key is pressed, the event is ignored unless the default filter is used. In any case, ModalDialog returns if the editLine item is enabled, or does nothing if the editLine item is disabled.
- For a key-down event with no editLine item present, ModalDialog does nothing.
- For a mouse-down event in any control (except scroll bars), ModalDialog calls the Control Manager routine TrackControl. If the mouse button is released inside the control and the control is enabled, ModalDialog returns; otherwise, it does nothing.
- For a mouse-down event in a scroll bar item, ModalDialog calls the Control Manager routine TrackControl with a special action procedure that calls your dialog scroll bar action procedure.
- For a mouse-down event in any other enabled item in the dialog box, ModalDialog returns.
- For a mouse-down event in any other disabled item or in no item, or if any other event occurs, ModalDialog does nothing.

$2C15 ModalDialog2

If the frontmost window is a modal dialog, ModalDialog2 repeatedly gets and handles events in the dialog's window. After handling an event involving an enabled dialog item, it returns with the part code and the item ID in *itemHitInfo*. Normally you'll then do whatever is appropriate in response to an event in that item. For example, if a key is pressed and an editLine item is present in the dialog, the part code is `inEditLine` and the item ID is the ID of the current active editLine. Part codes are documented in Chapter 4, "Control Manager."

Call ModalDialog2 after creating a modal dialog and displaying its window in the frontmost plane.

❖ *Note:* If the frontmost window is not a modal dialog box (for instance, if it is a regular window or a modeless dialog), ModalDialog2 returns immediately with *itemHitInfo* set to 0.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *filterProcPtr* --	**Long**—POINTER to a filter procedure; NIL for standard
	← **SP**

Stack after call

previous contents	
-- *itemHitInfo* --	**Long**—High-order word = part code of item hit, low-order word = item
	← **SP**

Errors

$150D `notModalDialog` Frontmost window not a modal dialog window

C

```
extern pascal LongWord ModalDialog2(filterProcPtr)

WordProcPtr     filterProcPtr;
```

$0D15 NewDItem

Adds a new item to a specified dialog's item list.

The possible item types are buttonItem, checkItem, radioItem, scrollBarItem, userCtlItem, userCtlItem2, statText, longStatText, longStatText2, editLine, iconItem, picItem, and userItem. For more information about how the item type affects the *itemDescr* and *itemValue* parameters, see the section "Item Descriptor and Item Value" in this chapter.

Warning

You must not have any item with an ID of 0.

If the bottom-right coordinate of the rectangle pointed to by *itemRectPtr* is specifed as (0,0), the Dialog Manager provides a default size for simple buttons, radio buttons, and check boxes.

Parameters

Stack before call

previous contents	
— *theDialogPtr* —	**Long**—POINTER to dialog this item belongs to
itemID	**Word**—Item identifier for all item-related calls
— *itemRectPtr* —	**Long**—POINTER to rectangle enclosing item in dialog's local coordinates
itemType	**Word**—Item type
— *itemDescr* —	**Long**—Depends upon item type
itemValue	**Word**—Depends upon item type
itemFlag	**Word**—Includes visible/invisible flag (0 for default flag)
— *itemColorPtr* —	**Long**—POINTER to item's default color table; NIL for default
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors		
$150A	badItemType	Inappropriate item type
$150B	newItemFailed	Item creation failed

C

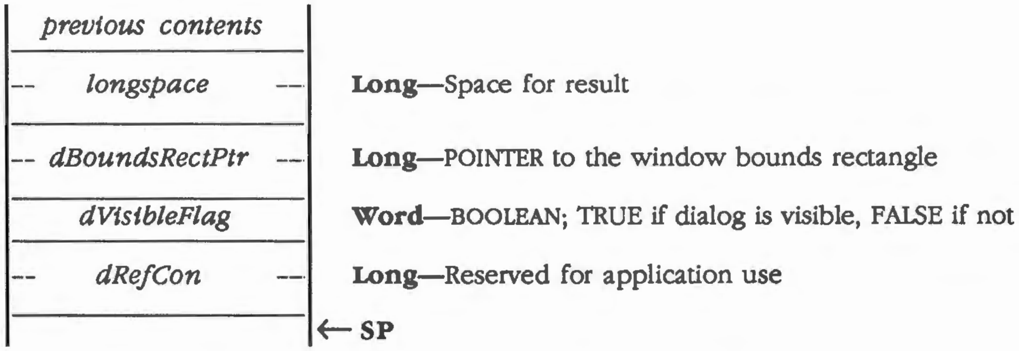
```
extern pascal void NewDItem(theDialogPtr,itemID,itemRectPtr,itemType,
itemDescr,itemValue,itemFlag,itemColorPtr)
GrafPortPtr     theDialogPtr;
Word     itemID;
Rect *itemRectPtr;
Word     itemType;
Pointer     itemDescr;
Word     itemValue;
Word     itemFlag;
CtlColorTablePtr     itemColorPtr;
```

$0A15 NewModalDialog

Creates a specified modal dialog and returns a pointer to the GrafPort of the new dialog.

Parameters

Stack before call



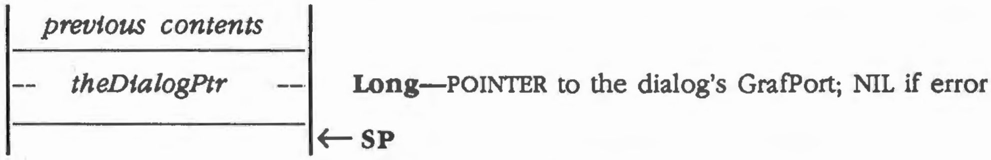
Long—Space for result

Long—POINTER to the window bounds rectangle

Word—BOOLEAN; TRUE if dialog is visible, FALSE if not

Long—Reserved for application use

Stack after call



Long—POINTER to the dialog's GrafPort; NIL if error

Errors

Memory Manager errors   Returned unchanged

C

```
extern pascal GrafPortPtr NewModalDialog(dBoundsRectPtr,dVisibleFlag,
dRefCon)
Rect *dBoundsRectPtr;
Boolean     dVisibleFlag;
LongWord     dRefCon;
```

More about NewModalDialog parameters

The rectangle pointed to by *dBoundsRectPtr* determines the dialog window's size and location and must be specified in global coordinates. Remember that the top coordinate of this rectangle should be at least 25 pixels below the top of the screen for a modal dialog to allow for the menu bar.

If the *dVisibleFlag* parameter is TRUE, the dialog window is drawn on the screen. If it's FALSE, the window is initially invisible and may later be shown by a call to the Window Manager routine ShowWindow or SelectWindow.

❖ *Note:* NewModalDialog generates an update event for the entire window contents, so the items aren't drawn immediately. Thus, the routine allows you to perform some processing on the items before you draw them. However, if you change the value of an item or its text, the Control Manager draws the item immediately. If you find that all the items should be drawn at the same time, make the dialog invisible initially and then call ShowWindow.

The *dRefCon* parameter is reserved for the application use.

NewModalDialog sets the font of the dialog window's GrafPort to the system font or, if you previously called SetDAFont, to the specified font. It also sets the dialog type in the dialog record to Modal_Type.

$0B15 NewModelessDialog

Creates a specified modeless dialog and returns a pointer to the GrafPort of the new dialog. Modeless dialogs are described in the section "Dialog and Alert Windows" in this chapter.

Parameters

Stack before call

previous contents	
— *longspace* —	**Long**—Space for result
— *dBoundsRectPtr* —	**Long**—POINTER to RECT defining window bounds rectangle
— *dTitlePtr* —	**Long**—POINTER to string for dialog's title; NIL if no title
— *dBehindPtr* —	**Long**—POINTER to window the dialog should be behind
dFlag	**Word**—Bit flag describing the dialog's frame
— *dRefCon* —	**Long**—Reserved for application use
— *dFullSizePtr* —	**Long**—POINTER to RECT to be used as content's zoomed size
	← **SP**

Stack after call

previous contents	
— *theDialogPtr* —	**Long**—POINTER to the dialog's GrafPort; NIL if error
	← **SP**

Errors

Memory Manager errors	Returned unchanged

C

```
extern pascal GrafPortPtr NewModelessDialog(dBoundsRectPtr,dTitlePtr,
dBehindPtr,dFlag,dRefCon,dFullSizePtr)

Rect *dBoundsRectPtr;

Pointer     dTitlePtr;

GrafPortPtr     dBehindPtr;

Word     dFlag;

LongWord     dRefCon;

Rect *dFullSizePtr;
```

More about NewModelessDialog parameters

The rectangle pointed to by *dBoundsRectPtr* determines the dialog window's size and location and must be given in global coordinates. Remember that the top coordinate of this rectangle should be at least 25 pixels below the top of the screen for a modal dialog to allow for the menu bar.

The *dBehindPtr* parameter points to the window behind which the dialog window is to be placed on the desktop. Pass –1 ($FFFFFFFF) to display the dialog window in front of all other windows.

The *dFlag* parameter allows you to specify the frame of the dialog box, as described in the section "NewWindow" in Chapter 25, "Window Manager," in Volume 2.

The *dRefCon* parameter is reserved for application use.

The *dFullSizePtr* parameter points to a rectangle describing the size and location of the dialog after the user has zoomed in on the dialog.

$1915 NoteAlert

Performs the same functions as the Alert routine, except that before drawing the items of the alert in the alert box, NoteAlert draws the note icon in the upper-left corner of the box. The note icon is shown in Figure 6-13.

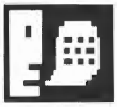


Figure 6-13
Note Icon

The alert template is defined in Figure 6-8 under the Alert routine.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *alertTemplatePtr* --	**Long**—POINTER to an alert template
-- *filterProcPtr* --	**Long**—POINTER to filter procedure; NIL for default filter
	← **SP**

Stack after call

previous contents	
itemHit	**Word**—ID of item Hit
	← **SP**

Errors

None

C

```
extern pascal Word NoteAlert(alertTemplatePtr,filterProcPtr)
AlertTempPtr      alertTemplatePtr;
WordProcPtr      filterProcPtr;
```

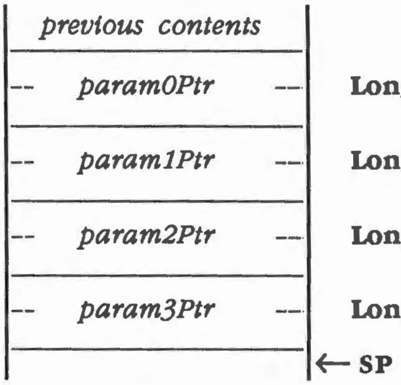
$1B15 ParamText

Specifies text for 1–4 special strings in statText, longStatText, and longStatText2 items. The strings pointed to by *param0Ptr* through *param3Ptr* replace the special strings "^0" through "^3" in all static-text type items in all subsequent dialog or alert boxes. Your application can make the call as many times as necessary.

You may pass NIL for parameters not used or for strings that are not to be changed.

Parameters

Stack before call



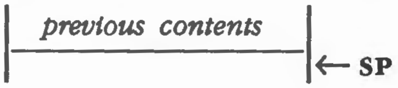
Long—POINTER to string ^0; NIL = no change

Long—POINTER to string ^1; NIL = no change

Long—POINTER to string ^2; NIL = no change

Long—POINTER to string ^3; NIL = no change

Stack after call



Errors

None

C

```
extern pascal void ParamText(param0Ptr,param1Ptr,param2Ptr,param3Ptr)

Pointer      param0Ptr;

Pointer      param1Ptr;

Pointer      param2Ptr;

Pointer      param3Ptr;
```

$0E15 RemoveDItem

Removes a specified item from a specified dialog and erases it from the screen. The routine also invalidates the item area, so that any other items behind the specified item are redrawn.

Parameters

Stack before call

previous contents	
-- *theDialogPtr* --	**Long**—POINTER to the dialog's GrafPort
itemID	**Word**—ID of item to be removed
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$150C	`itemNotFound`	No such item

C

```
extern pascal void RemoveDItem(theDialogPtr,itemID)

GrafPortPtr     theDialogPtr;

Word     itemID;
```

$3515 ResetAlertStage

Resets the stage of the last occurrence of an alert so that the next occurrence of that same alert will be treated as its first stage. This is useful, for example, when you've used the ParamText routine to change the text of an alert such that, from the user's point of view, it becomes a different alert.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

None

C

```
extern pascal void ResetAlertStage()
```

$2115 SelectIText

Sets the selection range or insertion point for a specified editLine item in a specified dialog box. SelectIText also makes the specified editLine item the current edit item when the dialog box contains more than one such item.

Parameters

Stack before call

previous contents	
— *theDialogPtr* —	**Long**—POINTER to the dialog
itemID	**Word**—ID of item in dialog
startSel	**Word**—Start of selection, by character position
endSel	**Word**—End of selection, by character position
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SelectIText(theDialogPtr,itemID,startSel,endSel)

GrafPortPtr     theDialogPtr;

Word     itemID;

Word     startSel;

Word     endSel;
```

(continued)

More about SelectIText

Given a pointer to a dialog and the item ID of an editLine item in the dialog box, SelectIText does the following:

- If the item contains text, SelectIText sets the selection range to extend from character position *startSel* up to but not including character position *endSel*. The selection range is inverted unless *startSel* equals *endSel*, in which case a blinking vertical bar is displayed to indicate an insertion point at that position.
- If the item doesn't contain text, SelectIText simply displays the insertion point.

For example, if the user makes an unacceptable entry in the editLine item, the application can display an alert box reporting the problem and then select the entire text of the item so it can be replaced by a new entry. (Without this procedure, the user would have to select the item before making the new entry.)

❖ *Note:* You can select the entire text by specifying 0 for *startSel* and 32767 for *endSel*. For details about selection range and character position, see Chapter 10, "LineEdit Tool Set."

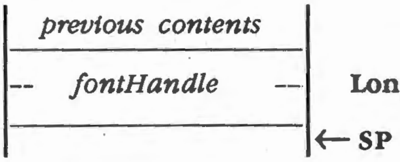
$1C15 SetDAFont

Specifies the font for the dialog or alert window's GrafPort. SetDAFont affects statText items, editLine items, and standard controls.

If you don't call this routine, the system font is used. For more information about fonts, see Chapter 8, "Font Manager."

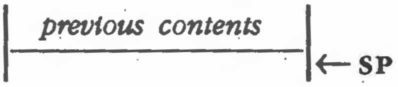
Parameters

Stack before call



Long—HANDLE to the new font

Stack after call



Errors

None

C

```
extern pascal void SetDAFont(fontHandle)

FontHndl     fontHandle;
```

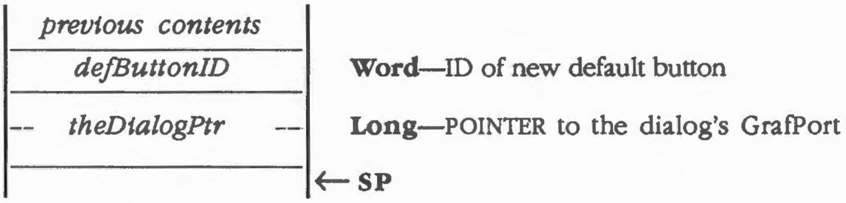
$3815 SetDefButton

Sets the ID of the default button to a specified ID.

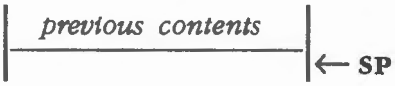
Important
The *defButtonID* must be the ID of a button item.

Parameters

Stack before call


previous contents
defButtonID
Word—ID of new default button
-- theDialogPtr --
Long—POINTER to the dialog's GrafPort
← SP

Stack after call



Errors

None

C

```
extern pascal void SetDefButton(defButtonID,theDialogPtr)

Word      defButtonID;

GrafPortPtr     theDialogPtr;
```

$2915 SetDItemBox

Changes the display rectangle of a specified item to a new display rectangle.

The routine does not redraw the item. This allows you to change the enclosing rectangle for several items and then redraw all of the changes at the same time.

❖ *Note:* If only one item is changed, the best way to change the display rectangle is to call HideDItem, SetDItemBox, and ShowDItem, in that order.

Parameters

Stack before call

previous contents	
-- *theDialogPtr* --	**Long**—POINTER to the dialog's GrafPort
itemID	**Word**—ID of item in dialog
-- *itemBoxPtr* --	**Long**—POINTER to RECT defining new display rectangle
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$150C `itemNotFound` No such item

C

```
extern pascal void SetDItemBox(theDialogPtr,itemID,itemBoxPtr)

GrafPortPtr     theDialogPtr;

Word     itemID;

Rect *itemBoxPtr;
```

$2715 SetDItemType

Changes a specified item to a new specified item type. The routine does not redraw the item. This allows you to change the type of several items and then redraw all the changes at the same time.

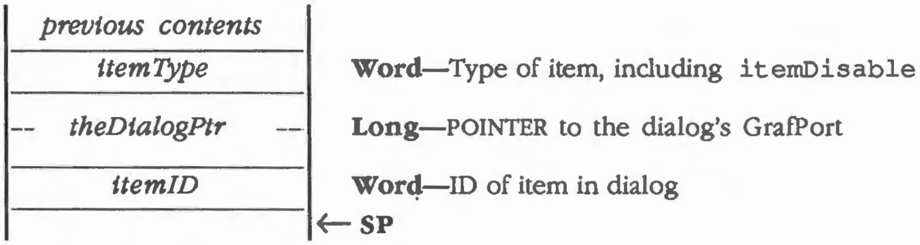
If you want the item to be disabled, add `itemDisable` to *itemType*.

Important

Changing the type of an item can be very dangerous. The itemDisable status can be changed by the DisableDItem or EnableDItem routines.

Parameters

Stack before call

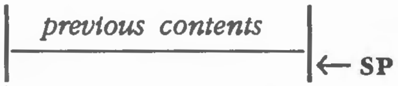


Word—Type of item, including `itemDisable`

Long—POINTER to the dialog's GrafPort

Word—ID of item in dialog

Stack after call



Errors

$150C	`itemNotFound`	No such item

C

```
extern pascal void SetDItemType(itemType,theDialogPtr,itemID)

Word        itemType;

GrafPortPtr      theDialogPtr;

Word        itemID;
```

$2F15 SetDItemValue

Sets the value of a specified item to a new desired value and redraws the item.

For standard controls, *itemValue* is the new value of the control. For the other types of items, *itemValue* may have special meaning, as follows:

- For a longStatText or longStatText2 item, the value is the length for the text.
- For a userItem, iconItem, or statText item, the value is reserved for the application's use.

Parameters

Stack before call

previous contents	
itemValue	**Word**—New value
— *theDialogPtr* —	**Long**—POINTER to the dialog's GrafPort
itemID	**Word**—ID of item in dialog
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$150C `itemNotFound` No such item

C

```
extern pascal void SetDItemValue(itemValue,theDialogPtr,itemID)

Word        itemValue;

GrafPortPtr     theDialogPtr;

Word        itemID;
```

$2015 SetIText

Provides the text for a specified statText or editLine item in a specified dialog box and draws the item.

For example, suppose the exact content of a dialog's text item cannot be determined until the dialog is created, but the display rectangle is already defined. Call the SetIText routine with the desired text.

Parameters

Stack before call

previous contents	
-- *theDialogPtr* --	**Long**—POINTER to the dialog's GrafPort
itemID	**Word**—ID of item in dialog
-- *theStringPtr* --	**Long**—POINTER to the new text string
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$150A	`badItemType`	Inappropriate item type; only statText and editLine allowed
$150C	`itemNotFound`	No such item

C

```
extern pascal void SetIText(theDialogPtr,itemID,theStringPtr)

GrafPortPtr     theDialogPtr;

Word     itemID;

Pointer     theStringPtr;
```

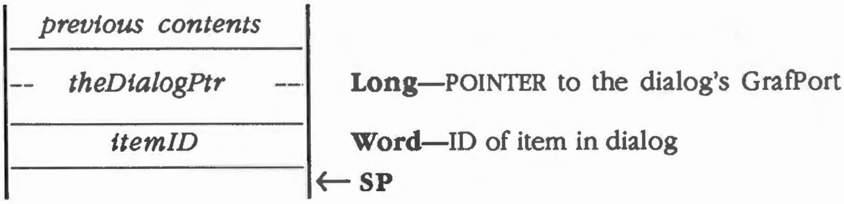
$2315 ShowDItem

Makes visible a specified item from a specified dialog. The item may have been hidden by HideDItem or may have been invisible when created.

If the item is already visible, ShowDItem does nothing.

Parameters

Stack before call



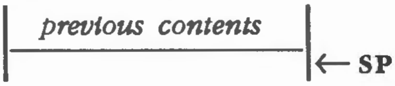
previous contents

-- *theDialogPtr* -- **Long**—POINTER to the dialog's GrafPort

itemID **Word**—ID of item in dialog

← **SP**

Stack after call



previous contents

← **SP**

Errors

$150C	`itemNotFound`	No such item

C

```
extern pascal void ShowDItem(theDialogPtr,itemID)

GrafPortPtr     theDialogPtr;

Word      itemID;
```

$1815 StopAlert

Invokes an alert defined by a specified alert template and draws the stop icon in the upper-left corner of the box. The stop icon is shown in Figure 6-14.

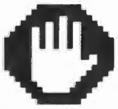


Figure 6-14
Stop Icon

The alert template is defined in Figure 6-8 under the Alert routine.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *alertTemplatePtr* --	**Long**—POINTER to an alert template
-- *filterProcPtr* --	**Long**—POINTER to filter procedure; NIL for standard filter
	← **SP**

Stack after call

previous contents	
itemHit	**Word**—ID of item hit
	← **SP**

Errors

None

C

```
extern pascal Word StopAlert(alertTemplatePtr,filterProcPtr)

AlertTempPtr     alertTemplatePtr;

WordProcPtr     filterProcPtr;
```

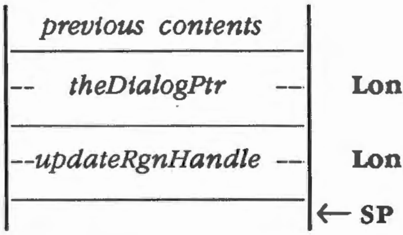
$2515 UpdateDialog

Redraws the part of a specified dialog that is in a specified update region.

If the specified update region is part of a region that will be updated in an upcoming update event, you should call the Window Manager routine ValidRgn to prevent the Dialog Manager from redrawing the region twice. For more information, see the sections "GetUpdateRgn" and "How a Window Is Drawn" in Chapter 25, "Window Manager," in Volume 2.

Parameters

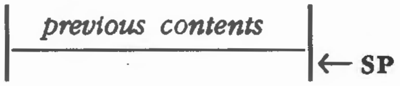
Stack before call



Long—POINTER to the dialog's GrafPort

Long—HANDLE to update region

Stack after call



Errors

None

C

```
extern pascal void UpdateDialog(theDialogPtr,updateRgnHandle)

GrafPortPtr     theDialogPtr;

RgnHandle      updateRgnHandle;
```

Dialog Manager summary

This section briefly summarizes the constants, data structures, and tool set errors contained in the Dialog Manager.

Important

These definitions are provided in the appropriate interface file.

Table 6-6
Dialog Manager constants

Name	Value	Description
Dialog scroll bar commands		
`getInitView`	$0001	View size at creation
`getInitTotal`	$0002	Total size at creation
`getInitValue`	$0003	Value at creation
`scrollLineUp`	$0004	Scroll one line up
`scrollLineDown`	$0005	Scroll one line down
`scrollPageUp`	$0006	Scroll one page up
`scrollPageDown`	$0007	Scroll one page down
`scrollThumb`	$0008	Scroll to thumb position
Item types		
`buttonItem`	$000A	Standard button control
`checkItem`	$000B	Standard check box control
`radioItem`	$000C	Standard radio button control
`scrollBarItem`	$000D	Special scroll bar for dialogs
`userCtlItem`	$000E	Application-defined control
`statText`	$000F	Static text; text that cannot be edited
`longStatText`	$0010	Static text
`editLine`	$0011	Text that can be edited
`iconItem`	$0012	An icon
`picItem`	$0013	A QuickDraw II picture
`userItem`	$0014	Application-defined item
`userCtlItem2`	$0015	Application-defined control
`longStatText2`	$0016	Static text; text cannot be edited and can contain embedded commands
`itemDisable`	$8000	Added to any item, this disables that item
Item type range		
`minItemType`	$000A	Minimum valid item type
`maxItemType`	$0016	Maximum valid item type

Table 6-6 (continued)
Dialog Manager constants

Name	Value	Description
Item IDs		
`ok`	$0001	OK button
`cancel`	$0002	Cancel button
Part codes		
`inButton`	$0002	User clicked simple button
`inCheckBox`	$0003	User clicked check box
`inRadioButton`	$0004	User clicked radio button
`inUpArrow`	$0005	User clicked up arrow
`inDownArrow`	$0006	User clicked down arrow
`inPageUp`	$0007	User clicked in page-up area
`inPageDown`	$0008	User clicked in page-down area
`inStatText`	$0009	User clicked statText item
`inGrow`	$000A	User clicked size box
`inEditLine`	$000B	User clicked in text that can be edited
`inUserItem`	$000C	User clicked application-defined item
`inLongStatText`	$000D	User clicked longStatText item
`inIconItem`	$000E	User clicked an icon
`inLongStatText2`	$000F	User clicked longStatText2 item
`inThumb`	$0081	User clicked in thumb area of scroll bar
Stage bit flags		
`okDefault`	$0000	OK is the default button for alert
`cancelDefault`	$0040	Cancel is the default button for alert
`alertDrawn`	$0080	Draw alert

Table 6-7
Dialog Manager data structures

Name	Offset	Type	Definition
AlertTemplate			
atBoundsRect	$00	Rect	Alert bounds rectangle
atAlertID	$08	Word	ID identifying alert
atStage1	$0A	Byte	First stage of alert
atStage2	$0B	Byte	Second stage of alert
atStage3	$0C	Byte	Third stage of alert
atStage4	$0D	Byte	Fourth stage of alert
atItemList	$0E	ItemTempPtr	Points to first item in item list; list terminated by a NIL pointer

(continued)

Table 6-7 (continued)
Dialog Manager data structures

Name	Offset	Type	Definition
DialogTemplate			
dtBoundsRect	$00	Rect	Dialog bounds rectangle
dtVisible	$08	Boolean	TRUE if dialog is to be visible
dtRefCon	$0A	Long	Reserved for application use
dtItemList	$0E	ItemTempPtr	Points to first item in item list; list terminated by a NIL pointer
IconRecord			
iconRect	$00	Rect	Bounds rectangle (width is multiple of 8) of rectangle enclosing icon
iconImage	$08	Array	Pixel image
ItemTemplate			
itemID	$00	Word	Number identifying item
itemRect	$02	Rect	Display rectangle, in local coordinates
itemType	$0A	Word	Type of item (button, check, scroll, and so on)
itemDescr	$0C	Pointer	Item descriptor
itemValue	$10	Word	Item value
itemFlag	$12	Word	Bit flag (0 for default)
itemColor	$14	CtlColorTablePtr	Pointer to color table (NIL for default)
UserCtlItemPB			
defProcParm	$00	LongProcPtr	Address of definition procedure
titleParm	$04	Pointer	Pointer to title string
param2	$08	Word	First parameter
param1	$0A	Word	Second parameter

Note: The actual assembly-language equates have a lowercase *o* (the letter) in front of all of the names given in this table.

Table 6-8
Dialog Manager error codes

Code	Name	Description
$150A	badItemType	Inappropriate item type
$150B	newItemFailed	Item creation failed
$150C	itemNotFound	No such item
$150D	notModalDialog	Frontmost window not a modal dialog window

Chapter 7

Event Manager

This chapter describes the **Event Manager,** the Apple IIGS tool set that allows applications to monitor the user's actions, such as those involving the mouse, keyboard, and keypad. These actions are reported to the application as **events.** For example, whenever the user presses or releases the mouse button, the Event Manager records the action as an event.

A typical event-driven application decides what to do from moment to moment by asking the Event Manager for events and responding to them one by one in whatever way is appropriate.

In general, events are collected from a variety of sources and reported to the application on demand, one at a time. The Event Manager doesn't necessarily report the events in the order they occurred because some have a higher priority than others.

The Event Manager is also used by other tool sets and managers; for instance, the Window Manager uses events to coordinate the sequence and display of windows on the screen.

A preview of the Event Manager routines

To introduce you to the capabilities of the Event Manager, all Event Manager routines are grouped by function and briefly described in Table 7-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the Event Manager routines (discussed in alphabetical order).

Table 7-1
Event Manager routines and their functions

Routine	Description
Housekeeping routines	
EMBootInit	Initializes the Event Manager; called only by the Tool Locator—must not be called by an application
EMStartUp	Starts up the Event Manager for use by an application
EMShutDown	Shuts down the Event Manager when an application quits
EMVersion	Returns the version of the Event Manager
EMReset	Returns an error if the Event Manager is active—must not be called by an application
EMStatus	Indicates whether the Event Manager is active
Event accessing routines	
GetNextEvent	Returns the next available event of a specified type or types; if the event is in the event queue, GetNextEvent removes the event from the queue
EventAvail	Allows an application to look at the next available event of a specified type or types
GetOSEvent	Returns the next available queue event of a specified type or types and removes it from the queue
OSEventAvail	Allows an application to look at the next available queue event of a specified type or types, but leaves the event in the queue
Mouse status routines	
GetMouse	Returns the current mouse location
Button	Returns the current status of a specified button on the mouse
Stilldown	Tests whether the specified mouse button is still down
WaitMouseUp	Tests whether the specified mouse button is still down and, if not, removes the preceding mouse-up event
Event queue routines	
PostEvent	Posts an event into the event queue
FlushEvents	Removes all queue events of the type or types specified by an event mask up to but not including the first event of any type specified by a stop mask
Miscellaneous routines	
TickCount	Returns the current number of ticks (in sixtieths of a second) since the system was last started
GetDblTime	Returns the maximum difference (in ticks) between mouse-up and mouse-down events allowed for the mouse clicks to be considered a double-click
GetCaretTime	Returns the time (in ticks) between blinks of the caret (usually indicated by a vertical bar) marking the insertion point in text that can be edited
SetEventMask	Specifies the system event mask
FakeMouse	Allows use of an alternative pointing device, such as a graphics tablet, in place of or in conjunction with the mouse
DoWindows	Returns the address of the direct page used by the Event Manager
SetSwitch	Generates a switch event

Two managers in one

Although the Event Manager is a single tool set, it can be conceptually divided into a high-level Event Manager and a low-level Event Manager. These two parts perform different functions. The low-level Event Manager detects low-level, hardware-related events such as mouse-button presses and keystrokes. It stores information about these events in the event queue and provides routines that access the queue.

The high-level Event Manager calls the low-level Event Manager to retrieve events from the event queue. In addition, the high-level Event Manager reports window and switch events, which aren't kept in the queue.

Event types

Events are of various **event types.** Some report actions by the user; others are generated by the Window Manager, device drivers, or the application itself. The system handles some events before the application sees them; it leaves others for the application to handle. The event types are discussed in the following sections.

Mouse events

When the user presses the mouse button, the system generates a **mouse-down event;** when the user releases the button, the system generates a **mouse-up event.** Movements of the mouse cause the cursor position to be updated but are not reported as events.

Keyboard events

When the user presses any character key on the keyboard or keypad, the system generates a **key-down event.** The character keys include all keys except Shift, Caps Lock, Control, Option, and Apple, which are called **modifier keys.** Modifier keys are treated differently and generate no keyboard events of their own. When an event is posted, the state of the modifier keys is reported in a field of the event record.

The character keys on the keyboard and keypad also generate **auto-key events** when the user holds them down. An auto-key event is generated only when the event queue is empty.

Two different time intervals are associated with auto-key events. The first auto-key event is generated after a certain initial delay has elapsed since the key was originally pressed; this is called the **repeat delay.** Subsequent auto-key events occur each time a certain interval has elapsed since the last such event; this is called the **repeat speed.** The user can change these values with the Control Panel.

Window events

The Window Manager generates events to coordinate the display of windows on the screen (see Chapter 25, "Window Manager," in Volume 2). These events are either activate or update events:

- **Activate events** are generated whenever an inactive window becomes active or an active window becomes inactive. These events generally occur in pairs (for example, one window is deactivated and then another is activated).
- **Update events** occur when all or part of a window's contents need to be drawn or redrawn, usually as a result of the user opening, closing, activating, or moving a window.

Other events

Device driver events are generated by device drivers in certain situations; for example, an application might set up a driver to report an event when its transmission of data is interrupted. The device driver uses the PostEvent routine to place device driver events in the event queue.

Switch events are reserved for future use.

A **desk accessory event** occurs whenever the user enters Control-Apple-Escape to invoke a classic desk accessory.

An application can define as many as four **application events** of its own and use them for any purpose. Application-defined events are placed in the event queue with the PostEvent routine.

The Event Manager returns a null event if it has no other events to report.

Event priority

The Event Manager generally retrieves events from the event queue in the order of their original posting. However, the way that various types of events are generated and detected causes some events to have higher priority than others. Also, not all events are kept in the event queue. Furthermore, when an application asks for an event, it can specify the types in which it is interested. Specifying the types can cause the Event Manager to pass over some events in favor of others.

The GetNextEvent and EventAvail routines always return the highest-priority event available among the requested types. They rank events in the following order:

1. Activate events (one window becomes inactive before another window becomes active)
2. Switch events (reserved for future use)
3. Mouse-down, mouse-up, key-down, auto-key, device driver, application-defined, and desk accessory events (in order of posting)
4. Update events (in front-to-back order of windows)

Activate events take priority over all others; they're detected in a special way and are never actually placed in the event queue. The GetNextEvent and EventAvail routines check for pending activate events before looking in the event queue, so they will always return such an event if one is available. Because of the special way the routines detect activate events, more than two such events cannot be pending at the same time; at most one event can be pending to make a window inactive followed by another event waiting to make a window active.

Next in priority are switch events, which also remain outside of the event queue.

❖ *Note:* At the time of publication, switch events were reserved for future use and weren't currently generated by the toolbox.

If no activate events are pending, the GetNextEvent and EventAvail routines check for a switch event before looking in the event queue. If a switch event is available, the routines then check to see if any update events are pending; if so, they return the update event to the application. GetNextEvent and EventAvail return a switch event to the application only when no update events are pending. This ensures that all windows are updated before the switch event is returned to the application.

The third category includes all of the event types in the event queue. The Event Manager retrieves events from the queue in the order of their posting. The GetOSEvent and OSEventAvail routines return only events from this category.

The final category of events are update events. Like activate and switch events, the update events are not placed in the event queue, but are detected in another way. If no higher-priority event is available, the GetNextEvent and EventAvail routines check for windows whose contents need to be drawn. If they find such a window, they return an update event for that window. The routines check the windows in the order in which the windows are displayed on the screen, from front to back. Thus, if two or more windows require updating, GetNextEvent and EventAvail return an update event for the frontmost window.

Finally, if no other event is available, a null event is returned.

❖ *Note:* If the queue becomes full, the Event Manager begins discarding old events to make room for new ones as they're posted. The events discarded are always the oldest ones in the queue.

Event records

Every event, including a null event, is represented by an **event record** containing all pertinent information about that event. The event record includes the following information:

- The type of event
- Event-specific information, such as which key the user pressed or which window is being activated
- The time the event was posted
- The location of the mouse at the time the event was posted (in global coordinates)
- The state of the mouse buttons and modifier keys at the time the event was posted

Every event, including a null event, has a 16-byte event record containing the preceding information defined as follows:

what	WORD	Event code
message	LONG	Event message
when	LONG	Tick count
where	POINT	Mouse location
modifiers	WORD	Modifier flags

The *when* field contains the number of ticks since the system was last started, and the *where* field gives the location of the mouse, in global coordinates, at the time the event was posted. The other three fields are described in the following sections.

Event codes

The *what* field of an event record contains an **event code** identifying the type of the event. Event codes are assigned as shown in Table 7-2.

Table 7-2
Event Manager event codes

Code	Name	Description
0	`nullEvt`	Null event
1	`mouseDownEvt`	Mouse-down event
2	`mouseUpEvt`	Mouse-up event
3	`keyDownEvt`	Key-down event
4	Undefined	
5	`autoKeyEvt`	Auto-key event
6	`updateEvt`	Update event
7	Undefined	
8	`activateEvt`	Activate event
9	`switchEvt`	Switch event (reserved for future use)
10	`deskAccEvt`	Desk accessory event
11	`driverEvt`	Device driver event
12	`app1Evt`	Application-defined event
13	`app2Evt`	Application-defined event
14	`app3Evt`	Application-defined event
15	`app4Evt`	Application-defined event

Event messages

The *message* field of an event record contains the **event message,** which conveys additional information about the event. The nature of this information depends on the event type, as shown in Table 7-3.

Table 7-3
Event messages

Event type	Event message
Mouse-down	Button number (0 or 1) in low-order word; high-order word undefined
Mouse-up	Button number (0 or 1) in low-order word; high-order word undefined
Key-down	ASCII character code in low-order byte (high bit clear); upper 3 bytes undefined
Auto-key	ASCII character code in low-order byte (high bit clear); upper 3 bytes undefined
Activate	Pointer to window
Update	Pointer to window
Device driver	Defined by the device driver
Application	Defined by the application
Switch	Undefined
Desk accessory	Undefined
Null	Undefined

Modifier flags

The *modifiers* field of an event record contains further information about activate events and about the state of the modifier keys and mouse buttons at the time the event was posted. For example, your application might look at this field to find out whether the Apple key was down when a mouse-down event was posted (which could affect the way objects are selected) or when a key-down event was posted (which could mean the user is choosing a menu item by typing its keyboard equivalent).

Figure 7-1 shows the bit positions of the modifier flags.

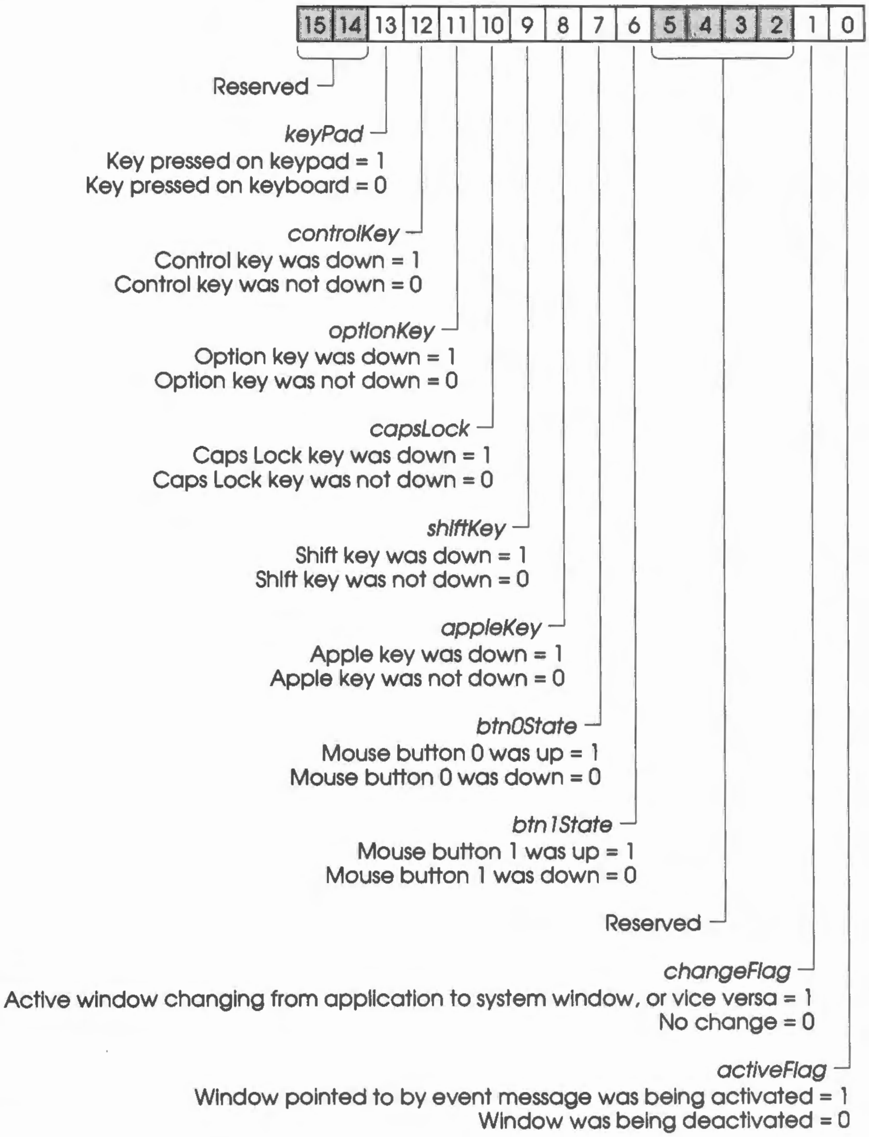

15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Reserved
keyPad
Key pressed on keypad = 1
Key pressed on keyboard = 0
controlKey
Control key was down = 1
Control key was not down = 0
optionKey
Option key was down = 1
Option key was not down = 0
capsLock
Caps Lock key was down = 1
Caps Lock key was not down = 0
shiftKey
Shift key was down = 1
Shift key was not down = 0
appleKey
Apple key was down = 1
Apple key was not down = 0
btn0State
Mouse button 0 was up = 1
Mouse button 0 was down = 0
btn1State
Mouse button 1 was up = 1
Mouse button 1 was down = 0
Reserved
changeFlag
Active window changing from application to system window, or vice versa = 1
No change = 0
activeFlag
Window pointed to by event message was being activated = 1
Window was being deactivated = 0

Figure 7-1
Modifier flags in event record

The *keyPad* bit gives further information about key-down events; it's set to 1 if the key pressed was on the keypad, or 0 if the key pressed was on the keyboard. Bits 12 through 6 indicate the state of the mouse button and modifier keys. Note that the *btn0State* and *btn1State* bits are set to 1 if the corresponding mouse button was up, whereas the bits for the five modifier keys are set to 1 if their corresponding keys were down.

❖ *Note:* On a one-button mouse, the button is button 0.

The *activeFlag* and *changeFlag* bits give further information about activate events. The *activeFlag* bit is set to 1 if the window pointed to by the event message is being activated, or 0 if the window is being deactivated. The *changeFlag* bit is set to 1 if the active window is changing from an application window to a system window or vice versa. Otherwise, it's set to 0.

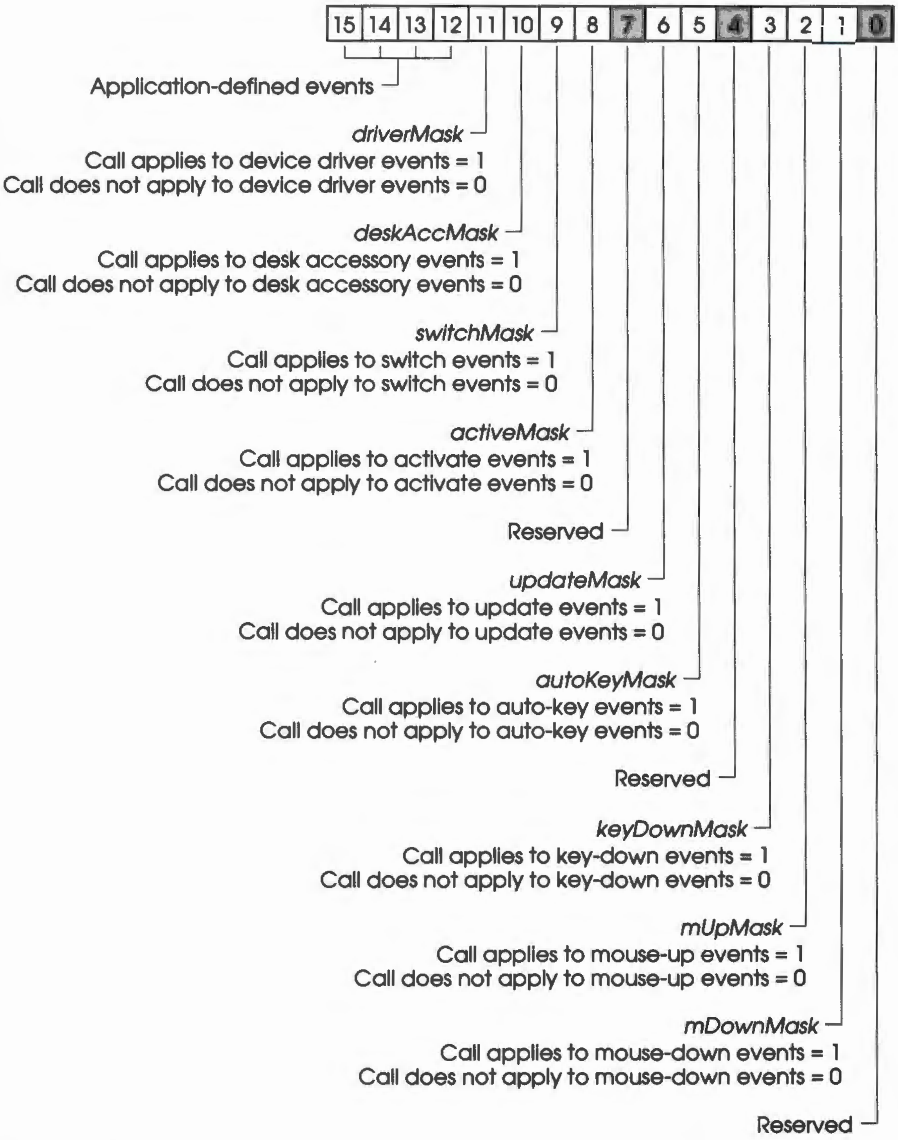
Event masks

Some Event Manager routines can be restricted to operate on a specific event type or group of types; in other words, the specified event types are enabled and all others are disabled. For instance, instead of just requesting the next available event, the application can specifically ask for the next keyboard event.

An application can specify which event types a particular call applies to by supplying an **event mask** as a parameter. This is a word with one bit position for each event type, as shown in Figure 7-2. The bit position representing a given type corresponds to the event code for that type—for example, update events (event code 6) are specified by bit 6 of the mask. A 1 in bit 6 means that this call applies to update events; a 0 means that it doesn't.

❖ *Note:* Null events can't be disabled; a null event is always reported when none of the enabled types of events are available.

There's also a global **system event mask** that controls which event types are posted into the event queue by the Event Manager. Only event types corresponding to bits set in the system event mask are posted; all others are ignored. When the system starts up, the system event mask is set to post all events.



Application-defined events

driverMask
Call applies to device driver events = 1
Call does not apply to device driver events = 0

deskAccMask
Call applies to desk accessory events = 1
Call does not apply to desk accessory events = 0

switchMask
Call applies to switch events = 1
Call does not apply to switch events = 0

activeMask
Call applies to activate events = 1
Call does not apply to activate events = 0

Reserved

updateMask
Call applies to update events = 1
Call does not apply to update events = 0

autoKeyMask
Call applies to auto-key events = 1
Call does not apply to auto-key events = 0

Reserved

keyDownMask
Call applies to key-down events = 1
Call does not apply to key-down events = 0

mUpMask
Call applies to mouse-up events = 1
Call does not apply to mouse-up events = 0

mDownMask
Call applies to mouse-down events = 1
Call does not apply to mouse-down events = 0

Reserved

Figure 7-2
Event masks

Using the Event Manager

This section discusses how the Event Manager routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

The Event Manager depends upon the presence of the tool sets shown in Table 7-4 and requires that at least the indicated version of the tool set be present.

Table 7-4
Event Manager—other tool sets required

Tool set number	Tool set name	Minimum version needed
$01 #01	Tool Locator	1.0
$02 #02	Memory Manager	1.0
$03 #03	Miscellaneous Tool Set	1.0
$04 #04	QuickDraw II	1.0
$05 #05	Desk Manager	1.0
$09 #09	ADB Tool Set	1.0

The first Event Manager call that your application must make is EMStartUp. Conversely, when you quit your application, you must make the EMShutDown call.

Any application that uses both the Event Manager and the Window Manager must start the Event Manager before starting up the Window Manager. Because the Event Manager shares data with the Window Manager, both must use the same direct-page work area. When an application starts up the Window Manager, the Window Manager automatically calls the Event Manager routine DoWindows to obtain the address of the direct-page work area assigned to the Event Manager. If DoWindows is not called, the Event Manager assumes that the application does not use windows and makes no attempt to return window events.

Event-driven applications have a main loop that repeatedly calls GetNextEvent to retrieve the next available event and then takes an appropriate action for each type of event. Some typical responses to commonly occurring events are described in the next section. Your application should respond only to those events that are directly related to its own operations. After calling GetNextEvent, it should test the Boolean result to find out whether it should respond to the event: TRUE means the event may be of interest to the application; FALSE means it will usually not be of interest. In some cases, you may want your application simply to look at a pending event while leaving it available for subsequent retrieval by GetNextEvent. You can use the EventAvail routine for this purpose. Also remember that, if you are using the Window Manager TaskMaster routine, TaskMaster will intercept and act upon certain types of events, such as mouse events occurring in a window's Close box, Zoom box, or scroll bars. For more details, see the section "Using TaskMaster" in Chapter 25, "Window Manager," in Volume 2.

Responding to mouse events

On receiving a mouse-down event, an application should first call the Window Manager to find out where the cursor was on the screen when the mouse button was pressed and then respond in whatever way is appropriate. Depending upon where the cursor was when the button was pressed, the application may have to call toolbox routines in the Menu Manager, the Desk Manager, the Window Manager, or the Control Manager.

If the application attaches special significance to the user pressing a modifier key or keys along with the mouse button, the application can discover the state of the modifier keys by examining the appropriate flags in the *modifiers* field of the event record.

If you want your application to respond to mouse double-clicks, it must detect them itself. It can do so by comparing the time and location of a mouse-up event with the time and location of the mouse-down event immediately following the mouse-up event. The application should assume a double-click has occurred if both of the following are true:

- The times of the mouse-up event and the mouse-down event differ by a number of ticks less than or equal to the value returned by the GetDblTime routine.
- The locations of the two mouse-down events separated by the mouse-up event are sufficiently close to each other; exactly how close depends on the particular application. For example, a word-processing application might consider two locations to be essentially the same if they fall on the same character, whereas a graphics application might consider the locations essentially the same if the horizontal and vertical changes in position total no more than five pixels.

Mouse-up events can be significant in other ways; for example, they can signal that the user has stopped dragging the mouse after selecting the last of several objects. Most simple applications, however, ignore mouse-up events.

Responding to keyboard events

For a key-down event, the application should first check the *modifiers* field of the event record to see whether the character was typed with the Apple key held down; if so, the user may have been choosing a menu item by typing its keyboard equivalent.

If the key-down event is not a menu command, the application should respond to the event in whatever way is appropriate. For example, if one of the windows is active, the application could insert the typed character into the active document; if none of the windows is active, it might choose to ignore the event.

Most applications can handle auto-key events the same way as they handle key-down events. However, you may want your application to ignore auto-key events that invoke commands that you don't want continually repeated.

Responding to window events

When the application receives an activate event for one of its own windows, the Window Manager will already have completed all of the normal housekeeping tasks associated with the event, such as highlighting or unhighlighting the window. The application can then take any further necessary action, such as showing or hiding a scroll bar or highlighting or unhighlighting a selection.

On receiving an update event for one of its own windows, the application usually should update the contents of the window.

Responding to other events

Applications never receive desk accessory events because the Desk Manager intercepts and handles such events. Switch events were not implemented at the time of publication; however, when they are generated, your application will be returned to after being switched out of by a switcher-type application. Thus, upon receiving a switch event, your application should check to make sure that the environment is still the same as it was before the switch event occurred.

Posting and removing events

An application using application-defined events must call PostEvent to post these events to the event queue. Device drivers can post events the same way. PostEvent can also be useful for reposting an event that the application removed from the event queue by calling the GetNextEvent or GetOSEvent routine.

In some situations, you may want to remove from the event queue some or all events of a certain type or types. You can do this with the FlushEvents routine.

Performing other operations

In addition to receiving the user's mouse and keyboard actions in the form of events, applications can directly read the mouse location and state of the mouse buttons by calling the GetMouse and Button routines, respectively. To follow the mouse when the user moves it with the button down, the application can use the StillDown or WaitMouseUp routines.

The TickCount routine returns the number of ticks since the last system start up. Your application can compare this to the *when* field of an event record to discover the delay since that event was posted.

The GetCaretTime routine returns the number of ticks between blinks of the caret (usually a vertical bar) marking the insertion point in editable text. Your application should call GetCaretTime if you want to control the blinking of the caret. The application should check this value each time through the main event loop to ensure a constant frequency of blinking.

Applications must never call the DoWindows or SetSwitch routines and will probably never call the GetOSEvent, OSEventAvail, and SetEventMask routines.

Capturing asynchronous key events

You may want your application to trap asynchronous key events that could affect the current executing process. For example, you may want the user to be able to press Control-C to abort a process. Using the Event Manager alone, your application would have to check the event queue in a synchronous process for the Control-C keyboard event. This requires that the current process have the extra overhead of the Tool Locator and the Event Manager.

Your application can process asynchronous keyboard events without this extra overhead. After you make the EMStartUp call, replace the keyboard interrupt vector with a vector that points to an interrupt handler that you write to process asynchronous keyboard events.

Besides processing the asynchronous keyboard events, your handler must pass to the Event Manager any keystrokes the handler does not recognize as asynchronous keyboard events. To do this, your application must save the keyboard interrupt vector initialized by the Event Manager. The handler must dispatch to the Event Manager when the handler doesn't recognize a keyboard event.

In addition, your asynchronous keyboard event handler must clear the keyboard strobe on recognized events, but leave the keyboard strobe set on unrecognized events.

Following is an example of how you can install an asynchronous keyboard event handler:

```
              _MTStartup              ; Startup call to Miscellaneous Tool Set
* (Push appropriate EMStartup parameters on the stack.)
              _EMStartup              ; Startup call to Event Manager
              pushlong #0             ; Space for result
              pea   $000F             ; Vector is keyboard interrupt handler
              _GetVector              ; Get the vector set by the Event Manager
              pla                     ; Low word of current keyboard IRQ vector
              sta   >EventKey+1       ; Store for Event Manager dispatches
              pla                     ; High word of current keyboard IRQ vector
              sep   #$20              ; 8=bit m
              sta   >EventKey+3       ; Store for Event Manager dispatches
              rep   #$20              ; 16=bit m
              pea   $000F             ; Vector is keyboard interrupt handler
              pushlong #AsyncKey      ; Address of asynchronous key handler
              _SetVector              ; Install my handler
```

Following is an example of how you can implement an asychronous keyboard interrupt handler.

```
AsyncKey      start
kbd           equ   $00C000           ; Keyboard data
kbdstrb       equ   $00C010           ; Keyboard strobe
length        equ   3                 ; Three asynchronous key events are possible
              longa off
              longi off
*
* Interrupt handlers are called in native mode with 8=bit m and x.
* Interrupt handlers return with an RTL instruction.  The carry flag
* must be cleared if the interrupt was handled.
*
```

```
             phb                       ; Must preserve data bank
             phk                       ; Set data bank to program bank
             plb
             lda    >kbd               ; Get current key
             and    #%01111111         ; Mask any key-down bit
             ldx    #length-1          ; Length of list containing asynchronous keys
AsyncLoop    anop
             cmp    AsyncTable,x       ; Is key in table?
             beq    AsyncDsptch        ; Yes
             dex                       ; Next key
             bpl    AsyncLoop
*
* If the key is not found in the list of valid asynchronous key events, the interrupt
* handler must dispatch to the Event Manager keyboard interrupt handler.  The application
* initialized the jump address in the dispatcher before it installed its own keyboard
* interrupt handler.
EventKey     anop
             plb                       ; Restore data bank
             jmp    >$000000           ; If not recognized, dispatch to Event Manager
*
* This dispatcher will execute a subroutine associated with the asynchronous key event.
AsyncDsptch  anop
             txa                       ; Make a word index
             asl    a
             tax
             rep    #$20               ; 16=bit m
             lda    DsptchTable,x      ; Get dispatch table address
             pea    DsptchRet-1        ; Push return address
             pha                       ; Push dispatch address
             sep    #$20               ; 8=bit m
             rts                       ; Dispatch to subroutine (8=bit m & x)
```

```
DsptchRet      anop
               sta    >kbdstrb          ; Clear the keyboard strobe
               clc                      ; Flag that interrupt was handled
               plb                      ; Restore data bank
               rtl                      ; Return from interrupt
*
* Following is a table of valid asynchronous event keys:
AsyncTable     anop
               dc     h'03'             ; Control-C
               dc     h'13'             ; Control-S
               dc     h'1B'             ; Escape
*
* Following is a table of dispatch addresses to subroutines for handling asynchronous key
events:
DsptchTable    anop
               dc     i'CtrlC-1'        ; Control-C handler address-1
               dc     i'CtrlS-1'        ; Control-S handler address-1
               dc     i'Escape-1'       ; Escape handler address-1

CtrlC          anop
* This routine would do whatever you want for an asynchronous Control-C key event:

               rts
CtrlS          anop
* This routine would do whatever you want for an asynchronous Control-S key event:
               rts

Escape         anop
* This routine would do whatever you want for an asynchronous Escape key event:
               rts
               end
```

Journaling mechanism

The Event Manager provides a **journaling mechanism** that can be accessed through assembly language. Figure 7-3 illustrates the journaling mechanism.

Important

At the time of publication, Apple did not provide a journaling driver; you will have to write your own.

The mechanism can separate the Event Manager from the user and feed the manager events from a file. The file can contain a record of all events that occurred during some portion of a user's session. Specifically, this file records all calls to the GetNextEvent, EventAvail, GetMouse, Button, and TickCount routines.

When a journal is being recorded, every call to any of these routines is sent to a journaling device driver, which records the call (and the results of the call) in a file. When the journal is played back, these recorded calls are taken from the journal file and sent directly to the Event Manager. The result is that the recorded sequence of user-generated events is reproduced when the journal is played back.

❖ *Note:* To play back a journal, an application must make exactly the same Event Manager calls in the same order as when the journal was recorded.

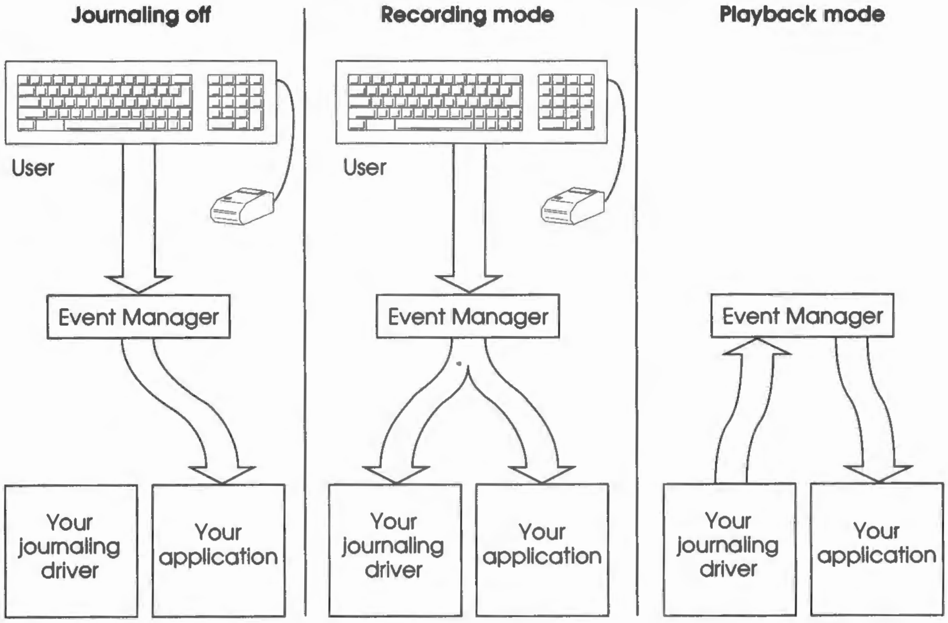


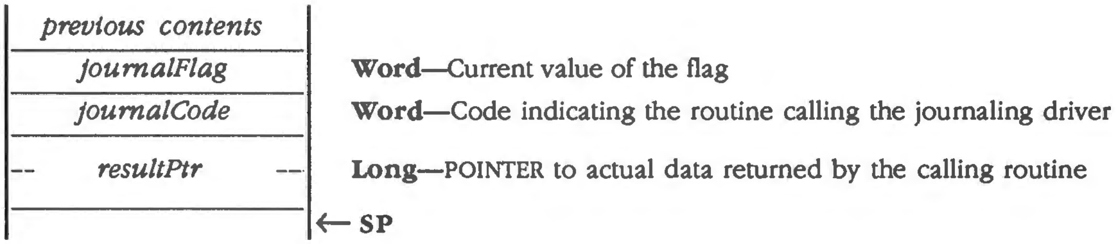
Figure 7-3
Journaling mechanism

To use journaling, you must place the address of the journaling driver in the Event Manager jump vector *journalPtr;* that is, you must place a $5C followed by the 3-byte address of the driver in *journalPtr*. The Event Manager then calls the journaling device driver by jumping through *journalPtr*. The *journalPtr* is set to $00000000 when EMStartUp is executed.

The *journalFlag* variable controls whether journaling is active, and, if so, whether it is in recording or playback mode. If *journalFlag* is set to 0, journaling is not active. If *journalFlag* is a nonzero value, journaling is active. A positive value indicates recording mode, and a negative value indicates playback mode. The *journalFlag* is set to $0000 when EMStartUp is executed. The locations of *journalPtr* and *journalFlag* should be obtained by calling the Miscellaneous Tool Set routine GetAddr.

If journaling is active, the routines GetNextEvent, EventAvail, GetMouse, Button, and TickCount will push information onto the stack and perform a JSL to the journaling device driver whose address is stored in *journalPtr*. The journaling driver should remove the information from the stack before returning with an RTL.

The information pushed onto the stack is as follows:



Word—Current value of the flag

Word—Code indicating the routine calling the journaling driver

Long—POINTER to actual data returned by the calling routine

Table 7-5 summarizes the values for *journalCode* and *resultPtr*.

Table 7-5
Journal codes and result pointers

Journal code value	Routine	Result pointer values
0	TickCount	LONG
1	GetMouse	POINT
2	Button	BOOLEAN
4	GetNextEvent	Event record
4	EventAvail	Event record

Using alternative pointing devices

❖ *Note:* You need to read this section only if you are writing a device driver for an alternative pointing device. All applications that use the Event Manager work the same with alternative pointing devices as they do with the mouse.

The Event Manager can use an **alternative pointing device,** such as a graphics tablet, light pen, or trackball, instead of the mouse. When an alternative pointing device is being used, its X-Y location and button status appear in the event records in place of the mouse information. Mouse-up and mouse-down events are posted when the alternative device's buttons change state.

More than one pointing device can also be used. In this case, whichever device is currently moving or changing state is the device whose X-Y location appears in the event records. The device that is currently moving or changing state also determines the cursor position.

For an alternative pointing device to be usable by the Event Manager, a device driver must be written for it and installed in the system.

Writing device drivers

Your device driver is called with the processor in native 8-bit mode, and the driver must exit in native 8-bit mode. If the device driver is to be installed as a Heartbeat Task, it must be written according to the instructions in Chapter 14, "Miscellaneous Tool Set." See the Heartbeat Interrupt routines in that chapter. All other device drivers must be written according to interrupt routine guidelines as found in the *Apple IIGS Firmware Reference.*

All device drivers should begin with a 6-byte header, as follows:

- BRA CodeStart (this generates 2 bytes of code)
- 2 bytes of device information
- 2 bytes initialized to $8989 (device driver signature)

If the device driver is installed as a Heartbeat Task, the driver header should be immediately after the Heartbeat Task header.

Initialize the low byte of device information as follows:

Bit 7	Set if the device is an absolute device
Bit 6	Set if the device is a relative device
Bit 5	Unused; set to 0
Bit 4	Unused; set to 0
Bit 3	Set if the device communicates through the Apple Desktop Bus
Bit 2	Set if the device communicates through the serial port
Bit 1	Set if the device has its own card and generates interrupts
Bit 0	Set if the device has its own card and does not generate interrupts

Initialize the high-order byte of device information to $FF. The startup program will then set up this byte as required by the type of device being installed, as follows:

Card device	Byte contains slot # in which the card was found
Serial device	Byte contains port # to which the device is connected
ADB device	Byte contains address # assigned to the device

A device driver should perform the following steps each time it is called:

1. Call the GetAddr routine in the Miscellaneous Tool Set to obtain the address of the relative or absolute clamp values (depending on whether the driver is for a relative or absolute device). Save the address so that you have to make the call only the first time the device driver is executed.
2. If the driver is installed as a Heartbeat Task, reset the Heartbeat Task counter to 1 or 2. Poll the device to obtain its current X-Y position and button state.

   If the driver is for a serial device, issue an InQStatus call to determine how many characters are in the serial firmware's input queue. Read the characters by calling the Serial Read routine.

   ❖ *Note:* For more information about the Serial Read routine, see the *Apple IIGS Firmware Reference.*

   If the driver is for an ADB device, the stack will have a buffer pointer at offset 7. The first byte in the buffer specifies the number of data bytes in the buffer. Read the data bytes.
3. Determine whether the device's X-Y position or button state has changed. If no changes, skip to step 11.
4. Push the following word onto the stack:

Bits 15–3	Unused; set to 0
Bit 2	Set to 1 if button state has changed; otherwise set to 0
Bit 1	Set to 1 if XY position has changed; otherwise set to 0
Bit 0	Unused; set to 0

5. Read the keyboard modifiers latch at $C025 (must be done in 8-bit mode) and push the byte onto the stack. Push a byte of 0 onto the stack.
6. Determine the device's absolute X position. Get the current X clamps, using the address saved in step 1, and clamp the X position; that is, make sure that the X position is within the clamp boundaries. Push a word containing the clamped, absolute X position on the stack.
7. Determine the device's absolute Y position. Get the current Y clamps, using the address saved above, and clamp the Y position; that is, make sure that the Y position is within the clamp boundaries. Push a word containing the clamped, absolute Y position on the stack.

8. Push the following word on the stack:

Bit 15	Current state of button 0 (1 if down, 0 if up)
Bit 14	Previous state of button 0
Bit 13	Unused; set to 0
Bit 12	Current state of button 1
Bits 11–9	Unused; set bit to 0
Bit 8	Previous state of button 1
Bits 7–0	Unused; set to 0

9. Call the Event Manager routine FakeMouse (must be called in native 16-bit mode).
10. Return to native 8-bit mode.
11. Return with an RTL.

Installing device drivers

The user installs the device driver by executing a startup program. If the startup program is a desk accessory, the user can install the driver while running an application. The startup program should initialize the device and install the device driver into the system as described in the following sections.

Devices using their own cards

If the device communicates using its own card, take the following steps to install the device driver:

1. Load the driver code into memory.
2. Determine which slot the device's card is in. Store the slot number in the appropriate byte of the device driver header (described in the section "Writing Device Drivers" in this chapter).
3. Perform any initialization needed by the device, such as setting up scaling and offset values or setting the correct operation mode.
4. Install the driver into either the heartbeat queue or the IRQ_Other interrupt vector, depending on whether the device generates interrupts.

   If the device does not generate interrupts, install the driver as a task in the heartbeat queue. Install the driver using the SetHeartBeat routine in the Miscellaneous Tool Set. (See the section "SetHeartbeat" in Chapter 14, "Miscellaneous Tool Set.")

If the device does generate the interrupts, install the driver in the IRQ_Other interrupt vector after saving the previous contents of the vector. The previous contents of the vector will be needed later when you remove the device driver. Obtain the contents of the vector by calling the GetVector routine (in the Miscellaneous Tool Set) with a reference number of $17. You can then install the driver by calling the SetVector routine (in the Miscellaneous Tool Set) with a reference number of $17. (See the sections "GetVector" and "SetVector" in Chapter 14, "Miscellaneous Tool Set.")

Devices communicating through the serial port

If the device communicates through the serial port, install the device driver by taking the following steps:

1. Load the driver code into memory.
2. Determine to which port the device is connected. Store the port number in the appropriate byte of the device driver header (described in the section "Writing Device Drivers" in this chapter).
3. Initialize the device by calling the Serial Init routine.

   ❖ *Note:* For more information about the Serial Init routine and for more details about the next two steps, consult the *Apple IIGS Firmware Reference.*
4. Install the driver in the serial firmware's completion vector by issuing a SetIntInfo call to the serial firmware. The command list for the call should specify that ***character available*** interrupts be passed to the driver.
5. Turn on buffering by calling the Serial Write routine with the following three characters: Control-I, B, E.

Devices communicating through the Apple Desktop Bus

If the device communicates through the Apple Desktop Bus (ADB), install the device driver as follows:

1. Load the driver code into memory.
2. Determine the address number assigned to the device. Store the address number in the appropriate byte of the device driver header (described in the section "Writing Device Drivers" in this chapter).
3. Install the driver in the ADB firmware's SRQ List completion vector by calling the SRQPoll routine in the ADB Tool Set (see Chapter 3, "Apple Desktop Bus Tool Set").
4. Enable SRQ for the device by calling the SendInfo routine in the ADB Tool Set.

Removing device drivers

The user removes the device driver by executing a shutdown program. If the shutdown program is a desk accessory, the user can remove the driver while running an application. The shutdown program should shut down the device and remove the device driver from the system as described in the following sections.

Devices using their own cards

If the device communicates using its own card, remove the device driver as follows:

1. Shut down the device if possible.
2. If the driver is installed in the Heartbeat queue, remove it by calling the DelHeartBeat routine in the Miscellaneous Tool Set. (See the section "DelHeartBeat" in Chapter 14, "Miscellaneous Tool Set.") If the driver is installed in the IRQ_Other interrupt vector, restore the previous contents of the vector.
3. Release all memory used by the driver code.

Devices communicating through the serial port

If the device communicates through the serial port, remove the device driver as follows:

1. Turn off buffering by calling the Serial Write routine with the following three characters: Control-I, B, D.

   ❖ *Note:* For more information about the Serial Write routine, see the *Apple IIGS Firmware Reference.*

2. Release all memory used by the driver code.

Devices communicating through the Apple Desktop Bus

If the device communicates through the Apple Desktop Bus, remove the device driver as follows:

1. Disable SRQ for the device using the SendInfo routine from the Apple Desktop Bus Tool Set. See Chapter 3, "Apple Desktop Bus Tool Set."
2. Remove the driver from the ADB firmware's SRQ List completion vector by calling the SRQRemove routine in the ADB Tool Set.
3. Release all memory used by the driver code.

$0106 EMBootInit

Initializes the Event Manager; called only by the Tool Locator.

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 Call must not be made by an application.

$0206 EMStartUp

Starts up the Event Manager, sets size of event queue, and sets minimum and maximum mouse clamp values.

Important

Your application must make this call before it makes any other Event Manager calls.

The **mouse clamp values** establish the minimum and maximum X and Y coordinates for the mouse position. Since these values are usually used to prevent the user from moving the mouse position off the screen, the clamp values input to EmStartUp are generally 0,320,0,200 for 320 mode, and 0,640,0,200 for 640 mode.

Before the Event Manager passes the clamp values to the mouse, it decrements *xMaxClamp* and *yMaxClamp* by 1.

The clamp values are also used to set the **absolute clamps** in order to support alternative pointing devices. If your application will change the relative or absolute clamps after initializing the Event Manager, the application should call either the ClampMouse or SetAbsClamp routine with the new clamp values. See Chapter 14, "Miscellaneous Tool Set," for more information about those routines.

If you change the clamp values, your application should not reinitialize the Event Manager.

Parameters

Stack before call

previous contents	
dPageAddr	**Word**—Bank $0 starting address for one page of direct-page space
queueSize	**Word**—Maximum number of event records in queue (0 = default of 20)
xMinClamp	**Word**—Minimum X clamp value for the mouse
xMaxClamp	**Word**—Maximum X clamp value for the mouse plus 1
yMinClamp	**Word**—Minimum Y clamp value for the mouse
yMaxClamp	**Word**—Maximum Y clamp value for the mouse plus 1
userID	**Word**—ID number Event Manager will use to get memory
	← **SP**

Stack after call

previous contents
← SP

Errors

$0601	`emDupStrtUpErr`	EMStartUp already called; duplicate call ignored
$0606	`emQSiz2LrgErr`	Size of event queue exceeds 3639
$0607	`emNoMemQueueErr`	Insufficient memory available for queue; Event Manager not initialized

C

```
extern pascal void
EMStartUp(dPageAddr,queueSize,xMinClamp,xMaxClamp,yMinClamp,yMaxClamp,
userID)

Word      dPageAddr;

Word      queueSize;

Integer      xMinClamp;

Integer      xMaxClamp;

Integer      yMinClamp;

Integer      yMaxClamp;

Word      userID;
```

Your application can also use the following alternate form of the call:

```
extern pascal void EMStartUp(dPageAddr,queueSize,clamp,userID)

Word      dPageAddr;

Word      queueSize;

ClampRec      clamp;

Word      userID;
```

$0306 EMShutDown

Shuts down the Event Manager and releases any workspace allocated to it.

Important

If your application has started up the Event Manager, the application must make this call before it quits.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

None

C

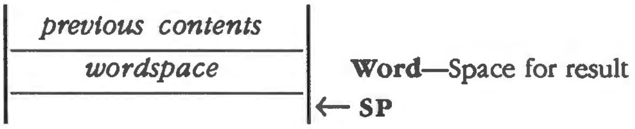
```
extern pascal void EMShutDown()
```

$0406 EMVersion

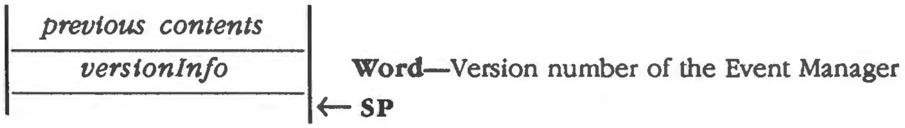
Returns the version number of the Event Manager.

Parameters

Stack before call



Stack after call



Errors

None

C

```
extern pascal Word EMVersion()
```

$0506 EMReset

Returns an error if the Event Manager is active; otherwise does nothing.

Warning
An application must not make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors $0602	`emResetErr`	Can't reset Event Manager

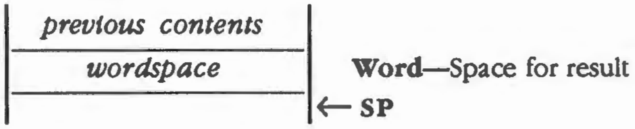
C Call must not be made by an application.

$0606 EMStatus

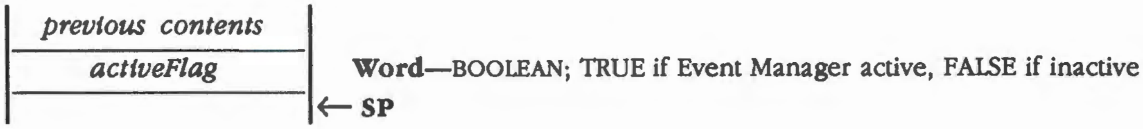
Indicates whether the Event Manager is active.

Parameters

Stack before call



Stack after call



previous contents
activeFlag
Word—BOOLEAN; TRUE if Event Manager active, FALSE if inactive
← **SP**

Errors None

C `extern pascal Boolean EMStatus()`

$0D06 Button

Returns the current state of the specified mouse button.

❖ *Note:* On a one-button mouse, the button number is 0.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for the result
buttonNum	**Word**—Number of button (0 or 1) to check
	← **SP**

Stack after call

previous contents	
buttonDown	**Word**—BOOLEAN; TRUE if button is down, or FALSE if it isn't
	← **SP**

Errors

$0605	`emBadBttnNoErr`	Button number specified is not 0 or 1

C

```
extern pascal Boolean Button(buttonNum)

Word     buttonNum;
```

$0906 DoWindows

Returns the address of the direct-page work area used by the Event Manager.

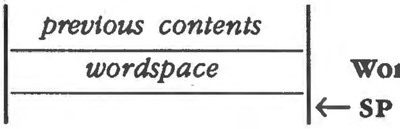
Warning

An application must never make this call.

DoWindows is called by the Window Manager when the Window Manager is initialized. The Window Manager uses the high end of the *dPageAddr* returned by DoWindows; the Event Manager uses the low end.

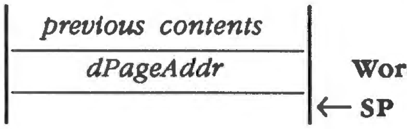
Parameters

Stack before call



Word—Space for result

Stack after call



Word—Bank $0 starting address for Event Manager's one-page work area

Errors

None

C

Call must not be made by an application.

$0B06 EventAvail

Allows an application to look at the next available event of a specified type or types. If the event is in the event queue, the event is left there for subsequent retrieval by GetNextEvent. If no event of the specified type or types is available, a null event is returned. EventAvail does not call the Desk Manager.

EventAvail follows the event priority order discussed in the GetNextEvent routine in this chapter.

A queue event returned by EventAvail will not be accessible later if, in the meantime, the queue has become full and the event has been discarded.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
eventMask	**Word**—Specifies which types of events are of interest
-- *eventPtr* --	**Long**—POINTER to event record in which event will be placed
	← **SP**

Stack after call

previous contents	
gotEventFlag	**Word**—BOOLEAN; TRUE if any specified event types available, FALSE if not
	← **SP**

Errors

None

C

```
extern pascal Boolean EventAvail(eventMask,eventPtr)

Word     eventMask;

EventRecordPtr     eventPtr;
```

$1906 FakeMouse

Allows an alternative pointing device, such as a graphics tablet, to be used in place of or in conjunction with the mouse. This call must be made only by a device driver. See the section "Using Alternative Pointing Devices" in this chapter for more information.

Parameters

Stack before call

previous contents	
changedFlag	**Word**—Indicating that device's position and/or button state has changed
modLatch | *padding*	**Byte**—Keyboard modifiers latch | **Byte**—Set to 0
xPosition	**Word**—Device's clamped absolute X position
yPosition	**Word**—Device's clamped absolute Y position
buttonStatus	**Word**—Device's button status
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

Call cannot be made from C.

$1506 FlushEvents

Removes all queue events of the type or types specified by an event mask up to but not including the first event of any type specified by a stop mask. If the event queue doesn't contain any events of the types specified by *eventMask*, FlushEvents does nothing.

To remove all events specified by *eventMask*, specify a *stopMask* of 0.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
eventMask	**Word**—Types of events to be removed from queue
stopMask	**Word**—Event types to stop removal process
	← **SP**

Stack after call

previous contents	
resultCode	**Word**—0 = all events removed from queue, or
	← **SP** event code = type of event that stopped process

Errors

None

C

```
extern pascal Word FlushEvents(eventMask,stopMask)

Word     eventMask;

Word     stopMask;
```

$1206 GetCaretTime

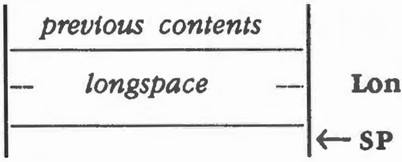
Returns the time (in ticks) between blinks of the caret (usually indicated by a vertical bar) marking the insertion point in text that can be edited.

If your application controls the blinking of the caret, the application should call this routine. On every pass through the program's main event loop, the application should check *numTicks* against the elapsed time since the last blink of the caret.

The user can adjust the *numTicks* value by changing the Cursor Flash setting in the Control Panel.

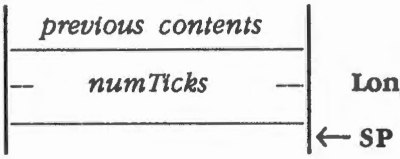
Parameters

Stack before call



Long—Space for result

Stack after call



Long—Number of ticks between blinks of the caret

Errors

None

C

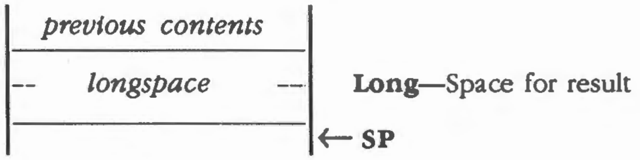
```
extern pascal LongWord GetCaretTime()
```

$1106 GetDblTime

Returns the maximum difference (in ticks) between mouse-up and mouse-down events allowed for the mouse clicks to be considered a double-click. The user can adjust the *maxTicks* value by changing the Double-Click setting in the Control Panel.

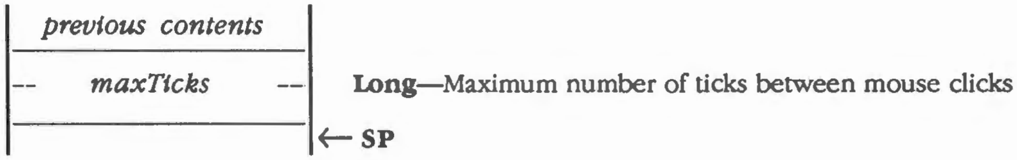
Parameters

Stack before call



Long—Space for result

Stack after call



Long—Maximum number of ticks between mouse clicks

Errors

None

C

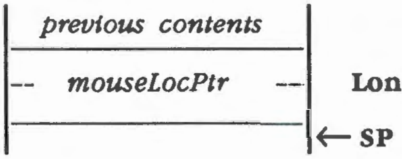
```
extern pascal LongWord GetDblTime()
```

$0C06 GetMouse

Returns the current mouse location. GetMouse gives the location in the local coordinate system of the current GrafPort (for example, the currently active window). In contrast, the mouse location stored in the *where* field of an event record is always specified in global coordinates.

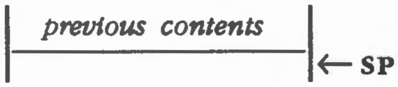
Parameters

Stack before call



Long—POINTER to POINT for current mouse location

Stack after call



Errors

None

C

```
extern pascal void GetMouse(mouseLocPtr)

Point *mouseLocPtr;
```

$0A06 GetNextEvent

Returns the next available event of a specified type or types; if the event is in the event queue, GetNextEvent removes the event from the queue.

Events in the queue that aren't designated in the mask remain in the queue. Your application can remove the events by calling the FlushEvents routine.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
eventMask	**Word**—Specifies which types of events are of interest
-- *eventPtr* --	**Long**—POINTER to the event record in which event will be placed
	← **SP**

Stack after call

previous contents	
handleEventFlag	**Word**—BOOLEAN; TRUE if event should be handled by application, FALSE if not
	← **SP**

Errors

None

C

```
extern pascal Boolean GetNextEvent(eventMask,eventPtr)

Word        eventMask;

EventRecordPtr      eventPtr;
```

(continued)

Event priority order

GetNextEvent returns the next available event of any type the mask designates, subject to the following priority order:

1. Activate event (one window becomes inactive before another window becomes active)
2. Switch event (reserved for future use)
3. Mouse-down, mouse-up, key-down, auto-key, device driver, application-defined, or desk accessory event (in the order they were posted)
4. Update event (in front-to-back order of windows)

If no event of any of the designated types is available, GetNextEvent returns a null event. This priority order is further discussed in the section "Event Priority" in this chapter.

GetNextEvent and the Desk Manager

Before reporting an event to the application, GetNextEvent calls the Desk Manager routine SystemEvent to see whether the system wants to intercept and respond to the event. If so, or if the event being reported is a null event, GetNextEvent returns a Boolean result of FALSE; a Boolean result of TRUE means that the application should handle the event itself. The Desk Manager intercepts the following events:

- Desk accessory events
- Activate and update events directed to a desk accessory
- Mouse-up and keyboard events, if the currently active window belongs to a desk accessory

In each case, the Desk Manager intercepts the event only if the desk accessory can handle that type of event. As a rule, all desk accessories should be set up to handle activate, update, and keyboard events and should not handle mouse-up events.

$1606 GetOSEvent

Returns the next available queue event of a specified type or types and removes it from the queue.

GetOSEvent returns the next available queue event of any type that the mask designates. If no event of any of the designated types is available, GetOSEvent returns a null event. GetOSEvent doesn't return window or switch events and doesn't call the Desk Manager before returning the event.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
eventMask	**Word**—Specifies which types of queue events are of interest
-- *eventPtr* --	**Long**—POINTER to the event record in which the event will be placed
	← **SP**

Stack after call

previous contents	
gotEventFlag	**Word**—BOOLEAN; TRUE if any specified event types available, FALSE if null event
	← **SP**

Errors

None

C

```
extern pascal Boolean GetOSEvent(eventMask,eventPtr)

Word     eventMask;

EventRecordPtr     eventPtr;
```

$1706 OSEventAvail

Allows an application to look at the next available queue event of a specified type or types, but leaves the event in the queue. OSEventAvail returns the next available queue event of any type the mask designates. If no event of any of the designated types is available, OSEventAvail returns a null event.

OSEventAvail doesn't return window or switch events and doesn't call the Desk Manager before returning the event.

An event returned by OSEventAvail will not be accessible later, if, in the meantime, the queue has become full and the event has been discarded.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
eventMask	**Word**—Specifies which types of queue events are of interest
— *eventPtr* —	**Long**—POINTER to the event record in which the event will be placed
	← **SP**

Stack after call

previous contents	
gotEventFlag	**Word**—BOOLEAN; TRUE if any specified event types available, FALSE if null event
	← **SP**

Errors

None

C

```
extern pascal Boolean OSEventAvail(eventMask,eventPtr)

Word      eventMask;

EventRecordPtr     eventPtr;
```

$1406 PostEvent

Posts an event into the event queue. PostEvent sets the fields of the event record as follows:

- Sets the *what* field to the specified event type
- Sets the *message* field to the specified message
- Sets the *when, where,* and *modifiers* fields to the current time, mouse location, and state of the modifier keys and mouse buttons

PostEvent should generally be used only to post application-defined events or driver events into the queue. Be careful if your application posts any other types of events into the queue. For example, an application that attempts to post an activate or update event (which aren't normally posted to the queue) will interfere with the normal operation of the Event Manager.

Important

If you use PostEvent to post a keyboard or mouse event, you must supply the state of the modifier keys and mouse buttons in the high-order word of *eventMsg*. This information will be used to set the *modifiers* field of the event record.

If you are posting a keyboard event, you must also supply the ASCII character code in the low-order byte of *eventMsg*. If you are posting a mouse event, you must also supply the button number in the low-order word of *eventMsg*.

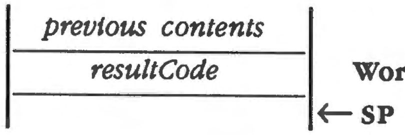
If your application uses PostEvent to repost an event that has been removed from the queue using GetNextEvent or GetOSEvent, the event time, mouse location, state of the modifier keys, and state of the mouse buttons will all be changed from the originally posted event. The meaning of the event may change in the process.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
eventCode	**Word**—Type of event to be placed in the queue
-- *eventMsg* --	**Long**—Event message
	← **SP**

Stack after call



Word—0 if event posted, 1 if event type not enabled in system event mask

Errors

$0604	`emBadEvtCodeErr`	Event code is greater than 15

C

```
extern pascal Word PostEvent(eventCode,eventMsg)
Word      eventCode;
LongWord      eventMsg;
```

$1806 SetEventMask

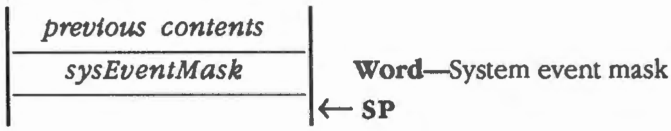
Specifies the system event mask.

The system event mask controls what types of events are posted into the event queue. The Event Manager posts only those event types that correspond to bits set in the system event mask. The Event Manager does not post activate, update, or switch events, because those events are not stored in the event queue.

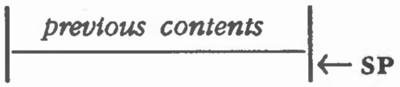
The system event mask is initially set to post all events. Your application should not change the system event mask, because desk accessories may depend upon receiving certain types of events.

Parameters

Stack before call



Stack after call



Errors

None

C

```
extern pascal void SetEventMask(sysEventMask)

Word     sysEventMask;
```

$1306 SetSwitch

Generates a switch event.

Warning

Only switcher-type applications should make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void SetSwitch()
```

$0E06 StillDown

Tests whether the specified mouse button is still down.

❖ *Note:* On a one-button mouse, the button number is 0.

Usually called after a mouse-down event, StillDown is a true test of whether the mouse button is still down from the original press.

❖ *Note:* The Event Manager Button routine is not a true test; that routine returns TRUE whenever the mouse button is currently down, even if the user has released and pressed the button again since the original mouse-down event.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
buttonNum	**Word**—Number of button (0 or 1) to check
	← **SP**

Stack after call

previous contents	
buttonDown	**Word**—BOOLEAN; TRUE if button down and no mouse events are pending
	← **SP** in event queue for specifed button, or FALSE if not

Errors

$0605 `emBadBttnNoErr` Button number specified is not 0 or 1

C

```
extern pascal Boolean StillDown(buttonNum)
Word     buttonNum;
```

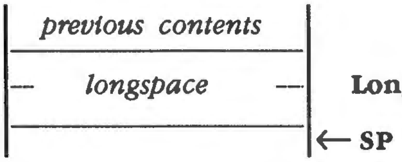
$1006 TickCount

Returns the current number of ticks (in sixtieths of a second) since the system was last started.

Your application should not depend upon an exact tick count. The tick count is incremented during the VBL interrupt, but that interrupt can be disabled. Also, because an interrupt task can keep control for more than one tick, your application should not rely on the tick count being incremented to a certain value (for example, it should not test whether the tick count has become equal to its old value plus 1). Instead, the application should check for a greater than or equal to condition.

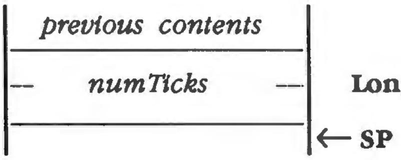
Parameters

Stack before call



Long—Space for result

Stack after call



Long—Number of ticks since system startup

Errors

None

C

```
extern pascal LongWord TickCount()
```

$0F06 WaitMouseUp

Tests whether the specified mouse button is still down. If the button is not still down from the original press, WaitMouseUp removes the preceding mouse-up event from the queue before returning FALSE.

❖ *Note:* On a one-button mouse, the button number is 0.

You could use WaitMouseUp, for example, if your application attaches some special significance to mouse double-clicks and to mouse-up events. WaitMouseUp would allow the application to recognize a double-click without being confused by the intervening mouse-up event.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
buttonNum	**Word**—Number of button (0 or 1) to check
	← **SP**

Stack after call

previous contents	
buttonDownFlag	**Word**—BOOLEAN; TRUE if button down and no mouse events are pending in event queue for the specifed button; FALSE if not
	← **SP**

Errors

$0605 `emBadBttnNoErr` Button number specified is not 0 or 1

C

```
extern pascal Boolean WaitMouseUp(buttonNum)

Word    buttonNum;
```

Event Manager summary

This section briefly summarizes the constants, data structures, and tool set errors contained in the Event Manager.

Important

These definitions are provided in the appropriate interface file.

Table 7-6
Event Manager constants

Name	Value	Description
Event codes		
`nullEvt`	$0000	Null event
`mouseDownEvt`	$0001	Mouse-down event
`mouseUpEvt`	$0002	Mouse-up event
`keyDownEvt`	$0003	Key-down event
`autoKeyEvt`	$0005	Auto-key event
`updateEvt`	$0006	Update event
`activateEvt`	$0008	Activate event
`switchEvt`	$0009	Switch event
`deskAccEvt`	$000A	Desk accessory event
`driverEvt`	$000B	Device driver event
`app1Evt`	$000C	Application-defined event
`app2Evt`	$000D	Application-defined event
`app3Evt`	$000E	Application-defined event
`app4Evt`	$000F	Application-defined event
Event masks		
`mDownMask`	$0002	Call applies to mouse-down events
`mUpMask`	$0004	Call applies to mouse-up events
`keyDownMask`	$0008	Call applies to key-down events
`autoKeyMask`	$0020	Call applies to auto-key events
`updateMask`	$0040	Call applies to update events
`activeMask`	$0100	Call applies to activate events
`switchMask`	$0200	Call applies to switch events
`deskAccMask`	$0400	Call applies to desk accessory events
`driverMask`	$0800	Call applies to device driver events
`app1Mask`	$1000	Call applies to application-defined app1Evt events
`app2Mask`	$2000	Call applies to application-defined app2Evt events
`app3Mask`	$4000	Call applies to application-defined app3Evt events
`app4Mask`	$8000	Call applies to application-defined app4Evt events
`everyEvent`	$FFFF	Call applies to all events

Table 7-6 (continued)
Event Manager constants

Name	Value	Description
Journal codes		
`jcTickCount`	$00	TickCount call
`jcGetMouse`	$01	GetMouse call
`jcButton`	$02	Button call
`jcEvent`	$04	GetNextEvent and EventAvail calls
Modifier Flags		
`activeFlag`	$0001	Set if window was activated
`changeFlag`	$0002	Set if active window changed state
`btn1State`	$0040	Set if button 1 was up
`btn0State`	$0080	Set if button 0 was up
`appleKey`	$0100	Set if Apple key was down
`shiftKey`	$0200	Set if Shift key was down
`capsLock`	$0400	Set if Caps Lock key was down
`optionKey`	$0800	Set if Option key was down
`controlKey`	$1000	Set if Control key was down
`keyPad`	$2000	Set if keypress was from key pad

Table 7-7
Event Manager data structures

Name	Offset	Type	Definition
Event record			
what	$00	Word	Event code
message	$02	Long	Event message
when	$06	Long	Tick count
where	$0A	Point	Mouse location
modifiers	$0E	Word	Modifier flags

Note: The actual assembly-language equates have a lowercase *o* (the letter) in front of all of the names given in this table.

Table 7-8
Event Manager error codes

Code	Name	Description
$0601	`emDupStrtUpErr`	EMStartUp already called
$0602	`emResetErr`	Can't reset Event Manager
$0603	`emNotActErr`	Event Manager not active
$0604	`emBadEvtCodeErr`	Event code is greater than 15
$0605	`emBadBttnNoErr`	Button number specified is not 0 or 1
$0606	`emQSiz2LrgErr`	Size of event queue is greater than 3639
$0607	`emNoMemQueueErr`	Insufficient memory available for queue
$0681	`emBadEvtQErr`	Event queue damaged—fatal system error
$0682	`emBadQHndlErr`	Queue handle damaged—fatal system error

Chapter 8

Font Manager

This chapter describes the **Font Manager,** the tool set that allows an application to use different fonts. A **font** is a complete set of characters of one typeface. If you want to allow the user to choose from all of the fonts available when the application is run, or if you're developing an application that requires a specific font, you can use the Font Manager.

As its name implies, the Font Manager is responsible for managing fonts. The Font Manager keeps lists in memory as to how many fonts are available and what their characteristics are and provides this information to applications. The expression **available font** means that the font is one of the following:

- The **ROM font** (at the time of publication, the ROM font is Shaston 8)
- A font that is in the FONTS subdirectory
- A font that the application tells the Font Manager about directly with the AddFontVar tool set routine

Other fonts may be on disk or may be put into use directly with the QuickDraw II SetFont routine; those fonts are not available to the Font Manager.

When your application requests a font, the Font Manager searches all available fonts for the closest fit, loads the font from disk if necessary, possibly scales an existing font to create a better fit, and gives QuickDraw II the information it needs to use the font selected.

Important

For the Font Manager to find fonts, the SYSTEM subdirectory must contain a subdirectory called FONTS. This subdirectory must contain all fonts (except the ROM font) that are to be available to applications.

QuickDraw II is then responsible for the actual drawing of the characters in the currently installed font. Thus, you can think of the Font Manager as a higher-level tool set that installs a font and directs QuickDraw II to use that font until directed otherwise. For details about how fonts are drawn and constructed, see Chapter 16, "QuickDraw II," in Volume 2.

A preview of the Font Manager routines

To introduce you to the capabilities of the Font Manager, all Font Manager routines are grouped by function and briefly described in Table 8-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the Font Manager routines (discussed in alphabetical order).

Table 8-1
Font Manager routines and their functions

Routine	Description
Housekeeping routines	
FMBootInit	Initializes the Font Manager; called only by the Tool Locator—must not be called by an application
FMStartUp	Starts up the Font Manager for use by an application
FMShutDown	Shuts down the Font Manager
FMVersion	Returns the version number of the Font Manager
FMReset	Returns an error if the Font Manager is active
FMStatus	Indicates whether the Font Manager is active
Font family routines	
CountFamilies	Returns the total number of distinct font families currently available to the Font Manager that match a given specification (either *all families* or *all families that have a plain-styled font available*)
FindFamily	Returns the number and name of a particular font family
GetFamInfo	Returns the family name and characteristics of a font family with a specified family number
GetFamNum	Returns the family number corresponding to a specified font family name
AddFamily	Enables the application to add a family number and name to the Font Manager's list of known font families
Font routines	
InstallFont	Finds a specified font or the available font with the best fit if the specified font isn't available; loads the font into memory, if necessary; possibly creates a new, scaled font to match the specified font's size; and makes the resulting font current and unpurgeable
SetPurgeStat	Makes a specified font in memory purgeable or unpurgeable

Table 8-1 (continued)
Font Manager routines and their functions

Routine	Description
Font routines	
CountFonts	Returns the number of fonts currently available to the Font Manager that fit a specified description
FindFontStats	Returns the font ID and the characteristics of a particular font
LoadFont	Loads a specified font into memory (if it is not already there) and makes it current and unpurgeable
LoadSysFont	Makes the system font the current font
FMSetSysFont	Makes a specified font the system font
FMGetSysFID	Returns the font ID of the system font
FMGetCurFID	Returns the font ID of the current font
AddFontVar	Enables the application to add a variation of a preexisting font family to the Font Manager's collection of available fonts
Menu and dialog routines	
FixFontMenu	Appends the names of available font families to a specified menu
ChooseFont	Displays a dialog box enabling the user to select a new font, size, and style
ItemID2FamNum	Translates a menu item ID into a font family number
FamNum2ItemID	Translates a font family number into a menu item ID

Font records and font families

Fonts for the Apple IIGS font are grouped into font families. Each font family has a name and a number. Further, each font in a family has a characteristic style and size. This section discusses the family names, family numbers, font styles, and font sizes, and some other preliminary concepts.

Family names and numbers

Each font belongs to a **font family**. The family has a name, such as Geneva, Helvetica, or Times. All fonts with the same name belong to the same font family. Different fonts in the same family are called variations of that family. For example, 10-point Helvetica, 12-point Helvetica in boldface, and 36-point Helvetica in boldface and underlined are all variations of the Helvetica font family.

The font's **family name** is restricted to 25 letters. A string of length 0 is not permitted. Two family names are considered to be the same only if they are identical; thus, the following are all different family names:

newYork   newyork   NewYork   New York

The family name of the font built into ROM is Shaston.

A font family also has an identifying number, called the **family number.** This is a number identifying a font family, independent of point size or style modifications (so London 9, London Bold 14, and London Bold Underline 24 would all have the same family number). A family number of $0000 can be used to specify the system font for most Font Manager calls; $FFFF is an invalid family number.

At the time of publication, the font family numbers are as shown in Table 8-2.

Table 8-2
Font family numbers

Number	Font family	Number	Font family
$0002 #2	newYork	$000B #11	cairo
$0003 #3	geneva	$000C #12	losAngeles
$0004 #4	monaco	$0014 #20	times
$0005 #5	Venice	$0015 #21	helvetica
$0006 #6	london	$0016 #22	courier
$0007 #7	athens	$0017 #23	symbol
$0008 #8	sanFran	$0018 #24	taliesin
$0009 #9	toronto	$FFFE #65524	Shaston

There is a one-to-one correspondence between family numbers and family names. That is, any two fonts with the same family number should have the same family name.

Font size

The size of a font is measured in **points.** A point is a measure borrowed from the typesetting world. In that world, a point is equal to about 1/72 of an inch.

❖ *Note:* Because measurements can't be exact on a bit-mapped output device, the actual font size may be different than what it would be in normal typography. Also be aware that two fonts with the same font size but from different font families may not appear to be the same size. Font size is more useful for distinguishing different sizes of the same font family.

The Font Manager defines the **font size** as a byte. Font size is specified as an unsigned number in the range 1–255 (0 is not allowed).

❖ *Note:* In the Apple IIGS font record and in QuickDraw II, font size is allocated an entire word, with only the low-order byte being meaningful. See Chapter 16, "QuickDraw II," in Volume 2 for more information about font records.

Font style

The **font style** is the style in which the font was designed. The Font Manager defines the font style as a byte, as shown in Figure 8-1.

❖ *Note:* In the Apple IIGS font record and in QuickDraw II, font style is allocated an entire word, with only the low-order byte being meaningful. See Chapter 16, "QuickDraw II," in Volume 2 for more information on font records.

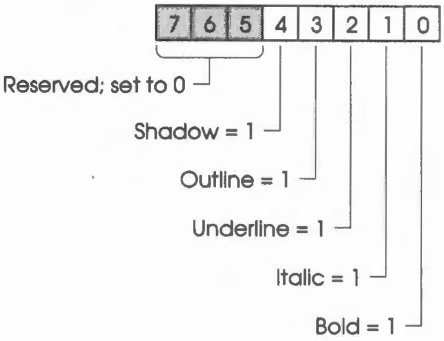


Figure 8-1
Font style byte

Important
Shadow, outline, and italic styles are available only if QuickDraw II Auxiliary has been loaded and started up.

A 1 (*on*) bit means the style is enabled; a value of $00000000 means plain style. The style bits are cumulative; for example, $0014 means shadowed and underlined.

The style byte allows you to use fonts that have been **prestyled.** Some fonts don't look very good when styled by the algorithms used by QuickDraw II. Thus, the Font Manager has been designed to handle fonts that have a certain style or combination of styles built into the particular font's design. By using such a prestyled font, your application can also improve text drawing speed if that style is going to be used frequently.

However, there are some disadvantages to prestyled fonts. First, they consume extra memory and disk space. Second, when used with strings of more than one character, a styled font can be inferior to a font produced by letting the software perform on-the-fly styling of a plain font, depending on the font and styles being used (fonts that kern are particularly vulnerable; see the discussion of fonts in Chapter 16, "QuickDraw II," in Volume 2).

You must decide on a case-by-case basis whether to create a prestyled font or to let the text-drawing routines handle the styling. Usually, leaving styling to the text-drawing routines is best. However, if you are using large fonts and screen quality is very important, it may be worthwhile to create fonts that are properly boldfaced, shadowed, and so on.

If a particular font has been prestyled for a particular style and that style has also been set in the GrafPort's *txFace* field, then the styling is not applied during text drawing. For example, fonts that have been prestyled as italic are not further italicized by QuickDraw II, regardless of the current value of *txFace.* If several styles are requested, any styles not already built into the font are applied. Also, QuickDraw II has no method for removing the styles from the current font if the font happens to have a built-in style that is not enabled by *txFace.*

For this reason, the best-fit font algorithm never chooses a font that has a style enabled that was not requested, even if it's an excellent match in other ways. This relationship between styles is called a ***partial match.*** That is, style A is considered to partially match style B if all of the styles enabled in style A are also enabled in style B. (For example, the style ***bold underline*** partially matches the style ***bold italics underline.*** The style ***plain*** partially matches all styles.) This concept is used in such calls as InstallFont and FindFontStats.

Font ID record

A **font ID record** specifies a font by family, style, and size. It is assumed that no two fonts match in all three of these characteristics; if two fonts do match in all three, the Font Manager won't be able to distinguish between them and may not be able to handle them correctly. The font ID record is used only by the Font Manager; that is, QuickDraw II does not use the record.

The structure of the font ID record is illustrated in Figure 8-2.

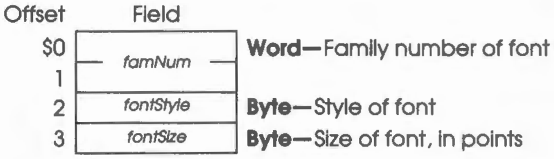


Figure 8-2
Font ID record

❖ *Note:* If you use the *txFace* and *txStyle* fields of the GrafPort to determine the *fontStyle* and *fontSize,* be careful; *txFace* and *txStyle* are words with their information in the low-order byte, and *fontStyle* and *fontSize* are bytes.

Many Font Manager routines require a font ID record as an input parameter.

Base families

A family is called a **base family** if it is the ROM font or if a plain-styled example of the family can be found among the fonts in the FONTS subdirectory. The word *base* is used because the Font Manager can build any style variation of the family at any time; that is, the family is not restricted to a particular style (because a plain font can be arbitrarily styled) and is always available (because a plain example is in the FONTS subdirectory, which is assumed always to be available). The plain-style examples of the family can thus serve as bases for any style variation of the family. (This does not rule out the use of prestyled fonts, if the application wants them.)

This definition of a base family is included because, in a typical application that allows users to choose from available fonts (a text-editing application or a paint program, for example), the application usually lists the base families in its font menu. Any other family might not be available in plain (and therefore arbitrarily styleable) form. However, this is not binding on the application; the application can put anything it wants in its font menu.

Real and scaled fonts

The Font Manager depends on the concept of real and unreal fonts for some capabilities. A **real font** exists on disk or was added by an application and marked as *real* (by a cleared *unRealBit;* see the section "FontStatBits Flag" in this chapter).

An **unreal font** is one that was scaled by the Font Manager from a real font of a different size or was added by an application and marked as *unreal.* **Scaling** means to take every character of a real font and make them bigger or smaller as appropriate to generate the requested font. For example, if your application specified a font of Courier 20, the Font Manager would, when it generated the font, make each Courier 10 character twice as big to generate the new characters. An example of such a multiplication is shown for a *T* character in Figure 8-3.

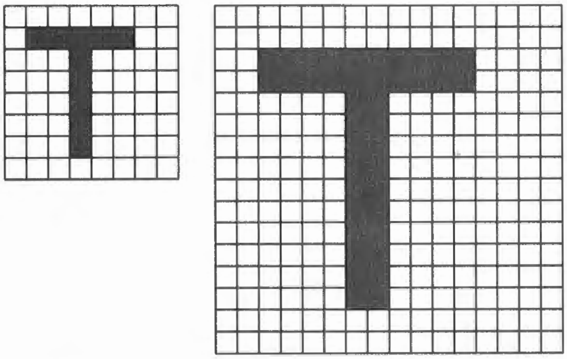


Figure 8-3
Font scaling

The Font Manager scales all the characters for the new font when it installs the font; that is, rather than generating each character as it is needed, the Font Manager scales a character, adds it to the new font, scales another character and adds it, and so on until all the characters in the real font have been scaled and added to the new font.

Fonts that have been algorithmically scaled generally do not look good as the original font, and scaling a scaled font produces even less attractive results. For this reason, when the Font Manager scales a font, it automatically marks that font as unreal. The Font Manager will not attempt to scale an unreal font; instead, it will scale from the original font.

When your application adds a font with the AddFontVar, you have a choice as to whether to mark the font as real, and thus allow further scaling, or unreal, and not allow further scaling. For more information about how to mark a font as real or unreal, see the sections "FontStatBits and FontSpecBits Bit Flags" and "AddFontVar" in this chapter. For more information about how the Font Manager uses real and unreal fonts, see the section "Best-Fit Font Algorithm" in this chapter.

Current and system fonts

The Font Manager also can affect the current and system fonts. The **current font** is the one that is currently being used by QuickDraw II to draw text; the **system font** is the one that QuickDraw II uses as the default current font when a new GrafPort is opened. The two fonts can be the same; in fact, the default current font is set by the FMStartUp routine to be the system font. If you want your application to use some other font, your application must use the Font Manager routines InstallFont, LoadFont, or ChooseFont to set the desired font as the current font.

FontStatBits and FontSpecBits bit flags

The Font Manager must maintain up-to-date information concerning all the fonts it knows about—not only the font ID and name, but also whether the font is currently in memory, whether it's a scaled version of another font, whether it was generated by the application, and so on. Some Font Manager routines allow the application to specify some of this information as a way of restricting the range of fonts that the call should consider available; that is, only fonts of a certain family, only fonts currently in memory, and so on.

Other calls return this kind of status information about a font to the application. Because many calls deal with the same kind of information, two bit flags are defined: **FontStatBits,** which reports on the status of a font, and **FontSpecBits,** which restricts the range of fonts available to the calling routine.

FontStatBits flag

The values for the various bits in FontStatBits are shown in Figure 8-4. The effects of the FontStatBits are cumulative; for example, $0011 means that the font is purgeable, real, and currently in memory.

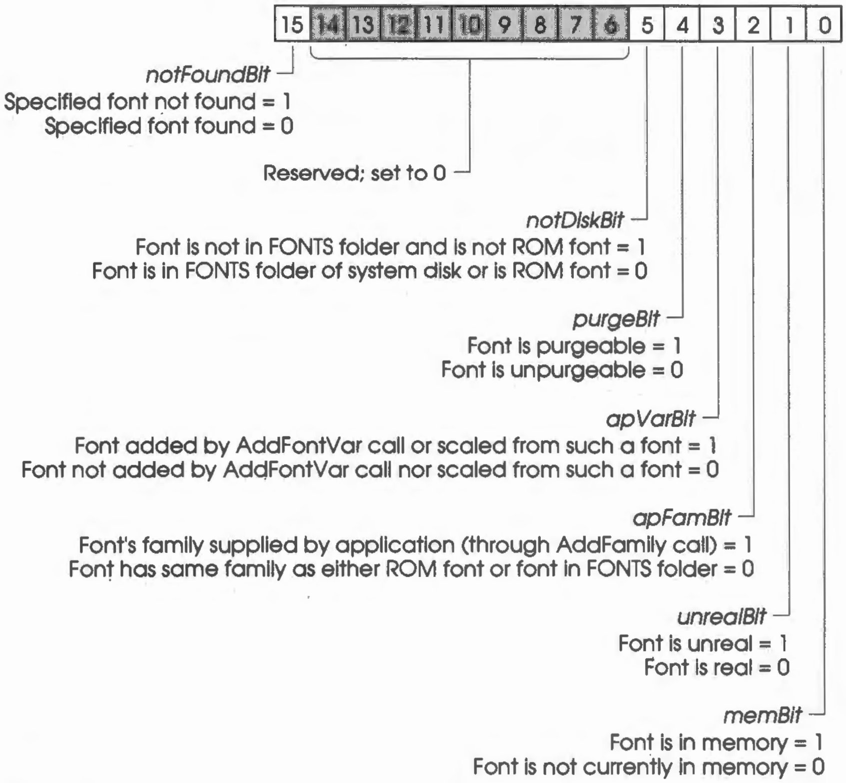

15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
notFoundBit
Specified font not found = 1
Specified font found = 0
Reserved; set to 0
notDiskBit
Font is not in FONTS folder and is not ROM font = 1
Font is in FONTS folder of system disk or is ROM font = 0
purgeBit
Font is purgeable = 1
Font is unpurgeable = 0
apVarBit
Font added by AddFontVar call or scaled from such a font = 1
Font not added by AddFontVar call nor scaled from such a font = 0
apFamBit
Font's family supplied by application (through AddFamily call) = 1
Font has same family as either ROM font or font in FONTS folder = 0
unrealBit
Font is unreal = 1
Font is real = 0
memBit
Font is in memory = 1
Font is not currently in memory = 0

Figure 8-4
FontStatBits values

Some of the Font Manager routines use FontStatBits as input but use only some of the bits. In these cases, the unused bits should be set to 0.

If *notFoundBit* is 1, the others are undefined. See the section "Using the Font Manager" in this chapter for information about the use of *notFoundBit*.

Whenever a font is made current—whether it is brought in from disk, scaled from another font, or handed over by the application—it is made unpurgeable. It can be made purgeable only by a SetPurgeStat call.

If *memBit* is 0, *purgeBit* is meaningless.

When the Font Manager scales a font, the *apVarBit* of the scaled font is set to be the same as that of the old font. Similarly, when a variant font of the family is created, either by Font Manager scaling or by the AddFontVar call, the *apFamBit* of the new font is set to be the same as that of the old font.

When the application adds a font with the AddFontVar call, it can set the *unrealBit* to indicate whether the font is to be considered real or unreal.

The FontStatRec, as shown in Figure 8-5, is defined for the convenience of some calls that return it. This record represents everything, aside from the name, that the Font Manager knows about a particular font.

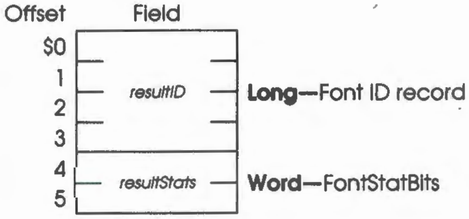


Figure 8-5
FontStatRec

FontSpecBits flag

The FontSpecBits flag is used by certain Font Manager calls to restrict the range of fonts available to that call. The effects of the FontSpecBits are cumulative; for example, $0011 means "allow only fonts in memory with the specified family number and style, real or unreal, any size."

The values for the various bits in the FontSpecBits flag are shown in Figure 8-6.

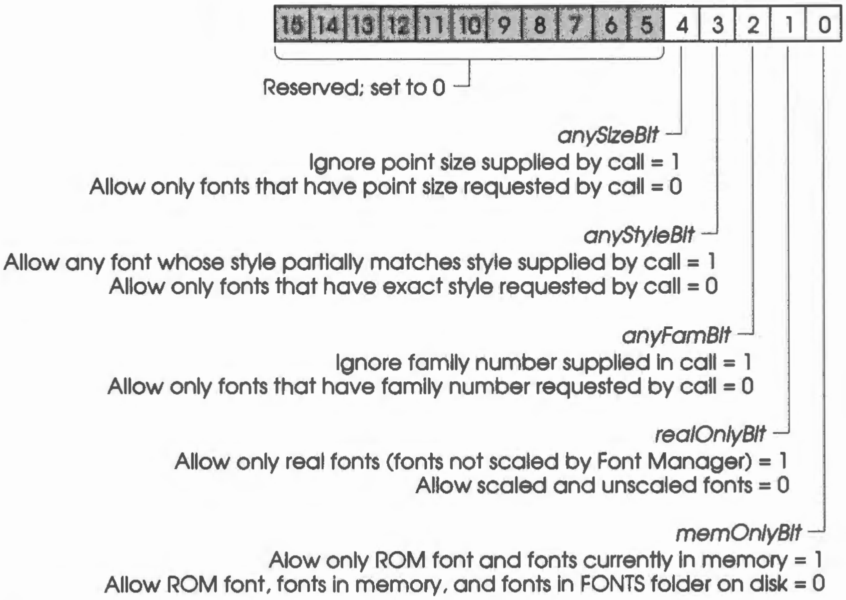

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Reserved; set to 0
anySizeBit
Ignore point size supplied by call = 1
Allow only fonts that have point size requested by call = 0
anyStyleBit
Allow any font whose style partially matches style supplied by call = 1
Allow only fonts that have exact style requested by call = 0
anyFamBit
Ignore family number supplied in call = 1
Allow only fonts that have family number requested by call = 0
realOnlyBit
Allow only real fonts (fonts not scaled by Font Manager) = 1
Allow scaled and unscaled fonts = 0
memOnlyBit
Alow only ROM font and fonts currently in memory = 1
Allow ROM font, fonts in memory, and fonts in FONTS folder on disk = 0

Figure 8-6
FontSpecBits values

The calls that use the FontSpecBits bit flag generally also need a *fontID* parameter, which consists of a family number, style, and point size.

❖ *Note:* Even if ***anyStyleBit*** is set to 1, only styles that partially match the requested style will be supplied. If you want the style parameter to be completely ignored, set ***anyStyleBit*** to 1 and the style byte in the *fontID* parameter of the call to $FF, which is partially matched by all font styles.

FamStatBits and FamSpecBits bit flags

The Font Manager also hunts for or reports on various font families. Thus, the FamStatBits and FamSpecBits bit flags are defined in order to provide information about the families. Some of the information in these flags is closely analogous to the font information in the FontStatBits and FontSpecBits flags and maintains corresponding bit positions.

FamStatBits reports on the status of a font family; **FamSpecBits** restricts, for certain Font Manager calls, the range of families available to the call.

FamStatBits flag

The values for the various bits in the FamStatBits flag are shown in Figure 8-7. The effects of the FamSpecBits are cumulative; for example, $0024 means that the family was supplied by the application and is not a base family.

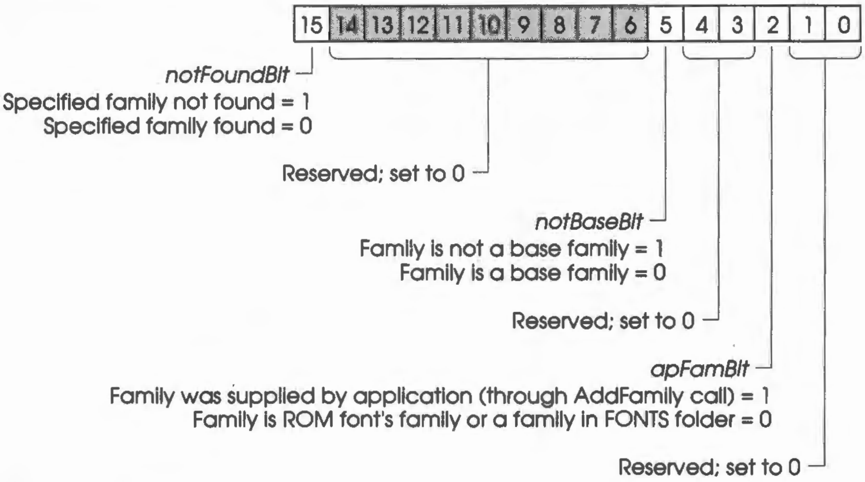

15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
notFoundBit
Specified family not found = 1
Specified family found = 0
Reserved; set to 0
notBaseBit
Family is not a base family = 1
Family is a base family = 0
Reserved; set to 0
apFamBit
Family was supplied by application (through AddFamily call) = 1
Family is ROM font's family or a family in FONTS folder = 0
Reserved; set to 0

Figure 8-7
FamStatBits values

See the section "Using the Font Manager" in this chapter for information about the use of *notFoundBit.*

FamSpecBits flag

The values for the various bits in the FamSpecBits flag are shown in Figure 8-8.

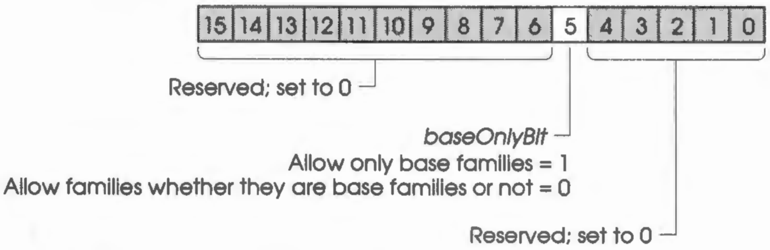
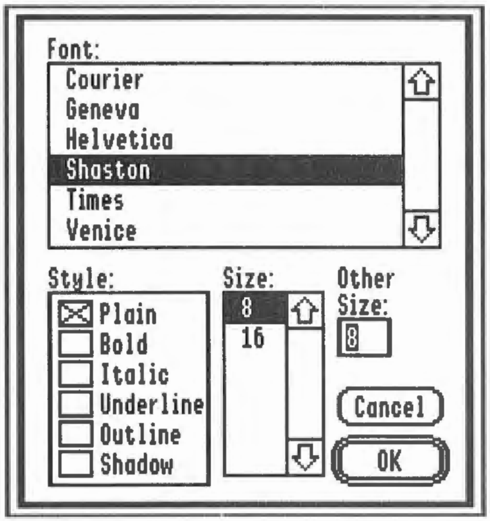

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Reserved; set to 0
baseOnlyBit
Allow only base families = 1
Allow families whether they are base families or not = 0
Reserved; set to 0

Figure 8-8
FamSpecBits values

Interaction with the user

The ChooseFont routine draws the dialog box illustrated in Figure 8-9 in 320 mode (example fonts are shown, and the box for 640 mode is similar).


Font:
Courier
Geneva
Helvetica
Shaston
Times
Venice

Style:
☒ Plain
□ Bold
□ Italic
□ Underline
□ Outline
□ Shadow

Size:
8
16

Other Size: 8

Figure 8-9
ChooseFont dialog box

After the dialog box is drawn, the user can select a new font family, style, and size. When a new family is selected, the list of sizes is updated to contain all available sizes for the selected family. The size list can hold a maximum of 12 sizes; however, any valid size can be entered in the Other Size text entry area. Keyboard input other than numbers, Return, Backspace, Left Arrow, Right Arrow, Control-F, Control-X, or Control-Y causes a beep. Clicking the mouse outside the dialog box also causes a beep.

The user can confirm or cancel the new selection by clicking the OK or Cancel button. If the user confirms a valid selection, ChooseFont automatically installs the selected font by calling the InstallFont routine.

Important

Besides the font family, font style, and font size supplied by the user, Choosefont also supplies a *scaleWord* parameter of 0 to the InstallFont routine. See the section "InstallFont" in this chapter for more information.

Using the Font Manager

This section discusses how the Font Manager routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

The Font Manager depends upon the presence of the tool sets shown in Table 8-2 and requires that at least the indicated version of the tool set be present.

Table 8-3
Font Manager—other tool sets required

Tool set number		Tool set name	Minimum version needed
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0
$04	#04	QuickDraw II	1.1
$0B	#11	Integer Math Tool Set	1.0

In addition to these tool sets, the FixFontMenu and ChooseFont routines require additional tool sets, as detailed under those routines in this chapter. Also, if you are using the shadowed, outlined, or underlined styles, QuickDraw II Auxiliary (tool set number $12 or #18) must be loaded and started up.

The first Font Manager call that your application must make is FMStartUp. FMStartUp searches the FONTS subdirectory in the SYSTEM subdirectory and compiles lists of the fonts and font families in that subdirectory.

Important

The FONTS subdirectory must exist and must contain all fonts (except the ROM font) that are to be available to applications through the Font Manager.

When you quit your application, you must make the FMShutDown call.

If you're using the Font Manager to allow the user to choose from the available fonts, the routines you'll be most concerned with are FixFontMenu and ChooseFont. FixFontMenu appends the names of the available font families onto a specified menu. ChooseFont allows the user to select a new font family, size, and style and then loads the font into memory.

Certain Font Manager routines—FindFamily, GetFamInfo, GetFamNum, FindFontStats, and LoadFont—return information about some family or font which is specified in their input parameters. In each case, the call is designed to return useful information even when the specified family or font does not exist. For example, GetFamNum can be used to determine whether a particular family name is currently in use. If the name is in use, the call returns the corresponding family number; if the name isn't in use, the call returns $FFFF (the illegal family number).

Another example is that FindFontStats is supposed to return information about the *n*th font matching certain specifications (given as input parameters to the call), where *n* = *positionNum* (another input parameter). If you wish to examine all such fonts in sequence, you do not have to first make a CountFonts call to find out how many such fonts there are and then call FindFontStats that many times. Instead, you only have to call FindFontStats with *positionNum* = 1,2,3, . . ., examining each specification, until FindFontStats informs you that there aren't any more fonts—your value of *positionNum* is greater than the number of available fonts. FindFontStats signals that there aren't any more fonts by setting the *notFoundBit* in the *resultStats* field of the returned FontStatRec. This saves you the call to CountFonts.

Because these calls are intended to give useful information even when a hunt for a family or font fails, they do not treat the condition of *no such family or font* as an error. No error code is returned. The *no such family or font* condition can always be detected by the value of a valid output parameter returned by the calls.

On the other hand, AddFontVar, SetPurgeStat, and FMSetSysFont generate errors if the entities they expect to find do not exist. These routines are not designed to yield information other than the error codes in these cases.

You shouldn't use the QuickDraw II SetFont call to make a font current when using the Font Manager; the call doesn't communicate everything needed for Font Manager information calls (for example, it doesn't associate a name with the font). However, SetFont, along with the QuickDraw II calls GetFont, SetFontID, and GetFontID, is handy for saving and restoring a GrafPort's font status, even if the fonts in question were originally installed using Font Manager calls.

Certain fonts being used for exotic purposes—like the specially designed control icon font used by the Control Manager—will probably continue to be invoked with SetFont, bypassing the Font Manager altogether. You can use this technique, but the Font Manager won't be able to help you. See Chapter 16, "QuickDraw II," in Volume 2 for more information.

Best-fit font algorithm

When the InstallFont routine installs a font, the routine uses the **best-fit font algorithm** to look for a font matching the specified font ID. The font ID record (called the *desiredID* in InstallFont's parameter list) has three fields: *desiredFam, desiredStyle,* and *desiredSize.* The following is the best-fit algorithm used by InstallFont at the time of publication (this algorithm may be improved in the future):

1. InstallFont first looks for a font with the same family and size and a style word that partially matches *desiredStyle.* If there is more than one such font, the one with the most styles enabled is used. The intent is that the font chosen has as many styles in common with *desiredStyle* as possible, without having any style enabled that the requested font doesn't have.
2. If the routine does not find a font as described in step 1, the routine narrows its search to only real (that is, not scaled) fonts.
3. InstallFont looks for a font with the same family number, twice the size, and a partially matching style word. If there is more than one such font, the one with the most styles enabled is used.
4. If the routine does not find any fonts as described in step 3, the routine looks for a font with the same family number, half the size, and a partially matching style word. In the case of a tie, the font with the most styles enabled is used.
5. If InstallFont still hasn't found any suitable font, the routine looks for a font with the same family number, a larger size than the requested size, and a partially matching style word. If the routine finds more than one such font, the one with the smallest size is used. If the routine finds more than one such smallest size, the font with the most styles enabled is used.

6. If there's still no match, the routine looks for a font with the same family number, a smaller size than the requested size, and a partially matching style word. If the routine finds more than one such font, the one with the largest size is used. If the routine finds more than one such largest size, the one with the most styles enabled is used.

❖ *Note:* In steps 1–6, it is noted that, in case of a tie, the font with the most styles enabled is used. Of course, a tie may still occur, with two fonts that have the same number of, but different, styles enabled. In such a case, a font in memory is selected over one on disk. If this rule still doesn't determine the font to be used, the last font found is used.

7. If there is no font with the requested family number, the system font is selected.

If the winner of this font hunt has a different size than the requested size, and the last bit of *scaleWord* is 0, then InstallFont creates the desired font by scaling the best-fit font. The section "InstallFont" in this chapter discusses which fonts are purgeable after this point and which are unpurgeable.

$011B FMBootInit

Initializes the Font Manager; called only by the Tool Locator.

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 Call must not be made by an application.

$021B FMStartUp

Starts up the Font Manager for use by an application.

Important

Your application must make this call before it makes any other Font Manager calls.

When you make the call, the Font Manager searches the FONTS subdirectory and makes lists of the following:

- All unique font families
- All family names
- All actual fonts in that subdirectory
- The file names of the fonts

Important

If the FONTS subdirectory does not exist, FMStartUp causes a ProDOS error, and the Font Manager is not initialized.

The Font Manager AddFamily and AddFontVar routines may add fonts, families, and names to this list. If the family names of any of the fonts found in the FONTS subdirectory are more than 25 characters long, the names are truncated to 25 characters.

Important

FMStartUp sets the system font to be the same as the built-in ROM font. Your application can change the system font by calling the Font Manager routine FMSetSysFont after the Font Manager has been started up. You should not use the QuickDraw II SetSysFont routine to change the system font while the Font Manager is in use, because the Font Manager's information doesn't reflect the changes made by SetSysFont.

Parameters

Stack before call

previous contents	
userID	**Word**—ID number of the application
dPageAddr	**Word**—Bank $0 starting address of one page of direct-page space
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$1B01	`fmDupStartUpErr`	Duplicate FMStartUp call
Memory Manager errors		Returned unchanged
ProDOS errors		Returned unchanged

C

```
extern pascal void FMStartUp(userID,dPageAddr)

Word     userID;

Word     dPageAddr;
```

$031B FMShutDown

Shuts down the Font Manager. All application-generated fonts are made purgeable. All other fonts the Font Manager knows about are disposed of if they are in memory, and then all the memory used by the Font Manager itself is released as well.

FMShutDown also sets the system font to be the same as the ROM font.

Important
If your application has started up the Font Manager, the application must make this call before it quits.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

$1B03	`fmNotActiveErr`	Font Manager not active

C

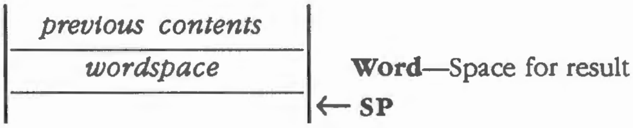
```
extern pascal void FMShutDown()
```

$041B FMVersion

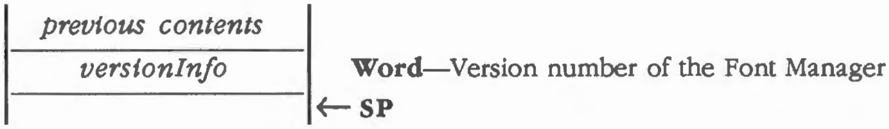
Returns the version number of the Font Manager.

Parameters

Stack before call



Stack after call



Errors

None

C

```
extern pascal Word FMVersion()
```

$051B FMReset

Resets the Font Manager; called 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

$1B02	`fmResetErr`	Can't reset the Font Manager

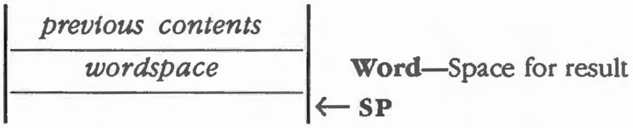
C Call must not be made by an application.

$061B FMStatus

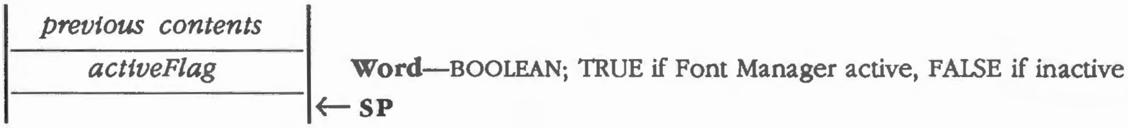
Indicates whether the Font Manager is active.

Parameters

Stack before call



Stack after call



previous contents
activeFlag
Word—BOOLEAN; TRUE if Font Manager active, FALSE if inactive
← **SP**

Errors None

C

```
extern pascal Boolean FMStatus()
```

$0D1B AddFamily

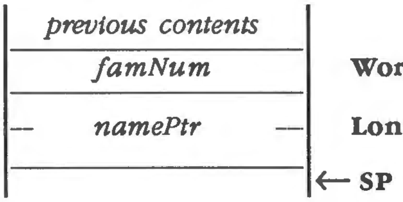
Enables the application to add a family number and name to the Font Manager's list of known families. Both *famNum* and the string pointed to by *namePtr* must be unique (that is, not already on the Font Manager's list). The Font Manager doesn't check for this; if there's any doubt, you can use the GetFamNum and GetFamInfo routines to find out. Families added with this call have the *notBaseBit* and *apFamBit* of FamStatBits both set to 1.

The string pointed to by *namePtr* must have a length byte as the first byte. If the name is greater than 25 characters long, the name is truncated to 25 characters. If the name has a length of 0, an error is returned.

After AddFamily is executed, the Font Manager knows about the family number and name, but won't have any fonts of that family available. You can add them using the AddFontVar routine; once some variations are in memory, more may be generated by the Font Manager's scaling algorithm.

Parameters

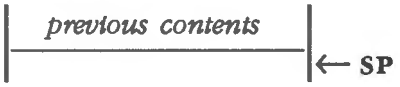
Stack before call



Word—Family number

Long—POINTER to family name

Stack after call



Errors

$1B08	fmBadFamNumErr	Illegal family number of $FFFF or $0000
$1B0A	fmBadNameErr	Illegal name length
Memory Manager errors		Returned unchanged

C

```
extern pascal void AddFamily(famNum,namePtr)

Word      famNum;

Pointer      namePtr;
```

$141B AddFontVar

Enables the application to add a variation of a preexisting font family to the Font Manager's collection of available fonts. The *fontHandle* parameter specifies the font to be added. The Font Manager gets the font's family number, style, and size out of the font record itself.

❖ *Note:* The font record is described in Chapter 16, "QuickDraw II," in Volume 2.

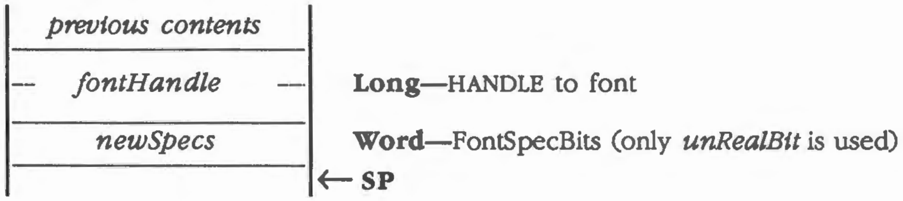
The family number must be the same as that of an existing family; the font will also be assigned the same name. However, the size and style can be different, and at least one of them should be, because the new font should not have a font ID identical to that of an existing font.

The *fontHandle* is left unlocked at the end of this routine. No check is made to see whether the new font ID is unique; that's up to your application. Also, the family number and size of the font are not checked for validity.

AddFontVar enables you to use application-generated versions of fonts (if you don't like the results of the Font Manager's scaling algorithm), fonts styled by the application, and so on.

Parameters

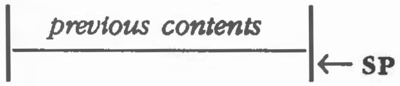
Stack before call



Long—HANDLE to font

Word—FontSpecBits (only *unRealBit* is used)

Stack after call



Errors

$1B04	`fmFamNotFndErr`	Family not found
Memory Manager errors		Returned unchanged

C

```
extern pascal void AddFontVar(fontHandle,newSpecs)
FontHndl     fontHandle;
Word     newSpecs;
```

About the preexisting font family

The preexisting font family can be either given (that is, it is the ROM font, or it in the FONTS folder) or previously supplied by the application with the AddFamily routine. AddFontVar checks to see whether the family number already exists. If it doesn't, then an error is returned, the new font is not added, and the current font is left alone. If the family does exist, then the following actions occur:

1. The new font is placed in the Font Manager's internal list of available fonts. The new font's FontStatBits are set as follows:

   memBit = 1
   purgeBit = 0
   apVarBit = 1
   notDiskBit = 1
   unRealBit = As specified by *newSpecs*
   apFamBit = Inherited from the preexisting font family

2. The font is made the current font and is made unpurgeable.
3. The QuickDraw II text buffer is enlarged (if necessary) to handle the new font.
4. The *fontID, txFace,* and *txSize* fields of the current GrafPort are set to the font ID, style, and size of the new font.

$161B ChooseFont

Displays a dialog box enabling the user to select a new font family, size, and style. When the dialog box is drawn, the family name, style, and size specified by *currentID* will be selected if they exist.

If the *baseOnlyBit* of *famSpecs* is 0, this routine lists the names of all distinct font families currently available to the Font Manager. The available families can change over time if your application adds new families by using the AddFamily routine.

If the *baseOnlyBit* is 1, this routine lists only the names of base font families. All other *famSpecs* bits are ignored.

If *famNum* of *currentID* = $0000, it is translated into the family number of the system font. If *currentID* = $00000000, it is translated into the font ID of the system font.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *currentID* --	**Long**—Font ID of the font currently in use
famSpecs	**Word**—FamSpecBits
	← **SP**

Stack after call

previous contents	
-- *selectedID* --	**Long**—Font ID of the selected font
	← **SP**

Errors

$1B08	`fmBadFamNumErr`	Illegal family number; *famNum* = $FFFF
$1B09	`fmBadSizeErr`	Illegal font size; *size* = $00
Memory Manager errors		Returned unchanged

C

```
extern pascal LongWord ChooseFont(currentID,famSpecs)
FontID     currentID;
Word     famSpecs;
```

❖ *Note:* C Pascal-type functions do not deal properly with data structures returned on the stack. The Long result returned by this call can be passed to any calls requiring a font ID as a parameter. You cannot use the C dot operator to access the individual font ID fields within the value returned by this call.

Tool sets required for ChooseFont

In addition to the tool sets required by the Font Manager, the ChooseFont routine requires the minimum versions of the tool sets shown in Table 8-4 to be loaded and started up.

Table 8-4
ChooseFont routine—other tool sets required

Tool set number		Tool set name	Minimum version needed
$03	#03	Miscellaneous Tool Set	1.2
$0E	#14	Window Manager	1.3
$10	#16	Control Manager	1.3
$14	#20	LineEdit Tool Set	1.0
$15	#21	Dialog Manager	1.0
$1C	#28	List Manager	1.0

Interaction with the user

After the dialog box is drawn, the user can select a new family, style, and size. When a new family is selected, the list of sizes is updated to contain all available sizes for the selected family. The size list can hold a maximum of 12 sizes; however, any valid size can be entered in the Other Size text entry area. Keyboard input other than numbers, Return, Backspace, Left Arrow, Right Arrow, Control-F, Control-X, or Control-Y causes a beep. Clicking the mouse outside the dialog box also causes a beep.

The user can confirm or cancel the new selection. If the user cancels the selection, *selectedID* returns $00000000. If the user attempts to confirm a selection with an illegal size, an alert will be posted, and the size will be highlighted.

❖ *Note:* The size list will display the sizes of all available, real fonts.

When the user has confirmed a valid selection, the selected font is automatically installed. The font ID of the selected font is then returned in *selectedID.*

$091B CountFamilies

Returns the total number of distinct font families currently available to the Font Manager that match a given specification (either *all families* or *all families that have a plain-styled font available*).

If *baseOnlyBit* of the *famSpecs* parameter is 0, CountFamilies returns the number of distinct font families currently available to the Font Manager (in a one-to-one correspondence with distinct family numbers and distinct family names). The count can change over time if application-generated families are added.

If *baseOnlyBit* is 1, the routine returns the number of base font families. This number should not change over the course of an application, since font families added with the AddFamily call are never base families.

All other bits of the *famSpecs* parameter are ignored.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
famSpecs	**Word**—FamSpecBits
	← **SP**

Stack after call

previous contents	
famCount	**Word**—Number of font families available
	← **SP**

Errors

None

C

```
extern pascal Word CountFamilies(famSpecs)
Word     famSpecs;
```

$101B CountFonts

This call returns the number of fonts currently available to the Font Manager that fit a specified description. The *desiredID* parameter supplies a font family number, style, and size. The *fontSpecs* parameter determines the following:

- Which of those parameters must be satisfied (and which can be ignored)
- Whether the fonts to be counted must be in memory
- Whether the fonts to be counted must be real

See the section "FontSpecBits Flag" in this chapter for more information. All effects are cumulative (that is, a font is counted only if it satisfies all applicable specifications).

❖ *Note: CountFounts* does not count fonts that have been purged if the fonts were scaled by the Font Manager or were application-generated fonts.

If *famNum* = $0000, it is translated into the *famNum* of the system font.
If *desiredID* = $00000000, it is translated into the font ID of the system font.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *desiredID* --	**Long**—Font ID
fontSpecs	**Word**—FontSpecBits
	← **SP**

Stack after call

previous contents	
numOfFonts	**Word**—Number of fonts that fit *desiredID* and *fontSpecs*
	← **SP**

Errors

$1B08	`fmBadFamNumErr`	Illegal family number; *famNum* of *desiredID* = $FFFF and *anyFamBit* of *fontSpecs* = 0
$1B09	`fmBadSizeErr`	Illegal font size; size of *desiredID* = $00 and *anySizeBit* of *fontSpecs* = 0

C

```
extern pascal Word CountFonts(desiredID,fontSpecs)
FontID     desiredID;
Word     fontSpecs;
```

Some examples of CountFonts

For example, if *desiredID* consists of a family number of 3, a style of 0, a point size of 12, and a *fontSpecs* parameter of $0002 (*anySizeBit* = 0, *anyStyleBit* = 0, *anyFamBit* = 0, *realOnlyBit* = 1, *memOnlyBit* = 0), the routine returns the number of real Geneva Plain 12 fonts available, whether these are in memory or not. If the result is 0, no such font exists; if the result is 1, such a font exists. (If the result is 2 or more, then two fonts with the same font ID have been loaded.)

If *fontSpecs* is changed to $0018 (*anySizeBit* = 1, *anyStyleBit* = 1, *anyFamBit* = 0, *realOnlyBit* = 0, *memOnlyBit* = 0), the result is the number of Geneva fonts currently available—real or scaled, in memory or on disk, and of any style or size.

If *fontSpecs* is changed to $001C (*anySizeBit* = 1, *anyStyleBit* = 1, *anyFamBit* = 1, *realOnlyBit* = 0, *memOnlyBit* = 0), *desiredID* is completely ignored, and the call returns the total number of fonts available.

$1B1B FamNum2ItemID

Translates a font family number into a menu item ID. This routine can be called after a FixFontMenu call to determine which item ID was assigned to a particular font family. The application can then use this information to put a check mark next to the family name in the menu, for example.

See the section "ItemID2FamNum" in this chapter for information about translating the menu item ID into a family number.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
famNum	**Word**—Family number of menu item whose item ID will be returned
	← **SP**

Stack after call

previous contents	
itemID	**Word**—Item ID assigned to font family with *famNum*
	← **SP**

Errors

$1B0B	fmMenuErr	FixFontMenu never called
$1B04	fmFamNotFndErr	Family number not found

C

```
extern pascal Word FamNum2ItemID(famNum)
Word      famNum;
```

$0A1B FindFamily

Returns the family number and name of a particular font family. The family number is returned on the stack, and the name is returned in the string pointed to by *namePtr*. This routine can be used to step through the list of all available families.

The *positionNum* parameter specifies which font family to find. The Font Manager keeps information about every available font family in a list in memory; *positionNum* specifies the position of the font family within that list. For a *positionNum* of *n*, FindFamily returns the number and name of the *n*th match of the font families in the list.

The position of a particular font family depends upon the organization of the Font Manager's list in memory. The organization of the list does not change over time, but new font families can be added to the end of the list with the AddFamily routine.

If *positionNum* is greater than the number of font families, the call returns a *famNum* of $FFFF (illegal family number) and does not return a name.

The *namePtr* parameter should point to a string large enough to hold a font family name. If *namePtr* is NIL, FindFamily returns only the family number and does not return a name.

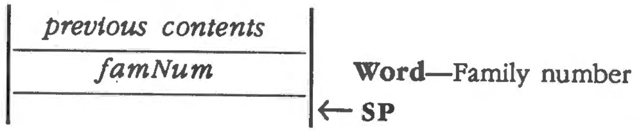
If *baseOnlyBit* of *famSpecs* is 0, FindFamily considers all font families when searching for the family with the correct *positionNum*. If *baseOnlyBit* is 1, the routine considers only base font families when searching for the family with the correct *positionNum*.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
famSpecs	**Word**—FamSpecBits
positionNum	**Word**—Position number of family in list of families matching *famSpecs*
-- *namePtr* --	**Long**—POINTER to space for family name
	← **SP**

Stack after call



Errors None

C

```
extern pascal Word FindFamily(famSpecs,positionNum,namePtr)
Word     famSpecs;
Word     positionNum;
Pointer     namePtr;
```

$111B FindFontStats

Places the font ID and the FontStatBits of a particular font into a specified FontStatRec. The routine can be used to step through the list of available fonts matching the given specifications.

The *desiredID* parameter supplies a font family number, style, and size; *fontSpecs* determines the following:

- Which of those parameters must be satisfied (and which can be ignored)
- Whether the fonts to be counted must be in memory
- Whether the fonts to be counted must be real

❖ *Note:* The font ID returned in the FontStatRec may be different than the *desiredID* if the *fontSpecs* parameter allows the routine to ignore one or more of the fields of *desiredID*.

For *positionNum* = *n*, FindFontStats returns the font ID and the FontStatBits of the *n*th font that satisfies *desiredID* and *fontSpecs*. If *positionNum* is greater than the number of fonts that satisfy *desiredID* and *fontSpecs*, the call sets *notFoundBit* in the FontStatRec to 1, with the other bits of the FontStatBits in the FonStatRec and the entire font ID remaining undefined.

If *famNum* of *desiredID* = $0000, it is translated into the family number of the system font. If *desiredID* = $00000000, it is translated into the font ID of the system font.

Parameters

Stack before call

previous contents	
-- *desiredID* --	**Long**—Font ID
fontSpecs	**Word**—FontSpecBits
positionNum	**Word**—Position number of font
-- *resultPtr* --	**Long**—POINTER to FontStatRec
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$1B08	fmBadFamNumErr	Illegal family number; *famNum* = $FFFF and *anyFamBit* = 0
$1B09	fmBadSizeErr	Illegal font size; *size* = $00 and *anySizeBit* = 0

C

```
extern pascal void FindFontStats(desiredID,fontSpecs,positionNum,resultPtr)

FontID     desiredID;

Word     fontSpecs;

Word     positionNum;

FontStatRecPtr     resultPtr;
```

FindFontStats example

To get the FontStatBits of a particular font, you could take the following steps:

1. Use the font ID of the font as *desiredID* in the call.
2. Set *fontSpecs* in the call to $0000 (which means find a font with the family number, style, and size as specified in *desiredID,* whether the font is in memory or not, and whether it is scaled or real).
3. Set *positionNum* in the call to 1 (you want to find the first such font, and there shouldn't be more than one for this example, since the family number, style, and size are completely specified).

In assembly language, the instructions would look like this, if the desired font has a family number of 6, a size of 12, and a style of 0:

```
PushLong #$0C000006          ; desiredID

PushWord #$0000              ; fontSpecs

PushWord #$0001              ; positionNum

PushPtr  FontStatRec

_FindFontStats
```

If the font exists, the *resultStats* field of the returned FontStatRec contains the FontStatBits for the font (the *resultID* field of the FontStatRec is redundant in this case, because it must be equal to *desiredID*).

If the font doesn't exist, the *notFoundBit* of the *resultStats* field of the FontStatRec provides the information (all other bits of that field and the other field of FontStatRec are undefined).

$151B FixFontMenu

Appends the names of available font families to a specified menu. The names are appended in alphabetical order, with the first family name assigned a menu item ID of *startingID,* the next family name assigned a menu item ID of *startingID* + 1, and so on. The Font Manager routine ItemID2FamNum can be used to translate a menu item ID into the family number assigned to it. Conversely, the Font Manager routine FamNum2ItemID can be used to translate a family number into a menu item ID.

After the application calls FixFontMenu, it can call Menu Manager routines to set the menu width.

If the *baseOnlyBit* of *famSpecs* is 0, this routine appends the names of all distinct font families currently available to the Font Manager. The available families can change over time if your application adds new families.

If the *baseOnlyBit* is 1, this routine appends only the names of base font families. This number shouldn't change over the course of an application. All other *famSpecs* bits are ignored.

FixFontMenu also requires tool sets in addition to the ones required by the rest of the Font Manager calls. See Table 8-5 for that information.

Parameters

Stack before call

previous contents	
menuID	**Word**—Menu ID of menu to which family names will be appended
startingID	**Word**—Item ID to assign to first family name appended to menu
famSpecs	**Word**—FamSpecBits
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors	Returned unchanged

C

```
extern pascal void FixFontMenu(menuID,startingID,famSpecs)
Word     menuID;
Word     startingID;
Word     famSpecs;
```

Tool sets required for FixFontMenu

In addition to the tool sets required by the Font Manager, the FixFontMenu routine requires the minimum versions of the tool sets shown in Table 8-5 to be loaded and started up.

Table 8-5
FixFontMenu routine—other tool sets required

Tool set number		Tool set name	Minimum version needed
$0F	#15	Menu Manager	1.3
$1C	#28	List Manager	1.0

$1A1B FMGetCurFID

Returns the font ID of the current font.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
	← **SP**

Stack after call

previous contents	
-- *curID* --	**Long**—Font ID of the current font
	← **SP**

Errors

None

C

```
extern pascal LongWord FMGetCurFID()
```

❖ *Note:* C Pascal-type functions do not deal properly with data structures returned on the stack. The Long result returned by this call can be passed to any calls requiring a font ID as a parameter. You cannot use the C dot operator to access the individual font ID fields within the value returned by this call.

$191B FMGetSysFID

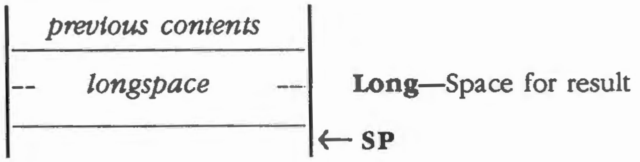
Returns the font ID of the system font.

Important

This is the system font as set by the FMStartUp or FMSetSysFont calls. If the QuickDraw II SetSysFont call has been used to set the system font, the Font Manager will not have that information.

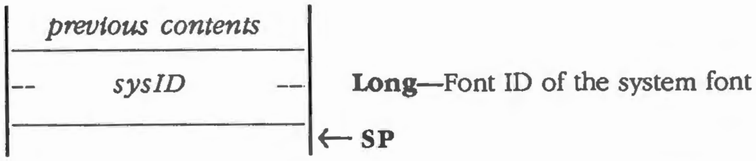
Parameters

Stack before call



Long—Space for result

Stack after call



Long—Font ID of the system font

Errors

None

C

```
extern pascal LongWord FMGetSysFID()
```

❖ *Note:* C Pascal-type functions do not deal properly with data structures returned on the stack. The Long result returned by this call can be passed to any calls requiring a font ID as a parameter. You cannot use the C dot operator to access the individual font ID fields within the value returned by this call.

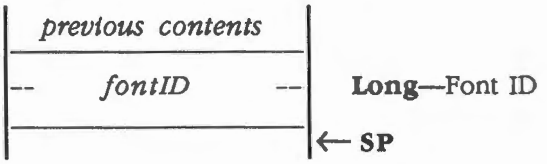
$181B FMSetSysFont

Loads a specified font into memory (if it's not already there), makes it unpurgeable, and makes it the system font.

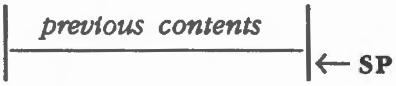
If *famNum* of *fontID* = $0000, it is translated into the family number of the system font. If *fontID* = $00000000, it is translated into the font ID of the system font.

Parameters

Stack before call



Stack after call



Errors

$1B05	fmFontNtFndErr	Font not found
$1B08	fmBadFamNumErr	Illegal family number
$1B09	fmBadSizeErr	Illegal font size
Memory Manager errors		Returned unchanged
ProDOS errors		Returned unchanged

C

```
extern pascal void FMSetSysFont(fontID)

FontID      fontID;
```

$0B1B GetFamInfo

Returns the name of the font family with a specified family number, placing the name wherever *namePtr* is pointing. It also returns *famStats,* with *apFamBit, notBaseBit,* and *notFoundBit* set to the correct values (see the section "FamStatBits Flag" in this chapter for definitions). The other bits of *famStats* are undefined.

GetFamInfo also tells you whether the family exists or not: If there is no family with the given *famNum, famStats* is returned with the *notFoundBit* equal to 1 (and the others undefined). In this case, nothing is altered in the buffer pointed to by *namePtr.*

If *namePtr* is 0, no name is returned; if you don't need the name, you can use this characteristic to get *famStats* without setting aside space for the name.

If *famNum* = $0000, it is translated into the family number of the system font.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
famNum	**Word**—Family number of font
— *namePtr* —	**Long**—POINTER to space (26 bytes) for font family name
	← **SP**

Stack after call

previous contents	
famStats	**Word**—FamStatBits
	← **SP**

Errors

$1B08 `fmBadFamNumErr` Illegal family number

C

```
extern pascal Word GetFamInfo(famNum,namePtr)

Word      famNum;

Pointer     namePtr;
```

$0C1B GetFamNum

Returns the family number corresponding to a specified font family name. The family name is pointed to by *namePtr.* The name must match exactly, including length, spaces, and uppercase and lowercase distinctions. If the name has a length of 0, an error is returned; if the name is more than 25 characters long, only the first 25 characters are used.

If there is no such family, $FFFF (the illegal family number) is returned in *famNum* (this also tells you whether a family name is currently in use).

❖ *Note:* All fonts with the same family number should have the same font family name and vice versa. If the font files in the FONTS subdirectory (or those added later by the application) do not observe this rule, some fonts can get lost; that is, some of the Font Manager calls won't know about them.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *namePtr* --	**Long**—POINTER to family name
	← **SP**

Stack after call

previous contents	
famNum	**Word**—Family number
	← **SP**

Errors

$1B0A	`fmBadNameErr`	Illegal name length

C

```
extern pascal Word GetFamNum(namePtr)

Pointer     namePtr;
```

$0E1B InstallFont

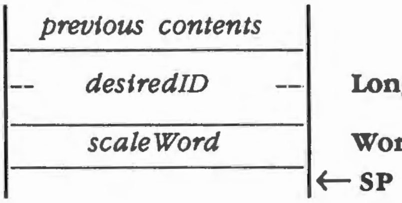
Performs the following actions:

- Finds a specified font or the available font with the best fit if the specified font isn't available
- Loads the font into memory, if necessary
- If the best-fit font has been used and scaling has not been disabled, creates a new, scaled font to match the specified font's size
- Makes the resulting font current and unpurgeable

If *famNum* of *desiredID* = $0000, it is translated into the family number of the system font. If *desiredID* = $00000000, it is translated into the font ID of the system font.

Parameters

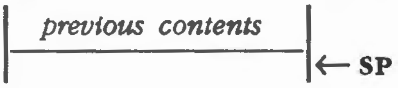
Stack before call



Long—Font ID of desired font

Word—Last bit of 0 enables scaling, 1 disables scaling (see Figure 8-10)

Stack after call



Errors

$1B08	fmBadFamNumErr	Illegal family number
$1B09	fmBadSizeErr	Illegal size
$1B0C	fmScaleSizeErr	Scaled size of font exeeds limits
Memory Manager errors		Returned unchanged (system font installed)
ProDOS errors		Returned unchanged (system font installed)

C

```
extern pascal void InstallFont(desiredID,scaleWord)

FontID     desiredID;

Word     scaleWord;
```

(continued)

InstallFont's actions

To find the font you requested in the *desiredID* parameter, the InstallFont routine does the following:

1. The routine searches all known fonts for the font that best fits the font identified by *desiredID*. The algorithm for determining best fit is described in the section "Best-Fit Font Algorithm" in this chapter. If the best-fit font isn't already in memory, InstallFont brings it in from disk.
2. The last bit of *scaleWord* controls scaling, as shown in Figure 8-10. If the best-fit font is the right size, or if it is the wrong size but scaling is disabled, then it is made the current font and made unpurgeable.

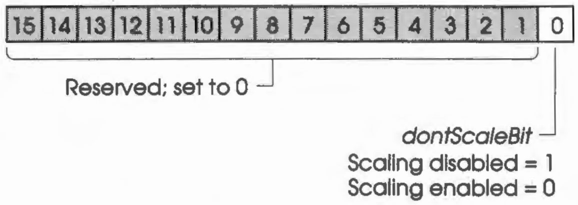


Figure 8-10
Font scale word

❖ *Note:* Scaling was implemented in Version 1.1 of the Font Manager. Only that version or later versions can use *dontScaleBit*. Earlier Font Manager versions always act as if *dontScaleBit* = 1.

3. If the best-fit font is the wrong size and scaling is enabled, then the Font Manager creates a new font by scaling the best-fit font and makes this new font current and unpurgeable. In this case, the best-fit font is either made purgeable (if it has just been brought in from disk by this call), or its purge status is left alone (if it was already in memory when this call was made).

   The FontStatBits for the font created by scaling have *memBit*, *notDiskBit*, and *unrealBit* set to 1 and inherit *apFamBit* and *apVarBit* from the best-fit font.
4. Installfont next enlarges the QuickDraw II text buffer, if necessary, to handle the font.
5. The routine then sets the font ID field of the current GrafPort equal to *desiredID*. Even if the precise font cannot be found at this time, future pictures and laser printers will know what font was requested.
6. Finally, Installfont sets the *txFace* and *txSize* fields of the current GrafPort to the style and size specified in *desiredID*.

$171B ItemID2FamNum

Translates a menu item ID into a font family number.

Use this routine after you have used the FixFontMenu routine to create a menu of family names. Because the Font Manager appended the names to the menu, the application has no way of knowing which menu item IDs correspond to which families. The ItemID2FamNum routine performs the translation.

❖ *Note:* See the section "FamNum2ItemID" in this chapter for information about translating a family number into a menu item ID.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
itemID	**Word**—Item ID of menu item whose family number will be returned
	← **SP**

Stack after call

previous contents	
famNum	**Word**—Family number of family corresponding to *itemID*
	← **SP**

Errors

$1B0B	`fmMenuErr`	FixFontMenu never called
$1B04	`fmFamNotFndErr`	Family not found; item ID not in menu

C

```
extern pascal Word ItemID2FamNum(itemID)

Word     itemID;
```

$121B LoadFont

Finds a particular font with a specified font ID and specifications (such as the FindFontStats routine), loads the font into memory (if it is not already there), and makes the font current and unpurgeable. The routine then enlarges the QuickDraw II text buffer (if necessary) to handle the font. Finally, the routine sets the *fontID, txFace,* and *txSize* fields of the current GrafPort to the font ID, style, and size of the font that was loaded. If no such font is found, LoadFont does not change the current font.

LoadFont was included for those occasions when you want your application to step through and use all the available fonts matching the given specifications.

If *famNum* of *desiredID* = $0000, it is translated into the family number of the system font. If *desiredID* = $00000000, it is translated into the font ID of the system font.

Parameters

Stack before call

previous contents	
-- *desiredID* --	**Long**— Font ID
fontSpecs	**Word**—FontSpecBits
positionNum	**Word**—Position number of font
-- *resultPtr* --	**Long**—POINTER to FontStatRec
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$1B08	`fmBadFamNumErr`	Illegal family number; *famNum* is $FFFF and *anyFamBit* is 0
$1B09	`fmBadSizeErr`	Illegal font size; *size* is $00 and *anySizeBit* is 0
Memory Manager errors		Returned unchanged
ProDOS errors		Returned unchanged

C

```
extern pascal void LoadFont(desiredID,fontSpecs,positionNum,resultPtr)
FontID     desiredID;
Word     fontSpecs;
Word     positionNum;
FontStatRecPtr     resultPtr;
```

$131B LoadSysFont

Makes the system font current without forcing the application to know its font ID. The routine then enlarges the QuickDraw II text buffer (if necessary) to handle the system font. Finally, the routine sets the *fontID, txFace,* and *txSize* fields of the current GrafPort to the font ID, style, and size of the system font.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

None

C

```
extern pascal void LoadSysFont()
```

$0F1B SetPurgeStat

Makes a specified font in memory unpurgeable or purgeable. If the routine finds the specified font, it makes it either unpurgeable (if *purgeBit* of *purgeStat* is 0) or purgeable (if *purgeBit* of *purgeStat* is 1). The other bits of *purgeStat* are not used.

Important

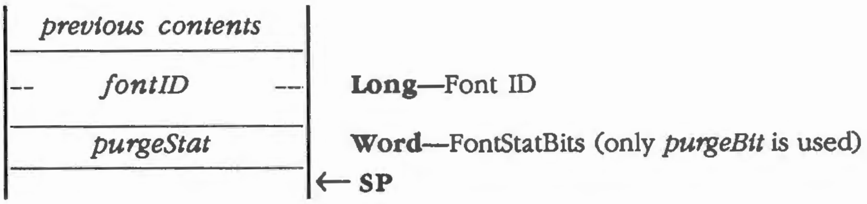
Don't make a font purgeable unless you intend to reload the font if and when your application needs the font again. For example, if a font was purged and a GrafPort that contains that font's handle was made current, any QuickDraw II text-drawing calls would cause the system to fail.

If the font isn't found, the call returns an error, and no purge status is changed.

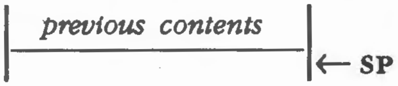
If *famNum* of *fontID* = $0000, it is translated into the family number of the system font. If *fontID* = $00000000, it is translated into the font ID of the system font.

Parameters

Stack before call



Stack after call



Errors

$1B05	`fmFontNtFndErr`	Font not found
$1B06	`fmFontMemErr`	Font not in memory
$1B07	`fmSysFontErr`	System font cannot be purgeable
$1B08	`fmBadFamNumErr`	Illegal family number
$1B09	`fmBadSizeErr`	Illegal font size

C

```
extern pascal void SetPurgeStat(fontID,purgeStat)

FontID     fontID;

Word     purgeStat;
```

Font Manager summary

This section briefly summarizes the constants, data structures, and tool set errors contained in the Font Manager.

Important

These definitions are provided in the appropriate interface file.

Table 8-6
Font Manager constants

Name	Value	Meaning
FontStatBits		
`memBit`	$0001	Font is in memory
`unrealBit`	$0002	Font is scaled from another font
`apFamBit`	$0004	Font family supplied by application
`apVarBit`	$0008	Font added by AddFontVar call or scaled from such a font
`purgeBit`	$0010	Font is purgeable
`notDiskBit`	$0020	Font not ROM font and not in FONTS subdirectory
`notFoundBit`	$8000	Specified font not found
FontSpecBits		
`memOnlyBit`	$0001	Allow only ROM font and fonts currently in memory
`realOnlyBit`	$0002	Allow only real (unscaled) fonts
`anyFamBit`	$0004	Ignore family number supplied in call
`anyStyleBit`	$0008	Allow any font whose style partially matches style supplied in call
`anySizeBit`	$0010	Ignore point size supplied by call
FamStatBits		
`apFamBit`	$0004	Font family supplied by application
`notBaseBit`	$0020	Family is not a base family
`notFoundBit`	$8000	Specified font family not found
FamSpecBits		
`baseOnlyBit`	$0020	Allow only base families
Scale word		
`dontScaleBit`	$0001	Disable font scaling

Table 8-6 (continued)
Font Manager constants

Name	Value	Meaning
Family numbers		
`newYork`	$0002	New York font
`geneva`	$0003	Geneva font
`monaco`	$0004	Monaco font
`venice`	$0005	Venice font
`london`	$0006	London font
`athens`	$0007	Athens font
`sanFran`	$0008	San Francisco font
`toronto`	$0009	Toronto font
`cairo`	$000B	Cairo font
`losAngeles`	$000C	Los Angeles font
`times`	$0014	Times font
`helvetica`	$0015	Helvetica font
`courier`	$0016	Courier font
`symbol`	$0017	Symbol font
`taliesin`	$0018	Taliesin font
`shaston`	$FFFE	Shaston font

Table 8-7
Font Manager data structures

Name	Offset	Type	Definition
FontID			
famNum	$00	Word	Family number of font
fontStyle	$02	Byte	Style of font (bold, italicized, and so on)
fontSize	$03	Byte	Size of font in points
FontStatRec			
resultID	$00	FontID	Includes family number, font style, and font size
resultStats	$04	Word	Equal to FontStatBits

Note: The actual assembly-language equates have a lowercase *o* (the letter) in front of all of the names given in this table.

Table 8-8
Font Manager error codes

Code	Name	Description
$1B01	`fmDupStartUpErr`	FMStartUp call already made
$1B02	`fmResetErr`	Can't reset the Font Manager
$1B03	`fmNotActiveErr`	Font Manager not active
$1B04	`fmFamNotFndErr`	Family not found
$1B05	`fmFontNtFndErr`	Font not found
$1B06	`fmFontMemErr`	Font not in memory
$1B07	`fmSysFontErr`	System font cannot be purgeable
$1B08	`fmBadFamNumErr`	Illegal family number
$1B09	`fmBadSizeErr`	Illegal font size
$1B0A	`fmBadNameErr`	Illegal name length
$1B0B	`fmMenuErr`	FixFontMenu never called
$1B0C	`fmScaleSizeErr`	Scaled size of font exceeds limits

Chapter 9

Integer Math Tool Set

The **Integer Math Tool Set** supports multiplication and division of several types of numbers and also converts numbers from one type to another. The types of entities dealt with are as follows:

- **Integer Math strings:** ASCII strings with no length indication supplied by the string itself
- **Integers:** 16-bit signed or unsigned values
- **Longints:** 32-bit signed or unsigned values
- **Fixed:** 32-bit signed values with 16 bits of fraction
- **Frac:** 32-bit signed values with 30 bits of fraction
- **Extended:** 80-bit signed floating-point values with 64 bits of fraction

❖ *Note:* The Extended type really serves as a pathway to the Standard Apple Numeric Environment (SANE). For more information, refer to the *Apple Numerics Manual.*

A preview of the Integer Math Tool Set routines

To introduce you to the capabilities of the Integer Math Tool Set, all Integer Math routines are grouped by function and briefly described in Table 9-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the Integer Math routines (discussed in alphabetical order).

Table 9-1
Integer Math Tool Set routines and their functions

Routine	Description
Housekeeping routines	
IMBootInit	Initializes the tool set; called only by the Tool Locator—must not be called by an application
IMStartUp	Starts up the Integer Math Tool Set for use by an application
IMShutDown	Shuts down the Integer Math Tool Set when an application quits
IMVersion	Returns the version number of the Integer Math Tool Set
IMReset	Resets the tool set; called only when the system is reset—must not be called by an application
IMStatus	Indicates whether the Integer Math Tool Set is active
Math routines	
Multiply	Multiplies two Integer inputs and produces a Longint result
SDivide	Divides two Integers and produces a signed Integer quotient and a signed Integer remainder
UDivide	Divides two unsigned Integer inputs and produces an Integer quotient and an unsigned Integer remainder
LongMul	Multiplies two Longint values and produces a 64-bit result
LongDivide	Divides two unsigned Longint inputs and produces a Longint unsigned quotient and a Longint unsigned remainder
FixRatio	Takes two signed Integers and produces a Fixed number as a ratio of the numerator and denominator
FixMul	Multiplies two 32-bit Fixed inputs and produces a 32-bit Fixed result
FracMul	Multiplies two Frac inputs and returns a rounded Frac result
FixDiv	Divides two like inputs and returns a rounded Fixed result (no remainder)
FracDiv	Divides two like inputs and returns a rounded Frac result (no remainder)
FixRound	Takes a Fixed input and returns a rounded Integer result
FracSqrt	Takes a Frac input and returns a rounded Frac square root
FracCos	Takes a Fixed input (in radians) and returns its Frac cosine
FracSin	Takes a Fixed input (in radians) and returns its Frac sine
FixATan2	Takes two like inputs and returns a Fixed arc tangent (in radians) of their coordinates
HiWord	Returns high-order word of a Long input
LoWord	Returns low-order word of a Long input
Long2Fix	Converts a specified Longint value to its corresponding Fixed value
Fix2Long	Converts a Fixed value to its corresponding Longint value
Fix2Frac	Converts a Fixed value to its corresponding Frac value
Frac2Fix	Converts a specified Frac value to its corresponding Fixed value
Fix2X	Converts a Fixed value to its corresponding Extended value
Frac2X	Converts a specified Frac value to its corresponding Extended value
X2Fix	Converts an Extended value to its corresponding Fixed value
X2Frac	Converts an Extended value to its corresponding Frac value

Table 9-1 (continued)
Integer Math Tool Set routines and their functions

Routine	Description
Integer Math string routines	
Int2Hex	Takes an unsigned Integer and produces an Integer Math string representing the value in hexadecimal format
Long2Hex	Takes an unsigned Longint value and produces an Integer Math string representing the value in hexadecimal format
Hex2Int	Takes an Integer Math string representing a hexadecimal value and returns an unsigned Integer
Hex2Long	Takes an Integer Math string representing a hexadecimal value and returns an unsigned Longint
Int2Dec	Takes a signed or unsigned Integer and produces an Integer Math string representing the value in decimal format
Long2Dec	Takes a signed or unsigned Longint value and produces an Integer Math string representing the value in decimal format
Dec2Int	Takes an Integer Math string representing a decimal value and returns a signed or unsigned Integer
Dec2Long	Takes an Integer Math string representing a decimal value and produces a Longint value
HexIt	Takes an unsigned Integer and returns a 4-byte Integer Math string representing the value in hexadecimal format

Rounding and pinning

When the result of a computation cannot be represented exactly in a destination's format due to insufficient precision, the Integer Math Tool Set will usually round the result. A **rounded result** is the nearest representable value to the actual value, with ties going to the value with the larger magnitude. Those Integer Math Tool Set routines that use rounding are identified as such in their descriptions. Some Integer Math routines do not round; in those cases, the delivered result will be the nearest representable value that is less than the actual value.

The Integer Math Tool Set uses another concept to handle overflows that occur when the magnitude of the result exceeds the destination format. In these cases, the tool set usually supports **pinning,** which assigns positive overflows to the largest positive representable value and negative overflows to the largest negative representable value.

Using the Integer Math Tool Set

The Integer Math Tool Set depends upon the presence of the tool sets shown in Table 9-2 and requires that at least the indicated version of the tool set be present.

Table 9-2
Integer Math Tool Set—other tool sets required

Tool set number		Tool set name	Minimum version needed
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0

Your application should make an IMStartUp call before making any other Integer Math Tool Set calls.

◆ *Note:* At the time of publication, the IMStartUp call was not an absolute requirement, because the Tool Locator automatically started up the Integer Math Tool Set at boot time. However, you should make the call anyway to guarantee that your application remains compatible with all future versions of the system.

If you have started up the tool set, your application should also make the IMShutDown call when the application quits.

Within the tool set, there are Math routines and Integer Math string routines. Math routines support multiplication and division of Integer, Longint, Fixed, and Frac numbers and convert from one type of value to another.

Integer Math string routines convert between a binary value and an ASCII character string representing that value. The binary value can be either an Integer or a Longint. The character string can be in either hexadecimal or decimal format.

$010B IMBootInit

Initializes the Integer Math Tool Set; called only by the Tool Locator.

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 Call must not be made by an application.

$020B IMStartUp

Starts up the Integer Math Tool Set for use by an application. Your application should make an IMStartUp call before making any other Integer Math Tool Set calls.

❖ *Note:* At the time of publication, the IMStartUp call was not an absolute requirement, because the Tool Locator automatically started up the Integer Math Tool Set at boot time. However, you should make the call anyway to guarantee that your application remains compatible with all future versions of the system.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void IMStartUp()
```

$030B IMShutDown

Shuts down the Integer Math Tool Set when an application quits.

Important

If your application has started up the Integer Math Tool Set, the application must make this call before it quits.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

None

C

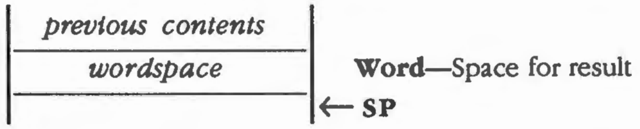
```
extern pascal void IMShutDown()
```

$040B IMVersion

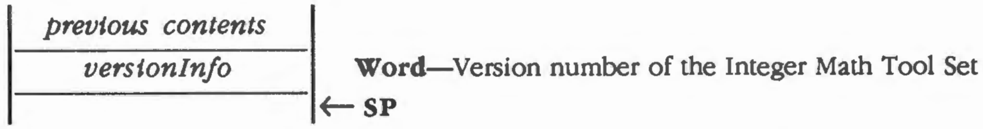
Returns the version number of the Integer Math Tool Set.

Parameters

Stack before call



Stack after call



Errors

None

C

```
extern pascal Word IMVersion()
```

$050B IMReset

Resets the Integer Math Tool Set; called 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

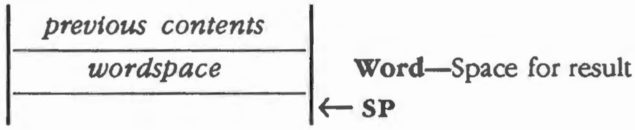
C Call must not be made by an application.

$060B IMStatus

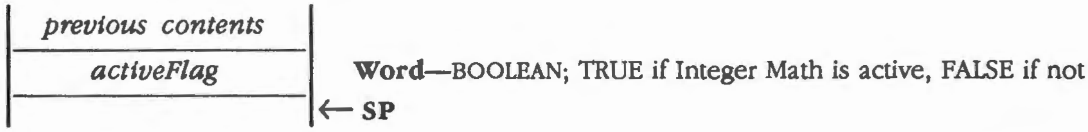
Indicates whether the Integer Math Tool Set is active.

Parameters

Stack before call



Stack after call



previous contents
activeFlag
Word—BOOLEAN; TRUE if Integer Math is active, FALSE if not
← **SP**

Errors None

C

```
extern pascal Boolean IMStatus()
```

$280B Dec2Int

Takes an Integer Math string representing a decimal value and returns a signed or unsigned Integer. The string must consist of digits and blanks. If the string does not fill up the space, pad the string at the left with blanks or zeros. The ASCII characters in the string may have the high-order bit either set or clear.

If the *signedFlag* is a nonzero value, the string may contain an ASCII plus or minus sign directly in front of the most significant digit.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
— *strPtr* —	**Long**—POINTER to Integer Math string
strLength	**Word**—Length of Integer Math string
signedFlag	**Word**—0 if *intResult* is unsigned, nonzero if *intResult* is signed
	← **SP**

Stack after call

previous contents	
intResult	**Word**—INTEGER equivalent of the Integer Math string
	← **SP**

Errors

$0B02	`imIllegalChar`	Illegal character in string
$0B03	`imOverflow`	Signed value is greater than 32,767 or less than –32,768, or unsigned value is greater than 65,535

C

```
extern pascal Integer Dec2Int(strPtr,strLength,signedFlag)

Pointer     strPtr;

Word     strLength;

Boolean     signedFlag;
```

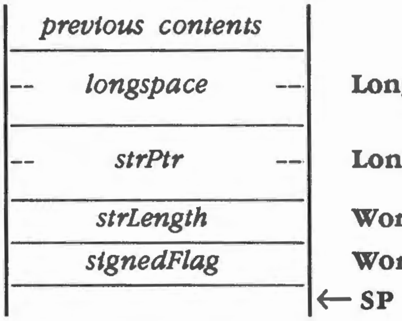
$290B Dec2Long

Takes an Integer Math string representing a decimal value and produces a Longint value. The string should be right-justified and may be padded at the left with blanks or zeros. The ASCII characters in the string may have the high-order bit either set or clear.

If the value is signed, the string may contain an ASCII plus or minus sign directly in front of the most significant digit.

Parameters

Stack before call



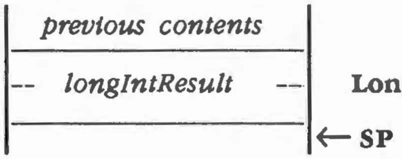
Long—Space for the result

Long—POINTER to Integer Math string

Word—Length of Integer Math string

Word—0 if *longIntResult* is unsigned, nonzero if *longIntResult* is signed

Stack after call



Long—LONGINT equivalent of the Integer Math string

Errors

$0B02	`imIllegalChar`	Illegal character in string
$0B03	`imOverflow`	Signed value is greater than 2,147,483,647 or less than –2,147,483,648, or unsigned value is greater than 4,294,967,295

C

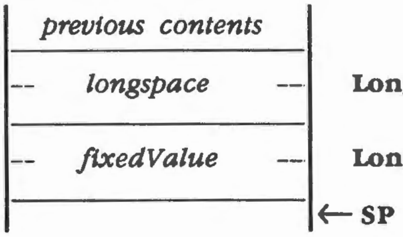
```
extern pascal Longint Dec2Long(strPtr,strLength,signedFlag)

Pointer     strPtr;

Word      strLength;

Boolean     signedFlag;
```

$1C0B Fix2Frac

Converts a Fixed value to its corresponding Frac value. Out-of-range values are pinned to the most positive or negative value, depending on the sign of the input.

Parameters

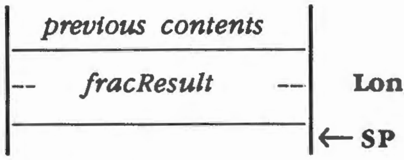
Stack before call



Long—Space for result

Long—FIXED value to be converted

Stack after call



Long—FRAC result of conversion; pinned if out-of-range

Errors

None

C

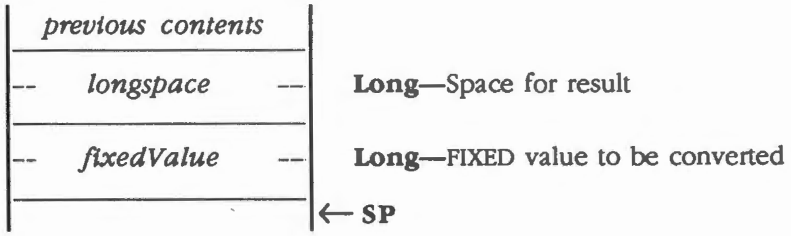
```
extern pascal Frac Fix2Frac(fixedValue)

Fixed     fixedValue;
```

$1B0B Fix2Long

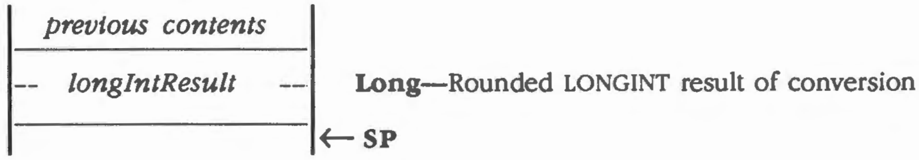
Converts a Fixed value to its corresponding Longint value. Conversions are rounded.

Parameters

Stack before call



Stack after call



Errors

None

C

```
extern pascal Longint Fix2Long(fixedValue)
Fixed     fixedValue;
```

$1E0B Fix2X

Converts a Fixed value to its corresponding Extended value.

Parameters

Stack before call

```
|   previous contents   |
|-----------------------|
|--   fixedValue     --|   Long—FIXED input
|-----------------------|
|--   extendPtr      --|   Long—POINTER to space for EXTENDED value
|-----------------------|
|                       |← SP
```

Stack after call

```
|   previous contents   |
|-----------------------|
|                       |← SP
```

Errors

None

C

```
extern pascal void Fix2X(fixedValue,extendPtr)

Fixed      fixedValue;

ExtendPtr     extendPtr;
```

$170B FixATan2

Takes two like inputs and returns a Fixed arc tangent (in radians) of their coordinates. The inputs can be Frac, Fixed, or signed Longint, but both must be of the same type.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *input1* --	**Long**—First input
-- *input2* --	**Long**—Second input (must be same type as first)
	← **SP**

Stack after call

previous contents	
-- *arcTanResult* --	**Long**—FIXED arc tangent, in radians, of *input1* and *input2*
	← **SP**

Errors

None

C

```
extern pascal Fixed FixATan2(input1,input2)
Longint      input1;
Longint      input2;
```

$110B FixDiv

Divides two like inputs and returns a rounded Fixed result (no remainder). Overflows are pinned to the most positive or negative value, depending on the XOR of the signs of the inputs. The inputs can be Frac, Fixed, or signed Longint, but both must be of the same type.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *dividend* --	**Long**—First input
-- *divisor* --	**Long**—Second input (must be same type as *dividend*)
	← **SP**

Stack after call

previous contents	
-- *fixedResult* --	**Long**—Rounded FIXED result; pinned if out-of-range
	← **SP**

Errors

None

C

```
extern pascal Fixed FixDiv(dividend,divisor)

Longint     dividend;

Longint     divisor;
```

$0F0B FixMul

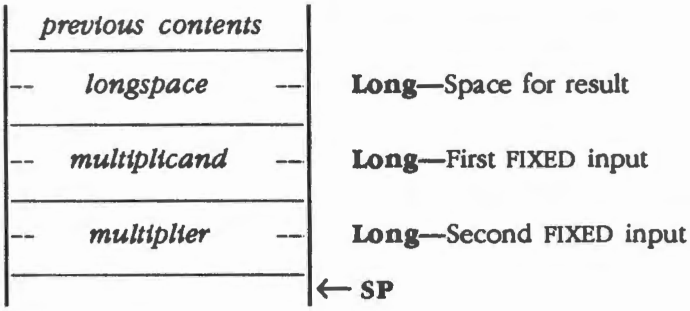
Multiplies two Fixed inputs and produces a Fixed result.

Important

The result is the same as if two 32-bit integers were multiplied, producing a 64-bit product, and only the middle 32 bits were returned.

Parameters

Stack before call



previous contents

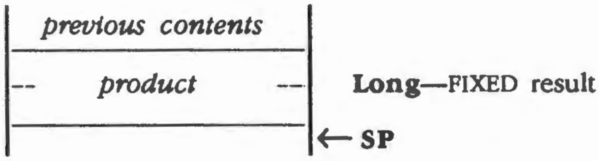
-- *longspace* -- **Long**—Space for result

-- *multiplicand* -- **Long**—First FIXED input

-- *multiplier* -- **Long**—Second FIXED input

← **SP**

Stack after call



previous contents

-- *product* -- **Long**—FIXED result

← **SP**

Errors

None

C

```
extern pascal Fixed FixMul(multiplicand,multiplier)

Fixed     multiplicand;

Fixed     multiplier;
```

$0E0B FixRatio

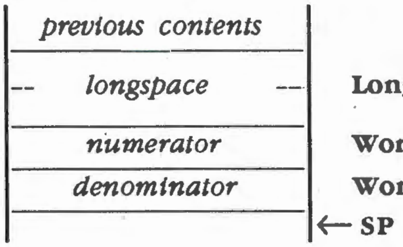
Takes two signed Integers and produces a Fixed number as a ratio of the numerator and denominator.

Important

FixRatio doesn't check for the divide-by-zero condition, nor does it cause an error to occur when that condition happens. Therefore, your application must prevent that condition from occurring.

Parameters

Stack before call

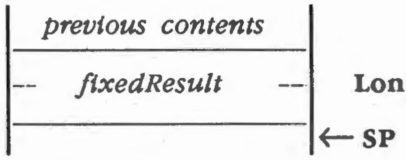


Long—Space for result

Word—INTEGER specifying the input numerator

Word—INTEGER specifying the input denominator

Stack after call



Long—FIXED result

Errors

None

C

```
extern pascal Fixed FixRatio(numerator,denominator)

Integer     numerator;

Integer     denominator;
```

$130B FixRound

Takes a Fixed input and returns a rounded Integer result.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *fixedValue* --	**Long**—Original FIXED value
	← **SP**

Stack after call

previous contents	
intResult	**Word**—Rounded INTEGER result
	← **SP**

Errors

None

C

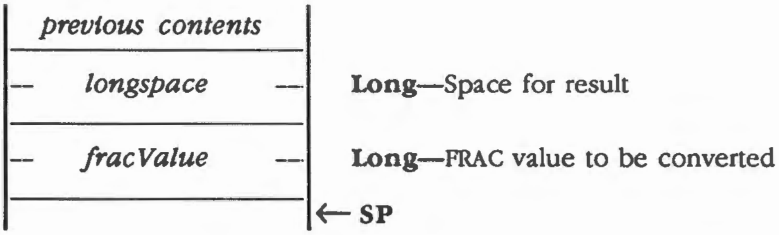
```
extern pascal Integer FixRound(fixedValue)

Fixed     fixedValue;
```

$1D0B Frac2Fix

Converts a specified Frac value to its corresponding Fixed value. Conversions are rounded.

Parameters

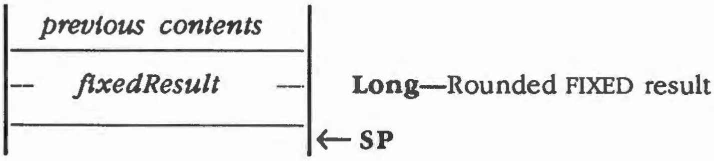
Stack before call



Long—Space for result

Long—FRAC value to be converted

Stack after call



Long—Rounded FIXED result

Errors

None

C

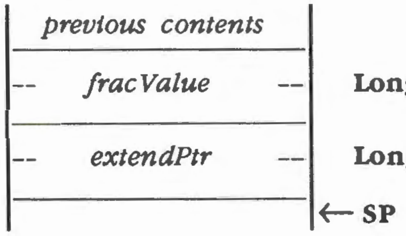
```
extern pascal Fixed Frac2Fix(fracValue)

Frac     fracValue;
```

$1F0B Frac2X

Converts a specified Frac value to its corresponding Extended value.

Parameters

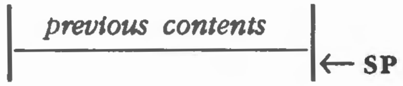
Stack before call



Long—FRAC value to be converted

Long—POINTER to space for EXTENDED value

Stack after call



Errors

None

C

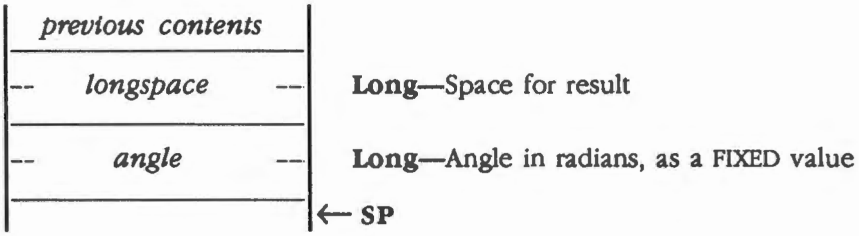
```
extern pascal void Frac2X(fracValue,extendPtr)

Frac      fracValue;

ExtendPtr     extendPtr;
```

$150B FracCos

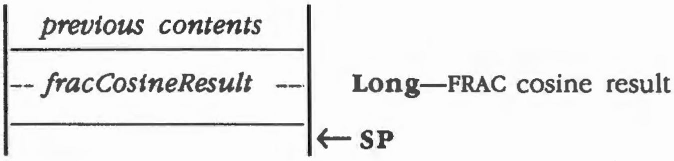
Takes a Fixed input (radians) and returns its Frac cosine.

Parameters

Stack before call



Stack after call



Errors

None

C

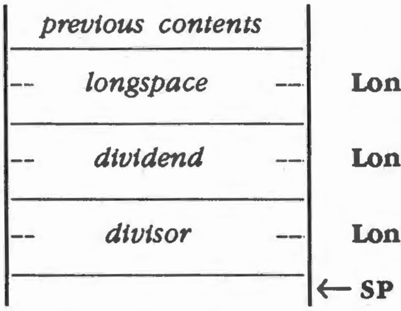
```
extern pascal Frac FracCos(angle)
Fixed     angle;
```

$120B FracDiv

Divides two like inputs and returns a rounded Frac result (no remainder). Overflows are pinned to the most positive or negative value, depending on the XOR of the signs of the inputs. The inputs can be Frac, Fixed, or signed Longint, but both must be of the same type.

Parameters

Stack before call

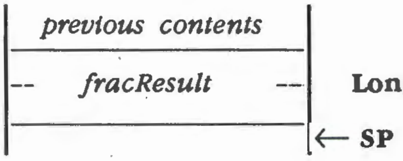


Long—Space for result

Long—First input

Long—Second input (must be the same type as *dividend*)

Stack after call



Long—Rounded FRAC result; pinned if out-of-range

Errors

None

C

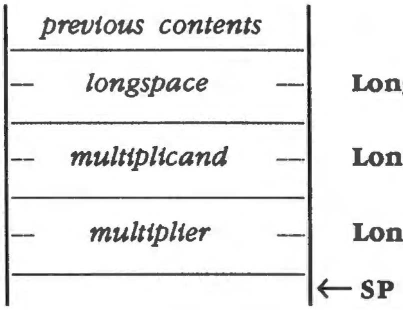
```
extern pascal Frac FracDiv(dividend,divisor)
Longint     dividend;
Longint     divisor;
```

$100B FracMul

Multiplies two Frac inputs and returns a rounded Frac result. Overflows are pinned to the most positive or negative value, depending on the XOR of the signs of the input.

Parameters

Stack before call

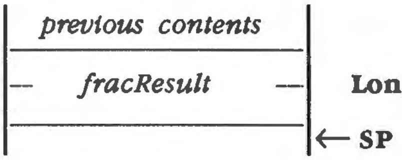


Long—Space for result

Long—First FRAC input

Long—Second FRAC input

Stack after call



Long—Rounded FRAC result; pinned if out-of-range

Errors

None

C

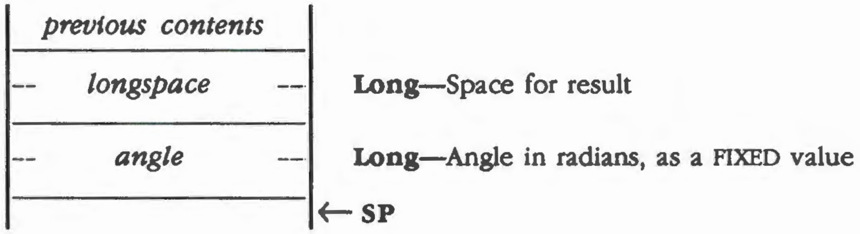
```
extern pascal Frac FracMul(multiplicand,multiplier)
Frac     multiplicand;
Frac     multiplier;
```

$160B FracSin

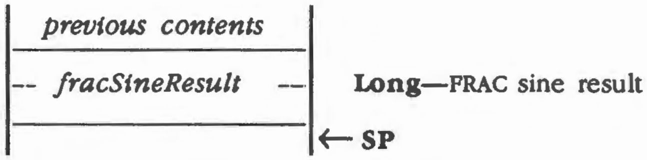
Takes a Fixed input (in radians) and returns its Frac sine.

Parameters

Stack before call



Stack after call



Errors

None

C

```
extern pascal Frac FracSin(angle)

Fixed     angle;
```

$140B FracSqrt

Takes a Frac input and returns a rounded Frac square root. The input is considered unsigned with the leading bit significant; that is, the input range is from 0 to almost 4.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *fracValue* --	**Long**—Original FRAC value
	← **SP**

Stack after call

previous contents	
-- *fracResult* --	**Long**—Rounded FRAC square root
	← **SP**

Errors

None

C

```
extern pascal Frac FracSqrt(fracValue)

Frac     fracValue;
```

$240B Hex2Int

Takes an Integer Math string representing a hexadecimal value and returns an unsigned Integer. The string must consist of digits and blanks. If the string does not fill up the space, pad the string at the left with blanks or zeros. The ASCII characters in the string may have the high-order bit either set or clear.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
— *strPtr* —	**Long**—POINTER to space for Integer Math string
strLength	**Word**—Length of Integer Math string
	← **SP**

Stack after call

previous contents	
intResult	**Word**—Unsigned INTEGER equivalent of the Integer Math string
	← **SP**

Errors

$0B02	`imIllegalChar`	Illegal character in string
$0B03	`imOverflow`	Hexadecimal value is greater than $FFFF

C

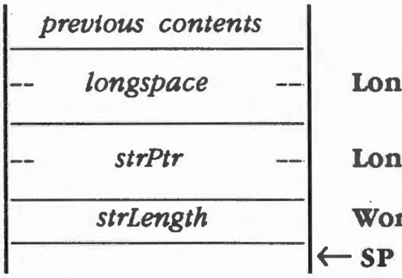
```
extern pascal Word Hex2Int(strPtr,strLength)

Pointer     strPtr;

Word     strLength;
```

$250B Hex2Long

Takes an Integer Math string representing a hexadecimal value and returns an unsigned Longint. The string must consist of digits and blanks. If the string does not fill up the space, pad the string at the left with blanks or zeros. The ASCII characters in the string may have the high-order bit either set or clear.

Parameters

Stack before call

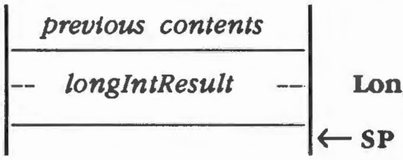


Long—Space for result

Long—POINTER to space for Integer Math string

Word—Length of Integer Math string

Stack after call



Long—Unsigned LONGINT equivalent of the Integer Math string

Errors

$0B02	`imIllegalChar`	Illegal character in string
$0B03	`imOverflow`	Hexadecimal value is greater than $FFFFFFFF

C

```
extern pascal LongWord Hex2Long(strPtr,strLength)

Pointer    strPtr;

Word     strLength;
```

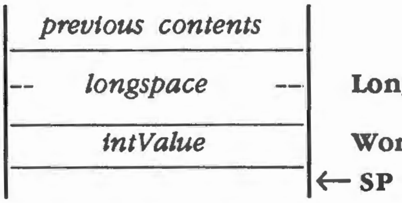
$2A0B HexIt

Takes an unsigned Integer and returns a 4-byte Integer Math string representing the value in hexadecimal format.

❖ *Note:* The difference between this routine and Int2Hex is that HexIt returns its result on the stack.

Parameters

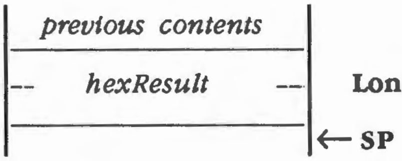
Stack before call



Long—Space for result

Word—Unsigned INTEGER to be converted

Stack after call



Long—4-byte Integer Math string representing value in hex format

Errors

None

C

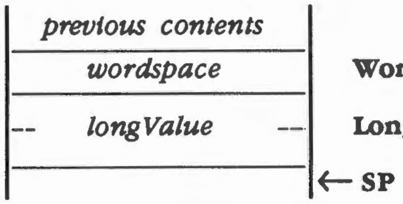
```
extern pascal LongWord HexIt(intValue)

Word     intValue;
```

$180B HiWord

Returns high-order word of a Long input.

Parameters

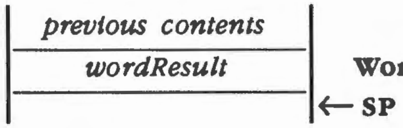
Stack before call



Word—Space for result

Long—Value whose high-order word will be returned

Stack after call



Word—High-order word of *longValue*

Errors

None

C

```
extern pascal Word HiWord(longValue)

LongWord     longValue;
```

$260B Int2Dec

Takes a signed or unsigned Integer and produces an Integer Math string representing the value in decimal format. The string must consist of digits and blanks. If the string does not fill up the space, pad the string at the left with blanks or zeros.

The ASCII characters in the string have the high-order bit clear. If *wordValue* is signed and negative, the string will contain an ASCII minus sign to the left of the most significant digit.

Parameters

Stack before call

previous contents	
wordValue	**Word**—INTEGER to be converted
-- *strPtr* --	**Long**—POINTER to space for Integer Math string
strLength	**Word**—Length of Integer Math string
signedFlag	**Word**—BOOLEAN; TRUE if *wordValue* is signed, FALSE if unsigned
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0B04	`imStrOverflow`	Length of string is too short to represent the value

C

```
extern pascal void Int2Dec(wordValue,strPtr,strLength,signedFlag)

Integer     wordValue;

Pointer     strPtr;

Word      strLength;

Boolean     signedFlag;
```

$220B Int2Hex

Takes an unsigned Integer and produces an Integer Math string representing the value in hexadecimal format. The string must consist of digits and blanks. If the string does not fill up the space, pad the string at the left with blanks or zeros. The ASCII characters in the output string have the high-order bit clear.

Parameters

Stack before call

previous contents	
intValue	**Word**—Unsigned INTEGER to be converted
— *strPtr* —	**Long**—POINTER to space for Integer Math string
strLength	**Word**—Length of Integer Math string
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0B04 `imStrOverflow` Length of string is too short to represent the value

C

```
extern pascal void Int2Hex(intValue,strPtr,strLength)

Word      intValue;

Pointer     strPtr;

Word      strLength;
```

$270B Long2Dec

Takes a signed or unsigned Longint value and produces an Integer Math string representing the value in decimal format. The string must consist of digits and blanks. If the string does not fill up the space, pad the string at the left with blanks or zeros.

The ASCII characters in the string have the high-order bit clear. If the *longValue* is signed and negative, the string will contain an ASCII minus sign to the left of the most significant digit.

Parameters

Stack before call

previous contents	
— *longValue* —	**Long**—LONGINT to be converted
— *strPtr* —	**Long**—POINTER to space for Integer Math string
strLength	**Word**—Length of Integer Math string
signedFlag	**Word**—BOOLEAN; TRUE if *longValue* is signed, FALSE if unsigned
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0B04	`imStrOverflow`	Length of string is too short to represent the value

C

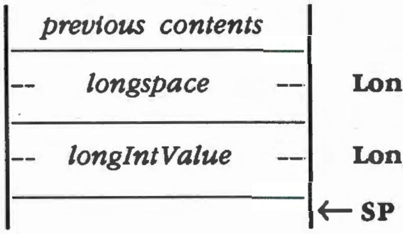
```
extern pascal void Long2Dec(longValue,strPtr,strLength,signedFlag)

Longint     longValue;

Pointer     strPtr;

Word     strLength;

Boolean     signedFlag;
```

$1A0B Long2Fix

Converts a specified Longint value to its corresponding Fixed value. Overflows are pinned to the most positive or negative value, depending on the sign of the input.

Parameters

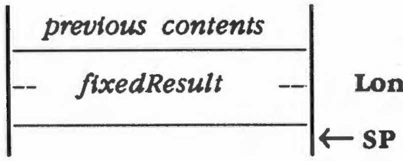
Stack before call



Long—Space for result

Long—LONGINT value to be converted

Stack after call



Long—FIXED result of the conversion; pinned if out-of-range

Errors

None

C

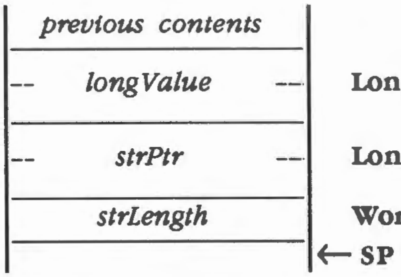
```
extern pascal Fixed Long2Fix(longValue)

Longint     longValue;
```

$230B Long2Hex

Takes an unsigned Longint value and produces an Integer Math string representing the value in hexadecimal format. The string must consist of digits and blanks. If the string does not fill up the space, pad the string at the left with blanks or zeros. The ASCII characters in the output string have the high-order bit clear.

Parameters

Stack before call

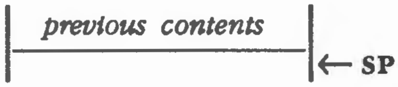


Long—Unsigned LONGINT value to be converted

Long—POINTER to space for Integer Math string

Word—Length of Integer Math string

Stack after call



Errors

$0B04	`imStrOverflow`	Length of string is too short to represent the value

C

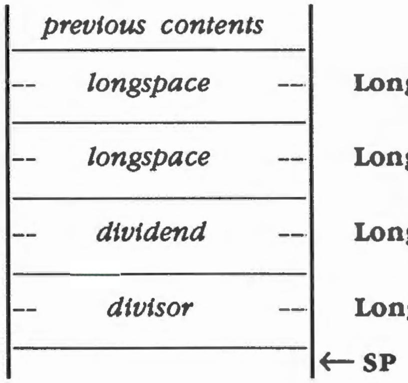
```
extern pascal void Long2Hex(longValue,strPtr,strLength)

LongWord     longValue;

Pointer     strPtr;

Word     strLength;
```

$0D0B LongDivide

Divides two unsigned Longint inputs and produces a Longint unsigned quotient and a Longint unsigned remainder.

Parameters

Stack before call



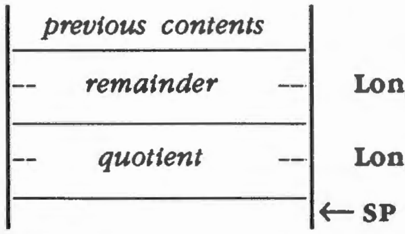
Long—Space for the remainder

Long—Space for the quotient

Long—Unsigned LONGINT dividend

Long—Unsigned LONGINT divisor

Stack after call



Long—Unsigned LONGINT remainder

Long—Unsigned LONGINT quotient

Errors

$0B01 `imBadInptParam` Bad input parameter

C

```
extern pascal LongDivRec LongDivide(dividend,divisor)
Longint     dividend;
Longint     divisor;
```

$0C0B LongMul

Multiplies two Longint values and produces a 64-bit result.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *longspace* --	**Long**—Space for result
-- *multiplicand* --	**Long**—First LONGINT input
-- *multiplier* --	**Long**—Second LONGINT input
	← **SP**

Stack after call

previous contents	
-- *msResult* --	**Long**—Most significant 32 bits of the result
-- *lsResult* --	**Long**—Least significant 32 bits of the result
	← **SP**

Errors

None

C

```
extern pascal LongMulRec LongMul(multiplicand,multiplier)

LongWord      multiplicand;

LongWord      multiplier;
```

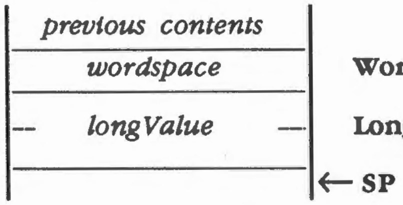
$190B LoWord

Returns low-order word of Long input.

❖ *Note:* To return the high-order word, use the HiWord routine.

Parameters

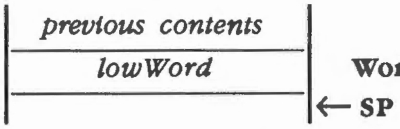
Stack before call



Word—Space for result

Long—Long input whose low-order word will be returned

Stack after call



Word—Low-order word of *longValue*

Errors

None

C

```
extern pascal Word LoWord(longValue)

LongWord     longValue;
```

$090B Multiply

Multiplies two Integer inputs and produces a Longint result.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
multiplicand	**Word**—First INTEGER input
multiplier	**Word**—Second INTEGER input
	← **SP**

Stack after call

previous contents	
-- *longIntResult* --	**Long**—LONGINT result
	← **SP**

Errors

None

C

```
extern pascal Longint Multiply(multiplicand,multiplier)

Integer     multiplicand;

Integer     multiplier;
```

$0A0B SDivide

Divides two Integers and produces a signed Integer quotient and a signed Integer remainder. The sign of the remainder will always be the same as the sign of the dividend.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
wordspace	**Word**—Space for result
dividend	**Word**—INTEGER dividend
divisor	**Word**—INTEGER divisor
	← **SP**

Stack after call

previous contents	
remainder	**Word**—INTEGER remainder
quotient	**Word**—INTEGER quotient
	← **SP**

Errors

$0B01 `imBadInptParam` Bad input parameter

C

```
extern pascal IntDivRec SDivide(dividend,divisor)

Integer     dividend;

Integer     divisor;
```

$0B0B UDivide

Divides two unsigned Integer inputs and produces an unsigned Integer quotient and an unsigned Integer remainder.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for remainder
wordspace	**Word**—Space for quotient
dividend	**Word**—INTEGER dividend
divisor	**Word**—INTEGER divisor
	← **SP**

Stack after call

previous contents	
remainder	**Word**—Unsigned INTEGER remainder
quotient	**Word**—Unsigned INTEGER quotient
	← **SP**

Errors

$0B01	`imBadInptParam`	Bad input parameter

C

```
extern pascal WordDivRec UDivide(dividend,divisor)
Word     dividend;
Word     divisor;
```

$200B X2Fix

Converts an Extended value to its corresponding Fixed value. Conversions are rounded. Overflows, NaNs (Not a Number), and infinities are pinned to the most positive or negative value, depending on the sign of the input.

❖ *Note:* For more information on NaNs and infinities, refer to the *Apple Numerics Manual.*

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *extendPtr* --	**Long**—POINTER to EXTENDED value
	← **SP**

Stack after call

previous contents	
-- *fixedResult* --	**Long**—Rounded FIXED result; pinned if out-of-range
	← **SP**

Errors

None

C

```
extern pascal Longint X2Fix(extendPtr)

ExtendPtr     extendPtr;
```

$210B X2Frac

Converts an Extended value to its corresponding Frac value. Conversions are rounded. Overflows, NaNs, and infinities are pinned to the most positive or negative value, depending on the sign of the input.

❖ *Note:* For more information on NaNs and infinities, refer to the *Apple Numerics Manual.*

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *extendPtr* --	**Long**—POINTER to EXTENDED value
	← **SP**

Stack after call

previous contents	
-- *fracResult* --	**Long**—Rounded FRAC result; pinned if out-of-range
	← **SP**

Errors

None

C

```
extern pascal Longint X2Frac(extendPtr)

ExtendPtr      extendPtr;
```

Integer Math Tool Set summary

This section briefly summarizes the constants and tool set error codes contained in the Integer Math Tool Set. There are no predefined data structures for the Integer Math Tool Set.

Important
These definitions are provided in the appropriate interface file.

Table 9-3
Integer Math Tool Set constants

Name	Value	Description
Limits		
`minLongint`	$80000000	Minimum negative signed Longint
`minFrac`	$80000000	Pinned value for negative Frac overflow
`minFixed`	$80000000	Pinned value for negative Fixed overflow
`minInt`	$8000	Minimum negative signed Integer
`maxInt`	$7FFF	Maximum positive signed Integer
`maxUInt`	$FFFF	Maximum unsigned Integer
`maxLongint`	$7FFFFFFF	Maximum positive signed Longint
`maxFrac`	$7FFFFFFF	Pinned value for positive Frac overflow
`maxFixed`	$7FFFFFFF	Pinned value for positive Fixed overflow
`maxULong`	$FFFFFFFF	Maximum unsigned Long
SignedFlag		
`unsignedFlag`	$0000	Value is not signed
`signedFlag`	$0001	Value is signed

Table 9-4
Integer Math Tool Set error codes

Code	Name	Description
$0B01	`imBadInptParam`	Bad input parameter
$0B02	`imIllegalChar`	Illegal character in string
$0B03	`imOverflow`	Integer or Longint overflow
$0B04	`imStrOverflow`	String overflow

Chapter 10

LineEdit Tool Set

This chapter describes the **LineEdit Tool Set.** The LineEdit routines provide line-editing capabilities that follow the *Human Interface Guidelines: The Apple Desktop Interface*.

These capabilities include

- Inserting new text
- Deleting characters that are backspaced over
- Translating mouse or arrow key activity into text selection
- Deleting selected text and possibly inserting it elsewhere
- Copying selected text without deleting it

The LineEdit routines support these standard user interface features:

- Positioning the insertion point by clicking the mouse
- Moving the insertion point one character at a time by using the Left and Right Arrow keys
- Moving the insertion point one word (plus its following spaces) at a time by using Option-Left Arrow or Option-Right Arrow
- Moving the insertion point to the beginning or end of the line by using Apple-Left Arrow or Apple-Right Arrow
- Selecting text by clicking and dragging with the mouse
- Selecting text by using Shift-Left Arrow and Shift-Right Arrow
- Selecting a word and its following spaces by double-clicking the mouse
- Selecting a word and its following spaces by using Shift-Option-Left Arrow or Shift-Option-Right Arrow
- Selecting the whole line by triple-clicking the mouse

- Selecting from the insertion point to the beginning or end of the line by using Shift-Apple-Left Arrow or Shift-Apple-Right Arrow
- Extending or shortening the selection by clicking the mouse while holding down the Shift key
- Deleting the selection or the character to the left of the insertion point by using Backspace
- Deleting the selection or the character to the right of the insertion point by using Control-F
- Deleting the selection or the whole line by using Control-X
- Deleting the selection or from the insertion point to the end of the line by using Control-Y
- Inverse highlighting the current text selection or displaying a blinking vertical bar at the insertion point
- Cutting (or copying) and pasting (LineEdit places cut or copied text into the LineEdit scrap)
- Left- or right-justified or centered text

In addition, the LineEdit routines that work with text that cannot be edited by the user support the following features:

- More than 256 characters per line
- Fill-justified text (text aligned with both the left and right margins)
- Automatic word wrap
- More than one font or stylistic variation per line

The LineEdit routines do not support

- Scrolling
- Fonts that kern characters (see the section "Fonts" in Chapter 16, "QuickDraw II," in Volume 2)
- "Intelligent" cut and paste operations (that is, LineEdit doesn't adjust spaces between words during cutting and pasting)
- Tabs

A preview of the LineEdit Tool Set routines

To introduce you to the capabilities of the LineEdit Tool Set, all LineEdit routines are grouped by function and briefly described in Table 10-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the LineEdit routines (discussed in alphabetical order).

Table 10-1
LineEdit Tool Set routines and their functions

Routine	Description
Housekeeping routines	
LEBootInit	Initializes the LineEdit Tool Set; called only by the Tool Locator—must not be called by an application
LEStartUp	Starts up the LineEdit Tool Set for use by an application
LEShutDown	Shuts down the LineEdit Tool Set and releases any workspace allocated to it
LEVersion	Returns the version number of the LineEdit Tool Set
LEReset	Returns an error if LineEdit is active
LEStatus	Indicates whether the LineEdit Tool Set is active
Edit record routines	
LENew	Allocates space for text, creates and initializes an edit record, and returns a handle to the new edit record
LEDispose	Releases the memory allocated for a specified edit record
LESetText	Incorporates a copy of the specified text into a specified edit record
LEGetTextHand	Returns a handle to the text of a specified edit record
LEGetTextLen	Returns the length of the text of a specified edit record in bytes
Insertion point and selection range routines	
LEIdle	Causes the caret at the insertion point (if any) in specified text to blink
LEClick	Controls the placement and highlighting of the selection range as determined by mouse events
LESetSelect	Sets the selection range of a specified edit record
LEActivate	Highlights the selection range or caret of a specified edit record
LEDeactivate	Unhighlights the selection range or caret of a specified edit record
Editing routines	
LEKey	Replaces the selection range or caret in the text of a specified edit record with a specified character and leaves an insertion point just past the inserted character
LECut	Removes the selection range from the text of a specified edit record and places it in the LineEdit scrap
LECopy	Copies the selection range from the text of a specified edit record to the LineEdit scrap
LEPaste	Replaces the selection range or caret in the text of a specified edit record with the contents of the LineEdit scrap and leaves an insertion point just past the inserted text
LEDelete	Removes the selection range from the text of a specified edit record without placing it in the LineEdit scrap
LEInsert	Takes specified text and inserts it just before the selection range or caret in the text of a specified edit record
Text display routines	
LEUpdate	Redraws the text of a specified edit record
LETextBox	Draws specified text in a specified rectangle, justifying the text as specified
LETextBox2	Draws specified text in a specified rectangle, justifying the text as specified (including fill justification), performing word wrap when necessary, and handling embedded changes
LESetJust	Sets the style of justification for the text of a specified edit record

(continued)

Table 10-1 (continued)
LineEdit Tool Set routines and their functions

Routine	Description
Scrap handling routines	
LEFromScrap	Copies the desk scrap to the LineEdit scrap
LEToScrap	Copies the LineEdit scrap to the desk scrap
LEScrapHandle	Returns a handle to the LineEdit scrap
LEGetScrapLen	Returns the size of the LineEdit scrap in bytes
LESetScrapLen	Sets the size of the LineEdit scrap to a specified number of bytes
Miscellaneous routines	
LESetHilite	Sets the *leHiliteHook* field of a specified edit record to a specified address
LESetCaret	Sets the *leCaretHook* field of a specified edit record to a specified address

Edit records

To edit a line of text on the screen, LineEdit needs to know where and how to display the text, where to store the text, and other information related to editing. This display, storage, and editing information is contained in an **edit record** that defines the complete editing environment.

You prepare to edit text by specifying a destination rectangle in which to draw the text and a view rectangle in which the text will be visible. LineEdit incorporates the rectangles and the drawing environment of the current GrafPort into an edit record and returns a handle to the record. Most LineEdit routines require you to pass this handle as a parameter.

In addition to the two rectangles and a description of the drawing environment, the edit record also contains

- A handle to the text to be edited
- A pointer to the GrafPort in which the text is displayed
- The current selection range, which determines which characters will be affected by the next editing operation
- The justification style of the text

You usually don't need to know the exact structure of an edit record, because you should always use the LineEdit routines to change any of the fields. However, in case your application needs to directly read some values from an edit record, the record's structure at the time of publication is illustrated in Figure 10-1.

❖ *Note:* The record may grow longer in the future.

Offset	Field	
$0 1 2 3	*ieLineHandle*	**Long**—Handle to text
4 5	*ieLength*	**Word**—INTEGER; current text length
6 7	*ieMaxLength*	**Word**—INTEGER; maximum text length
8 0F	*ieDestRect*	**Four words**—RECT data structure defining destination rectangle
10 17	*ieViewRect*	**Four words**—RECT data structure defining view rectangle
18 19 1A 1B	*iePort*	**Long**—POINTER to GrafPort
1C 1D	*ieLineHite*	**Word**—INTEGER; used for highlighting
1E 1F	*ieBaseHite*	**Word**—INTEGER; used for drawing text
20 21	*ieSelStart*	**Word**—INTEGER; used for start of selection range
22 23	*ieSelEnd*	**Word**—INTEGER; used for end of selection range
24 25	*ieActFlg*	**Word**—Reserved for internal use
26 27	*ieCarAct*	**Word**—Reserved for internal use
28 29	*ieCarOn*	**Word**—Reserved for internal use
2A 2D	*ieCarTime*	**Long**—Reserved for internal use
2E 31	*ieHiliteHook*	**Long**—POINTER to highlight routine
32 35	*ieCaretHook*	**Long**—POINTER to caret routine
36 37	*ieJust*	**Word**—Style of text justification; added in LineEdit Version 2.0

Figure 10-1
Edit record

Some of the fields of the record are discussed in the following sections.

The leDestRect and leViewRect fields

The **destination rectangle** is the rectangle that determines where the text will be drawn. The **view rectangle** is the rectangle within which the text is actually visible. In other words, the portion of the text that the user can see in the destination rectangle is determined by the view rectangle.

The view rectangle also determines the area in which mouse activity affects the text. If the user clicks or drags the mouse outside the view rectangle, that action does not affect the insertion point or selection range. In most cases, the view rectangle should be a few pixels larger than the destination rectangle. This provides users with some margin for error, so that their mouse operations still have the desired effect.

You specify both the destination and the view rectangle in the local coordinates of the GrafPort. The rectangles are illustrated in Figure 10-2.

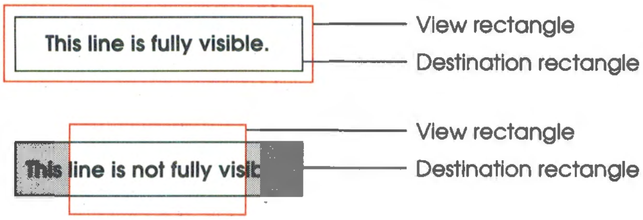


Figure 10-2
LineEdit destination and view rectangles

Edit operations may of course lengthen or shorten the text. LineEdit doesn't support scrolling or wrapping to the next line. If the text becomes too long to be enclosed by the destination rectangle, it's drawn beyond the edge as appropriate for the style of justification in effect, as illustrated in Figure 10-3.

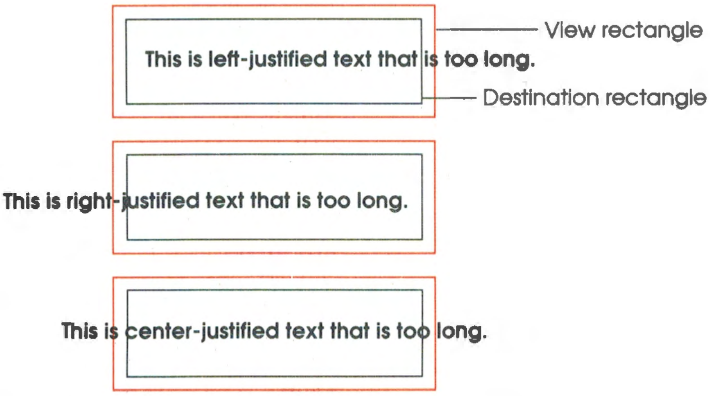

View rectangle
This is left-justified text that is too long.
Destination rectangle
This is right-justified text that is too long.
This is center-justified text that is too long.

Figure 10-3
Justification and the destination rectangle

Warning

If you're using right- or center-justification, make sure that the text does not extend past the left edge of the GrafPort; that is, the X value of the leftmost character must never be negative.

The bottom of the destination rectangle does not affect how the text is drawn; that is, LineEdit uses only the top of the destination rectangle.

❖ *Note:* If you draw a box around the text, the box should be outside the view rectangle. This prevents LineEdit from erasing the right side of the box when it redraws the text.

The leLineHite and leBaseHite fields

The *leBaseHite* field controls where the text is drawn relative to the top of the DestRect. The value of *leBaseHite* specifies the distance between the top of the DestRect and the base line (leading + ascent). The *leLineHite* field controls where the caret or highlighting of the selection range is drawn relative to the text. The value of *leLineHite* specifies the height of the line (leading + ascent + descent). See Figure 10-4.

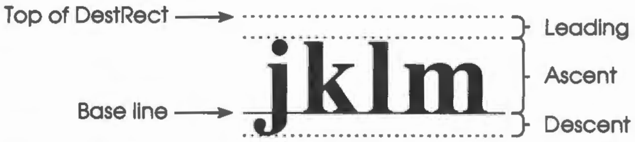


Figure 10-4
Line height and base line

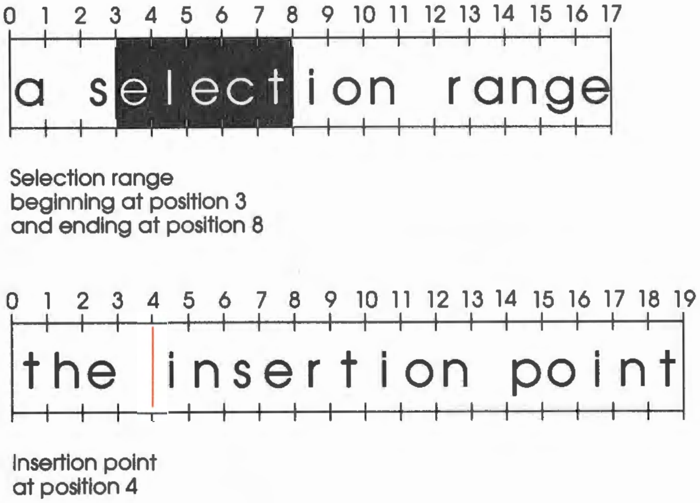
The leSelStart and leSelEnd fields

The edit record includes fields that specify the beginning and end of the **selection range.** The selection range is the series of characters where the next editing operation will occur. For example, the procedures that cut or copy from the text of an edit record do so to the selection range.

The selection range, which is inversely highlighted when the window and edit record are active, extends from the beginning character position to the end character position. A **character position** is an index into the text, with position 0 corresponding to the first character. Figure 10-5 shows a selection range between positions 3 and 8, consisting of five characters (the character at position 8 isn't included). The end position of a selection range may be one greater than the position of the last character of the text, so that the selection range can include the last character.

❖ *Note:* LineEdit highlights the selection range by calling the QuickDraw II routine InvertRect, not by swapping the text and background colors. You can supply your own highlight routine; see the section "The leHiliteHook and leCaretHook Fields" and "LESetHilite" in this chapter.

If the selection range is empty—that is, if its beginning and end positions are the same—that position is the text's **insertion point,** the position where characters will be inserted. By default, it's marked with a blinking caret (actually a vertical bar). See Figure 10-5.



Selection range
beginning at position 3
and ending at position 8

Insertion point
at position 4

Figure 10-5
Selection range and insertion point

If you call the LEKey routine to insert characters when there's a selection range of one or more characters rather than an insertion point, the routine automatically deletes the selection range and replaces it with an insertion point before inserting the characters.

The leHiliteHook and leCaretHook fields

The *leHiliteHook* and *leCaretHook* fields are used for text highlighting and for drawing the caret. These fields are initialized to $00000000. You can set the contents of these fields by calling the LESetHilite and LESetCaret routines.

If you store the address of a routine in *leHiliteHook*, that routine is used instead of the QuickDraw II routine InvertRect whenever a selection range is to be highlighted or unhighlighted. For example, you can write a routine that underlines selection ranges instead of highlighting them. The routine will be called with the stack containing a pointer to the rectangle enclosing the text being highlighted or unhighlighted. When your routine finishes its highlighting procedure, the routine must remove the pointer from the stack and return with an RTL.

If you store the address of a routine in the *leCaretHook* field, that routine is called whenever the caret needs to be drawn or removed. This enables you to change the appearance of the caret. The routine will be called with the stack containing a pointer to the rectangle that encloses the caret. When your routine finishes its actions on the caret, the routine must remove the pointer from the stack and return with an RTL.

Using the LineEdit Tool Set

This section discusses how the LineEdit Tool Set routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

The LineEdit Tool Set depends upon the presence of the tool sets shown in Table 10-2 and requires that at least the indicated version of the tool set be present.

Table 10-2
LineEdit Tool Set—other tool sets required

Tool set number		Tool set name	Minimum version needed
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0
$04	#04	QuickDraw II	1.1
$06	#06	Event Manager	1.0

In addition, if you are going to use LEToScrap, LEFromScrap, or LETextBox2, you must load and start up additional tool sets. See the descriptions of those routines for more information.

The first LineEdit call that your application must make is LEStartUp. Conversely, when you quit your application, you must make the LEShutDown call.

Call LENew to allocate an edit record; it returns a handle to the record. Most text editing routines require you to pass this handle as a parameter. You can also obtain the handle to the text of a specified edit record by using the LEGetTextHand routine, or you can determine the length of the text in a specified edit record by using the LEGetTextLen routine.

When you're completely done with an edit record and want to dispose of it, call LEDispose.

To make a blinking caret appear at the insertion point, call the LEIdle routine as often as possible (at least once each time through the main event loop); if it's not called often enough, the caret will blink irregularly.

When a mouse-down event occurs in the view rectangle (and the window is active), call the LEClick routine. LEClick controls the placement and highlighting of the selection range in response to mouse activity, including supporting use of Shift-Click to make extended selections.

Key-down, auto-key, and mouse events that pertain to text editing can be handled by several LineEdit routines:

- LEKey inserts characters; deletes characters backspaced over; controls the placement and highlighting of the selection range in response to the Left Arrow and Right Arrow keys; and handles the Control-F, Control-X, and Control-Y commands.
- LECut transfers the selection range to the LineEdit scrap, removing the selection range from the text.
- LEPaste inserts the contents of the LineEdit scrap. By calling LECut, changing the insertion point, and then calling LEPaste, you can perform a cut and paste operation, moving text from one place to another.
- LECopy copies the selection range to the LineEdit scrap. By calling LECopy, changing the insertion point, and then calling LEPaste, you can make multiple copies of text.
- LEDelete removes the selection range (without transferring it to the scrap). You can use LEDelete to implement the Clear command.
- LEInsert inserts specified text.

After each editing procedure, LineEdit redraws the text if necessary. You never have to set the selection range or insertion point yourself; LEClick and the editing routines leave it where it should be. If you want to set the selection range directly, however—to highlight an initial default name or value, for example—you can use the LESetSelect routine.

To implement cutting and pasting of text between different applications or between applications and desk accessories, you need to transfer the text between the **LineEdit scrap** (a private scrap used only by LineEdit) and the Scrap Manager's desk scrap. To do this, use the LEFromScrap and LEToScrap routines.

When an update event is reported for a window associated with an edit record, call LEUpdate (along with the Window Manager routine BeginUpdate, the QuickDraw II routine EraseRect, and the Window Manager routine EndUpdate) to redraw the text.

Your application will usually call the LEActivate and LEDeactivate routines when an activate event is reported for a window associated with an edit record. LEActivate highlights the selection range or displays a caret at the insertion point; LEDeactivate unhighlights the selection range or removes the caret.

The LESetText routine lets you change the text being edited. For example, if your application has several separate pieces of text that must be edited one at a time, you don't have to allocate an edit record for each of them. Allocate a single edit record and then use LESetText to change the text.

You can use the LESetJust routine to change the justification style of the text.

If you want to draw noneditable text in any specified rectangle, you can use the LETextBox or LETextBox2 routines.

If you want a custom caret or custom highlighting, use the LESetCaret or LESetHilite routines, respectively.

Moving or scrolling windows that contain LineEdit items

If you want to move a window that contains one or more LineEdit items, you must not be using the Window Manager routine TaskMaster. If you aren't using TaskMaster, you should call the LEDeactivate routine for each active LineEdit item, move the window, and then call the LEActivate routine for each previously active LineEdit item.

If you want to scroll a window with LineEdit items in it, you will need to use the Window Manger routines StartDrawing, BeginUpdate, EraseRect, EndUpdate, and the QuickDraw II routine SetOrigin in addition to the LineEdit Tool Set LEUpdate routine, in the following order:

1. Call StartDrawing for the current window's GrafPort.
2. Call BeginUpdate for the current window's GrafPort.
3. Call EraseRect to erase the view rectangle.
4. Call LEUpdate for the current edit record.
5. Call EndUpdate for the current window's GrafPort.
6. Call SetOrigin with coordinates (0,0).

$0114 LEBootInit

Initializes the LineEdit Tool Set; called only by the Tool Locator.

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 Call must not be made by an application.

$0214 LEStartUp

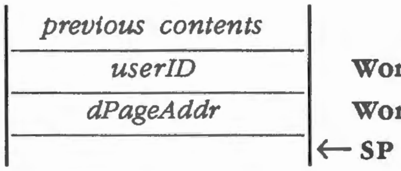
Starts up the LineEdit Tool Set for use by an application and allocates a handle for the LineEdit scrap. The scrap is initially empty.

Important

You should call LEStartUp even if your application doesn't use LineEdit, so that desk accessories, dialog boxes, and alert boxes will work correctly.

Parameters

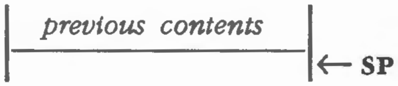
Stack before call



Word—ID number of the application

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

Stack after call



Errors

$1401	`leDupStrtUpErr`	LEStartUp already called
Memory Manager errors		NewHandle routine called; any errors returned unchanged

C

```
extern pascal void LEStartUp(userID,dPageAddr)

Word      userID;

Word      dPageAddr;
```

$0314 LEShutDown

Shuts down the LineEdit Tool Set and discards the LineEdit scrap.

Important
If your application has started up the LineEdit Tool Set, the application must make this call before it quits.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

$1403	`leNotActiveErr`	LineEdit Tool Set not active
Memory Manager errors		DisposeHandle called; any errors returned unchanged

C

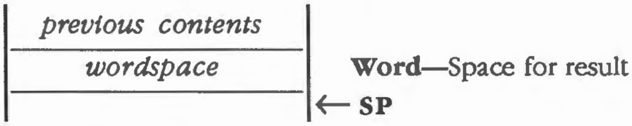
```
extern pascal void LEShutDown()
```

$0414 LEVersion

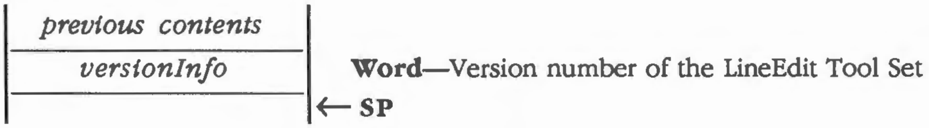
Returns the version number of the LineEdit Tool Set.

Parameters

Stack before call



Stack after call



Errors

None

C

```
extern pascal Word LEVersion()
```

$0514 LEReset

Returns an error if LineEdit is active.

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

$1402	`leResetError`	LineEdit can't be reset

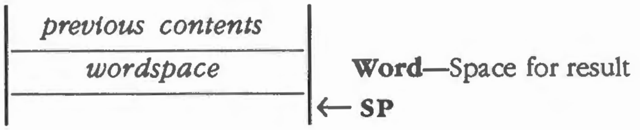
C Call must not be made by an application.

$0614 LEStatus

Indicates whether the LineEdit Tool Set is active.

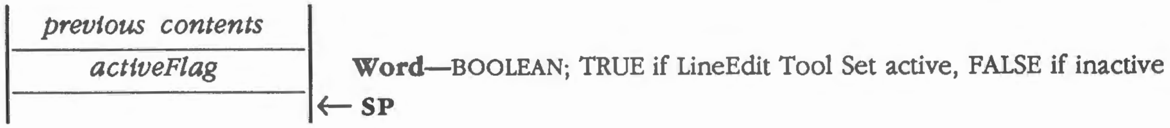
Parameters

Stack before call



Word—Space for result

Stack after call



Word—BOOLEAN; TRUE if LineEdit Tool Set active, FALSE if inactive

Errors None

C

```
extern pascal Boolean LEStatus()
```

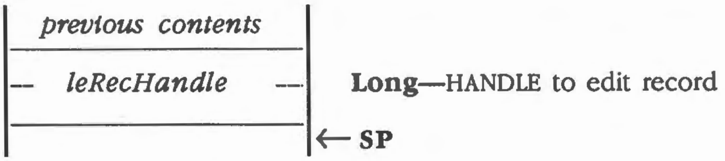
$0F14 LEActivate

Highlights the selection range in specified text. If the selection range is an insertion point, the routine displays a caret.

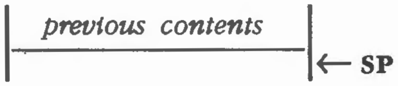
Your application will usually call LEActivate when an activate event is reported for a window associated with an edit record.

Parameters

Stack before call



Stack after call



Errors

Memory Manager errors	Returned unchanged
QuickDraw II errors	Returned unchanged

C

```
extern pascal void LEActivate(leRecHandle)
LERecHndl     leRecHandle;
```

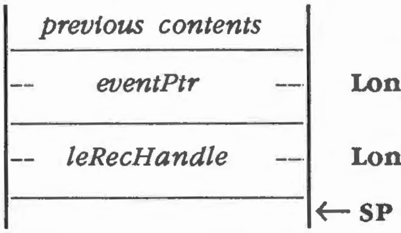
$0D14 LEClick

Controls the placement and highlighting of the selection range as determined by mouse events. Call LEClick whenever a mouse-down event occurs in the view rectangle of the edit record specified by *leRecHandle* and the window associated with that edit record is active. The *eventPtr* parameter should be a pointer to the mouse-down event record.

LEClick unhighlights the old selection range unless the selection range is being extended. If the mouse moves, meaning that a drag is occurring, LEClick expands or shortens the selection range accordingly. In the case of a double click, the word under the cursor (plus any following spaces) becomes the selection range; dragging expands or shortens the selection a word at a time. In the case of a triple click, the entire line becomes the selection range. LEClick keeps control until the mouse button is released.

Parameters

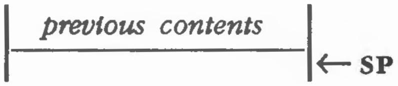
Stack before call



Long—POINTER to mouse-down event record

Long—HANDLE to edit record

Stack after call



Errors

Memory Manager errors	Returned unchanged
QuickDraw II errors	Returned unchanged

C

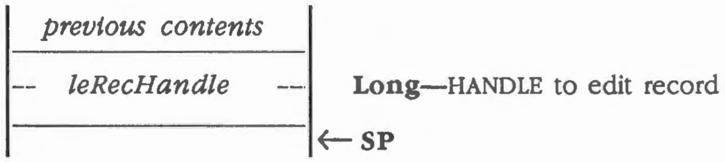
```
extern pascal void LEClick(eventPtr,leRecHandle)
EventRecordPtr     eventPtr;
LERecHndl      leRecHandle;
```

$1314 LECopy

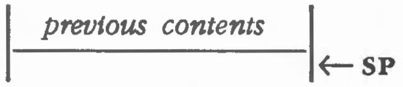
Copies the selection range from the specified text into the LineEdit scrap. Anything previously in the scrap is deleted. The selection range is not deleted. If the selection range is an insertion point, the scrap is emptied.

Parameters

Stack before call



Stack after call



Errors

Memory Manager errors	Returned unchanged
QuickDraw II errors	Returned unchanged

C

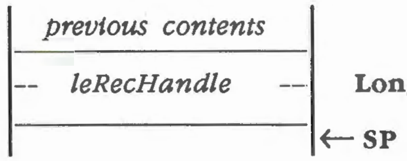
```
extern pascal void LECopy(leRecHandle)
LERecHndl     leRecHandle;
```

$1214 LECut

Removes the selection range from the specified text and places it in the LineEdit scrap. The text is redrawn as necessary. Anything previously in the scrap is deleted. If the selection range is an insertion point, the scrap is emptied.

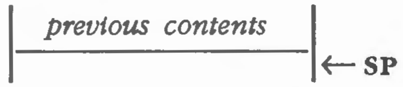
Parameters

Stack before call



Long—HANDLE to edit record

Stack after call



Errors

Memory Manager errors	Returned unchanged
QuickDraw II errors	Returned unchanged

C

```
extern pascal void LECut(leRecHandle)

LERecHndl     leRecHandle;
```

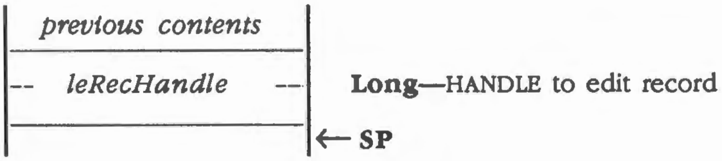
$1014 LEDeactivate

Unhighlights the selection range in the specified text. If the selection range is an insertion point, the routine removes the caret.

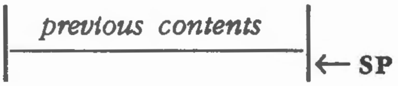
Your application will usually call LEDeactivate when an activate event (for a window becoming inactive) is reported for a window associated with an edit record.

Parameters

Stack before call



Stack after call



Errors

Memory Manager errors	Returned unchanged
QuickDraw II errors	Returned unchanged

C

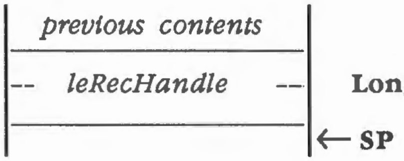
```
extern pascal void LEDeactivate(leRecHandle)

LERecHndl     leRecHandle;
```

$1514 LEDelete

Removes the selection range from the specified text and redraws the text as necessary. LEDelete is the same as LECut except that it doesn't transfer the selection range to the scrap. If the selection range is an insertion point, nothing happens.

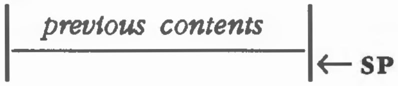
Parameters

Stack before call



Long—HANDLE to edit record

Stack after call



Errors

Memory Manager errors	Returned unchanged
QuickDraw II errors	Returned unchanged

C

```
extern pascal void LEDelete(leRecHandle)

LERecHndl     leRecHandle;
```

$0A14 LEDispose

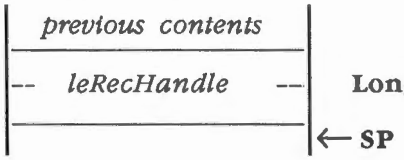
Releases the memory allocated for a specified edit record. Call this routine when you're completely through with an edit record.

Important

All edit records created by calling LENew must be disposed of by calling LEDispose before calling LEShutDown.

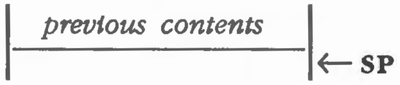
Parameters

Stack before call



Long—HANDLE to edit record

Stack after call



Errors

Memory Manager errors	DisposeHandle called; any errors returned unchanged

C

```
extern pascal void LEDispose(leRecHandle)

LERecHndl     leRecHandle;
```

$1914 LEFromScrap

Copies the desk scrap to the LineEdit scrap. If the number of characters in the desk scrap is greater than 256, an error is returned, and the scrap is not copied.

Important
The Scrap Manager must have already been loaded and started up.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

$1404	`leScrapErr`	Desk scrap too big to copy
Scrap Manager errors		GetScrapHandle and GetScrapSize called; any errors returned unchanged

C

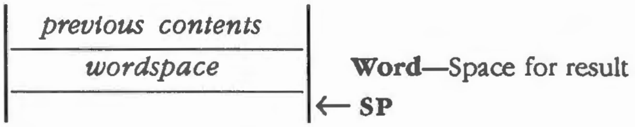
```
extern pascal void LEFromScrap()
```

$1C14 LEGetScrapLen

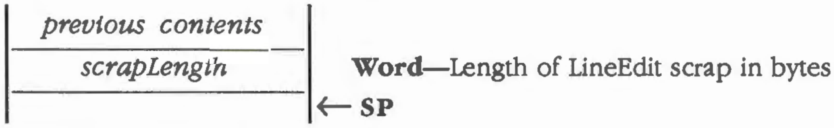
Returns the length of the LineEdit scrap in bytes.

Parameters

Stack before call



Stack after call



Errors

None

C

```
extern pascal Word LEGetScrapLen()
```

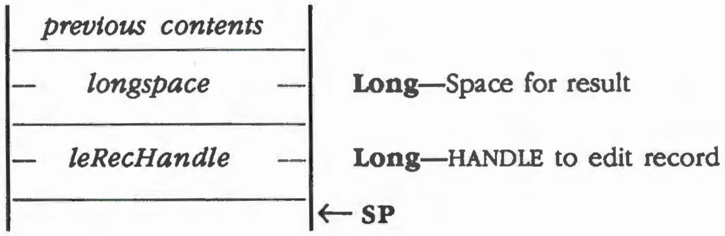
$2214 LEGetTextHand

Returns a handle to the text of a specified edit record.

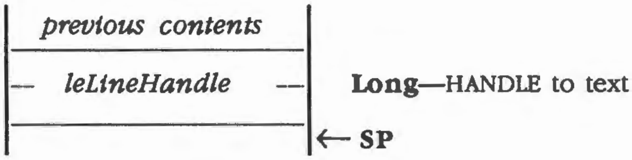
❖ *Note:* This call is available only in Version 2.0 or later versions of the LineEdit Tool Set.

Parameters

Stack before call



Stack after call



Errors

None

C

```
extern pascal Handle LEGetTextHand(leRecHandle)

LERecHndl     leRecHandle;
```

$2314 LEGetTextLen

Returns the length of the text, in bytes, of a specified edit record.

❖ *Note:* This call is available only in Version 2.0 or later versions of the LineEdit Tool Set.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *leRecHandle* --	**Long**—HANDLE to edit record
	← **SP**

Stack after call

previous contents	
leLength	**Word**—Length of the text in bytes
	← **SP**

Errors

None

C

```
extern pascal Word LEGetTextLen(leRecHandle)
LERecHndl     leRecHandle;
```

$0C14 LEIdle

Makes a blinking caret appear at the insertion point (if any) in the specified text. The caret appears only when the window and the edit record are active.

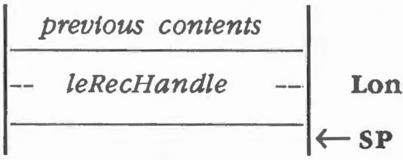
LineEdit observes a **minimum blink interval:** No matter how often LEIdle is called, the time between blinks will never be less than the minimum interval. The user can adjust the minimum blink interval with the Control Panel desk accessory.

To provide a constant frequency of blinking, LEIdle should be called as often as possible, at least once each time through the main event loop. Call it more than once if your application performs an unusually large amount of processing each time through the loop.

❖ *Note:* LEIdle actually only needs to be called when the window associated with the edit record is active.

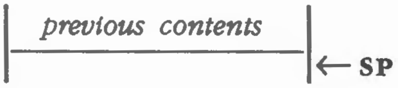
Parameters

Stack before call



Long—HANDLE to edit record

Stack after call



Errors

Memory Manager errors	Returned unchanged
QuickDraw II errors	Returned unchanged

C

```
extern pascal void LEIdle(leRecHandle)

LERecHndl      leRecHandle;
```

$1614 LEInsert

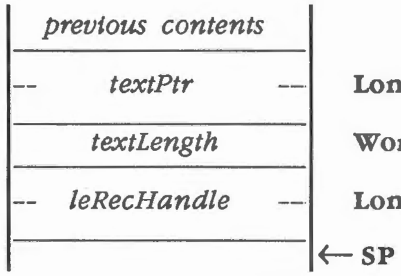
Takes specified text and inserts it just before the selection range in the specified text and redraws the text as necessary. The routine doesn't affect either the selection range or the scrap.

Important

The text pointed to by *textPtr* must not contain a Pascal-type length byte; the length of the text is passed as the *textLength* parameter.

Parameters

Stack before call



Long—POINTER to text to be inserted

Word—INTEGER; number of characters to be inserted

Long—HANDLE to edit record

Stack after call

previous contents
← SP

Errors

Memory Manager errors	Returned unchanged
QuickDraw II errors	Returned unchanged

C

```
extern pascal void LEInsert(textPtr,textLength,leRecHandle)
Pointer     textPtr;
Word     textLength;
LERecHndl     leRecHandle;
```

$1114 LEKey

Replaces the selection range in the specified text with a specified character and leaves an insertion point just past the inserted character. If the selection range is an insertion point, LEKey just inserts the character there. LEKey redraws the text as necessary.

Every time a keyboard event that your application decides should be handled by LineEdit is reported, you should call LEKey.

❖ *Note:* LEKey inserts every character passed in *theKey* (except for Backspace, Control-F, Control-X, Control-Y, Left Arrow, and Right Arrow, as detailed in the section "Special Characters" in this chapter), so it's up to the application to filter out all undesired characters (such as command keys and other control characters).

Parameters

Stack before call

previous contents	
theKey	**Word**—Key reported by the event record
modifiers	**Word**—Copy of the *modifiers* field in the event record
-- *leRecHandle* --	**Long**—HANDLE to edit record
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors	Returned unchanged
QuickDraw II errors	Returned unchanged

C

```
extern pascal void LEKey(theKey,modifiers,leRecHandle)
Word      theKey;
Word      modifiers;
LERecHndl     leRecHandle;
```

Special characters

If *theKey* contains one of the special characters, LEKey takes the action shown in Table 10-3.

Table 10-3
LEKey actions and special characters

Character	Action
Backspace	Deletes the selection range or the character immediately to the left of the insertion point
Control-F	Deletes the selection range or the character immediately to the right of the insertion point
Control-X	Deletes the selection range or the entire line
Control-Y	Deletes the selection range or the text from the insertion point to the end of the line
Left Arrow or Right Arrow	Moves the insertion point one character at a time
Option-Left Arrow or Option-Right Arrow	Moves the insertion point one word at a time
Apple-Left Arrow or Apple-Right Arrow	Moves the insertion point to the beginning or end of the line
Shift-Left Arrow or Shift-Right Arrow	Extends or shortens the selection range one character at a time
Shift-Option-Left Arrow or Shift-Option-Right Arrow	Extends or shortens the selection range one word at a time
Shift-Apple-Left Arrow or Shift-Apple-Right Arrow	Selects from the insertion point to the beginning or end of the line

$0914 LENew

Allocates space for text, creates and initializes an edit record for that text, and returns a handle to the new edit record. Call LENew once for every edit record you want allocated. The edit record incorporates the drawing environment of the current GrafPort and is initialized with an insertion point at character position 0.

❖ *Note:* The caret won't appear until you call the LEActivate routine.

The text will be limited to the length specified in the *maxTextLen* parameter.

Important
The view rectangle must not be empty. For example, don't make its right edge less than its left edge. If you don't want any text visible, specify a rectangle off the screen instead.

If you want the LineEdit item to appear in a special font, you must call the Font Manager routine InstallFont before you make the LENew call. If you want to change the font for an existing LineEdit item, you must make a LEDispose call for that record, make the InstallFont call, and then call LENew to create a new LineEdit record.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *destRectPtr* --	**Long**—POINTER to RECT data structure in current GrafPort's coordinates
-- *viewRectPtr* --	**Long**—POINTER to RECT data structure in current GrafPort's coordinates
maxTextLen	**Word**—INTEGER specifying number of bytes to allocate for text (1–256)
	← **SP**

Stack after call

previous contents	
-- *leRecHandle* --	**Long**—HANDLE to new edit record
	← **SP**

Errors

Memory Manager errors	NewHandle called; any errors returned unchanged

C

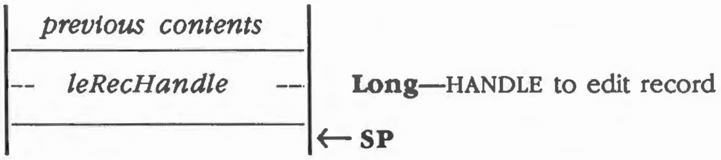
```
extern pascal LERecHndl LENew(destRectPtr,viewRectPtr,maxTextLen)
Rect *destRectPtr;
Rect *viewRectPtr;
Word     maxTextLen;
```

$1414 LEPaste

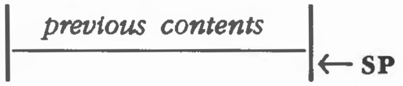
Replaces the selection range in the specified text with the contents of the LineEdit scrap and leaves an insertion point just past the inserted text. The text is redrawn as necessary. If the scrap is empty, the selection range is deleted. If the selection range is an insertion point, LEPaste inserts the scrap at that point.

Parameters

Stack before call



Stack after call



Errors

Memory Manager errors	Returned unchanged
QuickDraw II errors	Returned unchanged

C

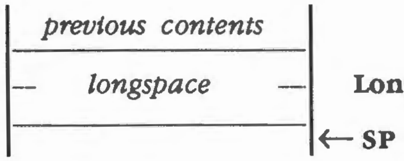
```
extern pascal void LEPaste(leRecHandle)

LERecHndl     leRecHandle;
```

$1B14 LEScrapHandle

Returns a handle to the LineEdit scrap.

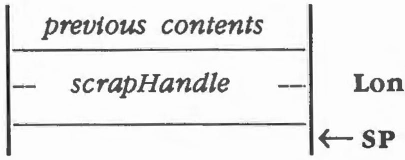
Parameters

Stack before call



Long—Space for result

Stack after call



Long—HANDLE to LineEdit scrap

Errors

None

C

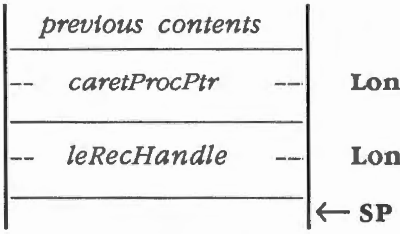
```
extern pascal Handle LEScrapHandle()
```

$1F14 LESetCaret

Sets the *leCaretHook* field in a specified edit record to point to a custom caret drawing procedure. LineEdit will use that procedure to both draw and erase the caret.

Parameters

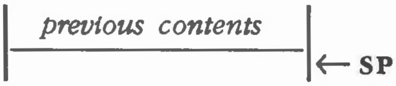
Stack before call



Long—POINTER to custom caret drawing procedure; NIL for standard

Long—HANDLE to edit record

Stack after call



Errors

None

C

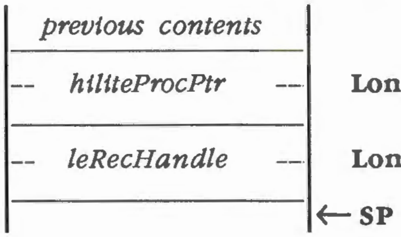
```
extern pascal void LESetCaret(caretProcPtr,leRecHandle)

VoidProcPtr     caretProcPtr;

LERecHndl     leRecHandle;
```

$1E14 LESetHilite

Sets the *leHiliteHook* field in the edit record to point to a custom highlighting procedure. LineEdit will use that procedure to both highlight and unhighlight the selection range.

Parameters

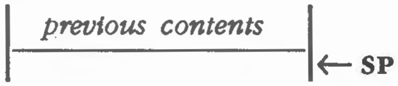
Stack before call



Long—POINTER to custom highlighting procedure; NIL for standard

Long—HANDLE to edit record

Stack after call



Errors

None

C

```
extern pascal void LESetHilite(hiliteProcPtr,leRecHandle)

VoidProcPtr     hiliteProcPtr;

LERecHndl     leRecHandle;
```

$2114 LESetJust

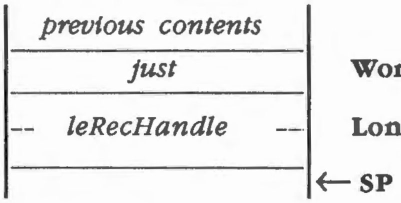
Sets the justification style of the text of the specified edit record. The text is justified to the destination rectangle supplied by the LENew call, as described in the section "The leDestRect and leViewRect Fields" in this chapter.

After you call LESetJust, call the Window Manager routine InvalRect so that the text will be redrawn using the new justification style.

❖ *Note:* This call is available only in Version 2.0 or later versions of the LineEdit Tool Set.

Parameters

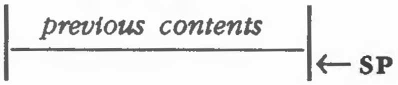
Stack before call



Word—0 = left justified, 1 = centered, $FFFF = right justified

Long—HANDLE to edit record

Stack after call



Errors

None

C

```
extern pascal void LESetJust(just,leRecHandle)

Word       just;

LERecHndl       leRecHandle;
```

$1D14 LESetScrapLen

Sets the size of the LineEdit scrap to a specified number of bytes. If *newLength* is greater than 256, it is set to 256.

Parameters

Stack before call

previous contents

newLength **Word**—Number of bytes, 0–256

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void LESetScrapLen(newLength)

Word     newLength;
```

$0E14 LESetSelect

Sets the selection range in the specified text.

The text selected is between *selStart* and *selEnd* in the text specified by *leRecHandle*. The old selection range is unhighlighted, and the new one is highlighted. If *selStart* equals *selEnd,* the selection range is an insertion point, and a caret is displayed.

The *selStart* parameter must be less than or equal to *selEnd.* If *selEnd* is beyond the last character of the text, the position just past the last character is used.

Parameters

Stack before call

previous contents	
selStart	**Word**—INTEGER; start of selection range, 0–256
SelEnd	**Word**—INTEGER; end of selection range, 0–256
-- *leRecHandle* --	**Long**—HANDLE to edit record
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors	Returned unchanged
QuickDraw II errors	Returned unchanged

C

```
extern pascal void LESetSelect(selStart,selEnd,leRecHandle)

Word      selStart;

Word      selEnd;

LERecHndl      leRecHandle;
```

$0B14 LESetText

Incorporates a copy of the specified text into the specified edit record.

The selection range is set to an insertion point at the end of the text. If the *textLength* parameter is greater than the maximum text length allowed for the edit record, only the maximum number of characters allowed will be copied into the edit record.

Important
The text pointed to by *textPtr* must not contain a Pascal-type length byte; the length of the text is passed as the *textLength* parameter.

LESetText doesn't redraw the text, so call the Window Manager routine InvalRect afterward, if necessary. Pass the InvalRect routine a pointer to the edit record's view rectangle, so that the view rectangle will be added to the window's update region.

Parameters

Stack before call

previous contents	
textPtr	**Long**—POINTER to text
textLength	**Word**—INTEGER; number of characters in text
leRecHandle	**Long**—HANDLE to edit record
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors	Returned unchanged
QuickDraw II errors	Returned unchanged

C

```
extern pascal void LESetText(textPtr,textLength,leRecHandle)

Pointer     textPtr;

Word     textLength;

LERecHndl     leRecHandle;
```

$1814 LETextBox

Draws the specified text in the specified rectangle, justifying the text as specified. LETextBox supports left, right, and centered justification.

❖ *Note:* The LETextBox2 routine also supports fill justification; see the section "LETextBox2" in this chapter.

LETextBox performs an EraseRect on the rectangle before drawing the text and then clips the text to the rectangle. LETextBox is not limited to a single line on the screen as the other LineEdit routines are. The routine wraps to the next line whenever a carriage return character (ASCII $0D) occurs in the text string. However, LETextBox does not automatically wrap when it reaches the right side of the specified rectangle; use the LETextBox2 routine for that function.

Important

The text pointed to by *textPtr* must not contain a Pascal-type length byte; the length of the text is passed as the *textLength* parameter.

LETextBox creates its own edit record, which it deletes when it's finished, so the text it draws cannot be edited. The routine does not allocate space for the text or make any copies of the text.

Parameters

Stack before call

previous contents	
— *textPtr* —	**Long**—POINTER to text
textLength	**Word**—INTEGER, length of text including carriage returns (0–32767)
— *rectPtr* —	**Long**—POINTER to a RECT specified in local coordinates
just	**Word**—0 = left justified, 1 = centered, $FFFF = right justified
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors	Returned unchanged
QuickDraw II errors	Returned unchanged

C

```
extern pascal void LETextBox(textPtr,textLength,rectPtr,just)

Pointer     textPtr;

Word      textLength;

Rect *rectPtr;

Word      just;
```

$2014 LETextBox2

Draws the specified text in a specified rectangle, justifying the text as specified. LETextBox2 supports left, right, centered, and fill justification and can also support embedded changes.

Important

LETextBox2 is available only in Version 2.0 or later versions of the LineEdit Tool Set. Also, in addition to the tool sets required by the LineEdit Tool Set, LETextBox2 requires that the Integer Math Tool Set be loaded and started up.

LETextBox2 performs an EraseRect on the rectangle before drawing the text and then clips the text to the rectangle. LETextBox2 is not limited to a single line on the screen. The routine will wrap to the next line whenever a carriage return character (ASCII $0D) occurs in the text string, or will automatically wrap to the next line whenever the text reaches the right side of the rectangle.

Important

The text pointed to by *textPtr* must not contain a Pascal-type length byte; the length of the text is passed as the *textLength* parameter.

LETextBox2 does not create an edit record, so the text it draws cannot be edited. The routine does not allocate space for the text or make any copies of the text.

Parameters

Stack before call

previous contents	
-- *textPtr* --	**Long**—POINTER to text (text can include embedded changes)
textLength	**Word**—Includes carriage returns and embedded changes (0–32767)
-- *rectPtr* --	**Long**—POINTER to RECT data structure specified in local coordinates
just	**Word**—0 = left , 1 = centered, $FFFF = right, 2 = fill justified
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors	Memory Manager errors	Returned unchanged

C

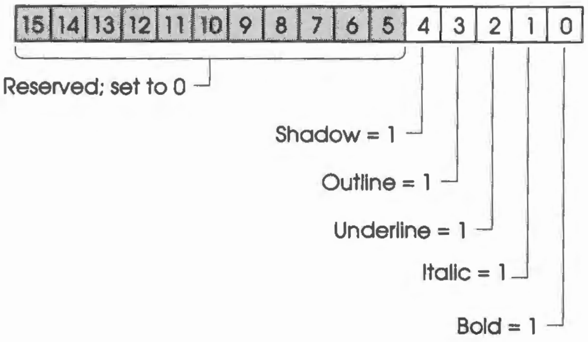
```
extern pascal void LETextBox2(textPtr,textLength,rectPtr,just)

Pointer     textPtr;

Word     textLength;

Rect *rectPtr;

Word     just;
```

Using embedded changes

You can change the appearance of the displayed text by using embedded changes in the text. To specify an embedded change, you insert a delta flag byte ($01) into the text, followed by a change flag byte, followed by the appropriate data. The values for the change flag byte and the data are given in Table 10-4.

Table 10-4
LETextBox2 embedded change values

Change flag value	Parameter changed	Data size	Data description

S ($53)	Font style	Word	Font style or combination of styles, as shown in the following bit flag (1 means that the style is applied):
			
			 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
			Reserved; set to 0
			Shadow = 1
			Outline = 1
			Underline = 1
			Italic = 1
			Bold = 1
			❖ *Note:* QuickDraw II Auxiliary must be loaded and started up for the shadow, outline, and italic styles to be applied.

(continued)

Table 10-4 (continued)
LETextBox2 embedded change values

Change flag value	Parameter changed	Data size	Data description
C ($43)	Foreground color	Word	Color of foreground, as an index into the current color table.
B ($42)	Background color	Word	Color of background, as an index into the current color table.
F ($46)	Font	FontID	Font ID specifying which font family number, font style, and font size to use. If the *famNum* field of the font ID is 0, it's translated into the family number of the system font. A family number of $FFFF is not allowed, nor is a font size of 0. ❖ *Note:* If this change flag is used, the Font Manager must have already been loaded and started up. See Chapter 8, "Font Manager," for more information.
J ($4A)	Justification	Word	How the text should be justified, as follows: 0 Left justification 1 Centered $FFFF Right justification 2 Fill justification (text is justified to both margins)
L ($4C)	Left margin	Word	Number of pixels to indent from the left edge of the destination rectangle.
M ($4D)	Right margin	Word	Number of pixels to indent from the right edge of the destination rectangle.
X ($58)	Extra spacing	Word	Number of extra pixels to be added to normal line spacing. This number may be negative, which can lead to overlapping text.

A simple assembly-language example that changes the style of a single word is as follows:

```
MyText dc c'This is '
       dc h'01',c'S',i'$0001'
       dc c'My'
       dc h'01',c'S',i'$0000'
       dc c' text'
```

LETextBox2 will print this text as

This is **My** text

$1A14 LEToScrap

Copies the LineEdit scrap to the desk scrap.

Important
The Scrap Manager must have already been loaded and started up.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors

Scrap Manager errors	PutScrap called; any errors returned unchanged

C

```
extern pascal void LEToScrap()
```

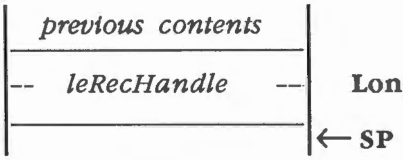
$1714 LEUpdate

LEUpdate redraws the text of the specified edit record. Your application should call LEUpdate every time an update event for a window associated with an edit window is reported. LEUpdate should be called after you call the Window Manager routine BeginUpdate and the QuickDraw II routine EraseRect, and before you call the Window Manager routine EndUpdate.

If you do not include the EraseRect call, the caret may sometimes remain visible when the window is deactivated.

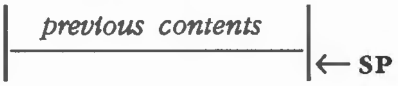
Parameters

Stack before call



Long—HANDLE to edit record

Stack after call



Errors

Memory Manager errors	Returned unchanged
QuickDraw II errors	Returned unchanged

C

```
extern pascal void LEUpdate(leRecHandle)

LERecHndl     leRecHandle;
```

LineEdit Tool Set summary

This section briefly summarizes the constants, data structures, and tool set errors contained in the LineEdit Tool Set.

Important

These definitions are provided in the appropriate interface file.

Table 10-5
LineEdit Tool Set constants

Name	Value	Description
Justification		
`leJustLeft`	$0000	Left justification
`leJustCenter`	$0001	Center justification
`leJustRight`	$FFFF	Right justification
`leJustFill`	$0002	Fill justification (text is justified to both the left and right margins)

Table 10-6
LineEdit Tool Set data structures

Name	Offset	Type	Definition
LERec (Edit Record)			
leLineHandle	$00	Handle	Handle to text
leLength	$04	Integer	Current length of text
leMaxLength	$06	Integer	Maximum text length
leDestRect	$08	Rect	Destination rectangle
leViewRect	$10	Rect	View rectangle
lePort	$18	GrafPortPtr	Pointer to GrafPort
leLineHite	$1C	Integer	Used for highlighting
leBaseHite	$1E	Integer	Used for drawing the text
leSelStart	$20	Integer	Start of selection range
leSelEnd	$22	Integer	End of selection range
leActFlg	$24	Word	Reserved for internal use
leCarAct	$26	Word	Reserved for internal use
leCarOn	$28	Word	Reserved for internal use
leCarTime	$2A	Longint	Reserved for internal use
leHiliteHook	$2E	VoidProcPtr	Pointer to highlight routine
leCaretHook	$32	VoidProcPtr	Pointer to caret routine
leJust	$36	Word	Text justification (Version 2.0 or later)

Note: The actual assembly-language equates have a lowercase *o* (the letter) in front of all of the names given in this table.

Table 10-7
LineEdit Tool Set error codes

Code	Name	Description
$1401	`leDupStrtUpErr`	LEStartUp already called
$1402	`leResetError`	Can't reset LineEdit
$1403	`leNotActiveErr`	LineEdit not active
$1404	`leScrapErr`	Desk scrap too big to copy

Chapter 11

List Manager

The **List Manager** allows your application to display lists of similar data from which the user can choose. Generally, the **list** is a list of strings, such as names of files, fonts, or pictures. However, a list may also be a list of icons, pictures, colors, patterns, or any graphic display. A list appears vertically, with a vertical scroll bar to the right of it.

The List Manager does not use a direct page, so it needs 768 bytes of stack space.

❖ *Note:* At the time of publication, the system did not allow a list control to be part of a dialog window.

The List Manager is basically a custom control definition procedure. The List Manager routines help your application interface with the control.

A preview of the List Manager routines

To introduce you to the capabilities of the List Manager, all List Manager routines are grouped by function and briefly described in Table 11-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the List Manager routines (discussed in alphabetical order).

Table 11-1
List Manager routines and their functions

Routine	Description
Housekeeping routines	
ListBootInit	Initializes the List Manager; called only by the Tool Locater—must not be called by an application
ListStartUp	Starts up the List Manager for use by an application
ListShutDown	Shuts down the List Manager when an application quits
ListVersion	Returns the version number of the List Manager
ListReset	Resets the List Manager; called only when the system is reset—must not be called by an application
ListStatus	Indicates whether the List Manager is active
List routines	
CreateList	Creates a list control using a specified list record
NewList	Resets the list control according to a specified list record
SortList	Alphabetizes a specified list by rearranging the array of member records
GetListDefProc	Returns a pointer to the list control's definition procedure
Member routines	
DrawMember	Draws one or all members of a specified list
NextMember	Searches a specified list record, starting with a specified member, and returns a pointer to the member record of the next selected member found
ResetMember	Searches a specified list record, starting with the first member, returns a pointer to the member record of the first selected member found, and deselects (but does not redraw) the member
SelectMember	Selects a specified member, deselects any other selected members in the list, and scrolls the list so that the specified member is at the top of the list display

List controls and list records

A **list control** is a custom control that displays lists of similar data. The initial appearance of the list control is defined by the **list record,** as shown in Figure 11-1.

Important

If you want to change the list control after it is created, you must change the list control record, not the list record. Changing information in the list record after the list control is created does not affect the control. See the section "List Control Records" in this chapter.

Offset	Field	
$0 7	*listRect*	**Four words**—RECT defining list's enclosing rectangle minus the scroll bar
8 9	*listSize*	**Word**—Number of members in list; 16383 ($3FFF) maximum
0A 0B	*listView*	**Word**—Maximum number of members that can be seen at once
0C 0D	*listType*	**Word**—Type of members (see Figure 11-2)
0E 0F	*listStart*	**Word**—Number of member at which the list will start
10 11 12 13	*listCtl*	**Long**—Handle to control associated with this list
14 15 16 17	*listDraw*	**Long**—POINTER to routine to draw list's members; NIL for default
18 19	*listMemHeight*	**Word**—Height of each member in pixels; normally 10 for standard list using system font
1A 1B	*listMemSize*	**Word**—Number of bytes in a member record
1C 1D 1E 1F	*listPointer*	**Long**—POINTER to the member list; an array of member records (see Figure 11-3)
20 21 22 23	*listRefCon*	**Long**—Reserved for application's use
24 25 26 27	*listScrollClr*	**Long**—POINTER to color table; NIL for default (see Figure 11-4)

Figure 11-1
List record

The fields of the list record are as follows:

listRect: This field defines the list's enclosing rectangle, not including the list's vertical scroll bar. The field is a RECT data structure defining the top, left, bottom, and right pixel coordinates. The specified points are local to the window in which the list appears; that is, the top number defines the number of pixels from the top of the window's content region, and the left number defines the numbers of pixels from the left side of the window's content region. This is true no matter where on the screen the window appears.

The minimum height of the rectangle is 40 pixels. The right side of the rectangle should be the left side plus the length of the largest possible string in the list, plus 20 (because strings are printed 10 pixels in from the left side of the list, and the right margin should also be 10).

The bottom side can be computed using the following simple equation:

topSide + (*listView* * *listMemHeight*) + 2

2 + (10 * 10) + 2 = 104

listSize: This field specifies the total number of members in the list. This value can range from $0000 to $3FFF, where 0 means no members in the list. The pointer to the list is stored in the *listPointer* field.

listView: This field specifies the number of members that could be displayed at any one time in the list. If the total number of members is greater than *listView,* the user must scroll to see the additional members.

If the total number of members is smaller than *listView,* empty slots will be displayed. In this case, the list's scroll bar will be inactive because all members are in view and the user does not need to scroll to see the data.

listType: Two bits of this field are defined, and the others reserved.

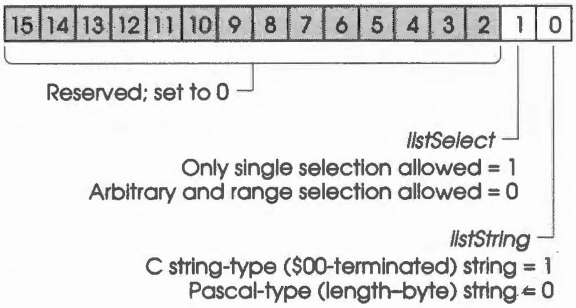

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Reserved; set to 0
listSelect
Only single selection allowed = 1
Arbitrary and range selection allowed = 0
listString
C string-type ($00-terminated) string = 1
Pascal-type (length-byte) string = 0

Figure 11-2
The *listType* bit flag

The *listString* bit should be 1 if the *memPtr* field of the member records points to C-type (null-terminated) strings. The bit should be 0 if the *memPtr* field of the member records point to Pascal-type strings (strings beginning with a length byte). If the *listDraw* field of the list record is not 0, the list is considered custom, and this bit's state is not used by the List Manager.

The *listSelect* bit should be 1 if only the single selection mode is allowed. The bit should be 0 if the arbitrary and range selection modes are allowed (see the section "Selection Process" in this chapter).

listStart: This field specifies the list control's starting value. It is the number of the first member appearing in the view (at the top). The value 1 indicates the first member, and 0 generally indicates that there are no members in the list.

In most cases this field should be 1. However, if you want to display a certain member in the middle of the view, then *listStart* should be member number less half the view size but greater than 0.

listCtl: The List Manager routine CreateList stores the handle to the list control in this field.

listDraw: This field indicates the address of a routine in your application that will draw members. If this field is NIL, the list is considered a standard list, and the *memPtr* fields in every member record must point to either a Pascal-type or a C-type string. If this field is not NIL, the address is called by the List Manager whenever a member must be drawn. When the draw routine is entered the stack will appear as follows:

previous contents		
-- *rectPtr* --		**Long**—POINTER to RECT enclosing member
-- *memRectPtr* --		**Long**—POINTER to member's record; handle is locked
-- *listHandle* --		**Long**—HANDLE to list control
RTL	*RTL*	**Three bytes**—RTL address
RTL	← **SP**	

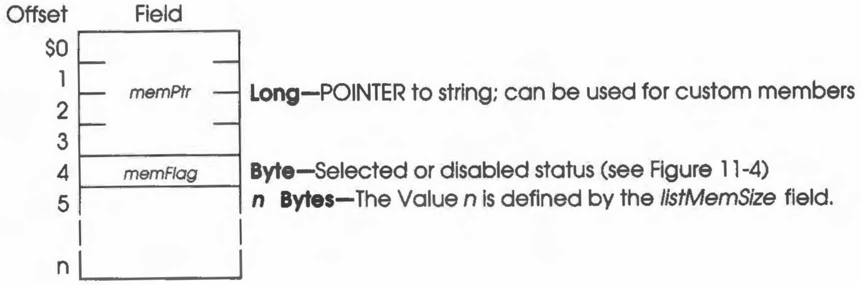
There is no return value, and the input parameters must be removed from the stack before returning.

The flags in the member's *memFlag* field should be considered when drawing a member.

listMemHeight: This field specifies the height of each member in pixels. The minimum height is 10 pixels, a good default value when using a standard list with the system font. This field is provided for lists that use a different size font or custom lists.

listMemSize: This field specifies the size of each member record in the list. The minimum size is 5. If, for example, you need to associate a different ID number with each member, you could extend the member by a word and set *listMemSize* to 7. The word for the ID number should be added after the *memFlag* field in each member record.

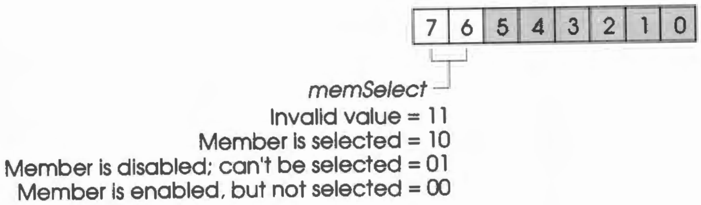
listPointer: This field indicates a pointer to the list of member records. If the list is empty, the pointer must be NIL. Each member record must be as shown in Figure 11-3.



Offset
Field
$0
1
2
3
4
5
n
memPtr
memFlag
Long—POINTER to string; can be used for custom members
Byte—Selected or disabled status (see Figure 11-4)
n **Bytes**—The Value *n* is defined by the *listMemSize* field.

Figure 11-3
Member record

The *memFlag* controls whether the member can be selected by the user as well as the current state of the member, as shown in Figure 11-4. A selected member is drawn using the highlight colors. Note that a disabled member cannot be selected because bit 6 of the member must be clear to be selected.

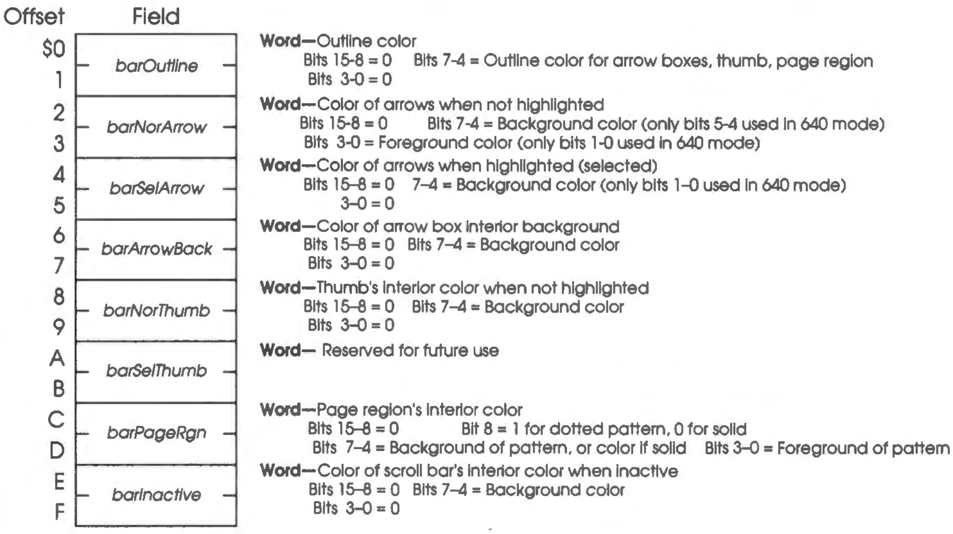


7
6
5
4
3
2
1
0
memSelect
Invalid value = 11
Member is selected = 10
Member is disabled; can't be selected = 01
Member is enabled, but not selected = 00

Figure 11-4
The *memFlag* bit flag

listRefCon: This field is reserved for use by the application.

listScrollClr: This field indicates a pointer to the color table of the list's scroll bar, or NIL for the default color table. The color table is shown in Figure 11-5.



Word—Outline color
Bits 15-8 = 0 Bits 7-4 = Outline color for arrow boxes, thumb, page region
Bits 3-0 = 0

Word—Color of arrows when not highlighted
Bits 15-8 = 0 Bits 7-4 = Background color (only bits 5-4 used in 640 mode)
Bits 3-0 = Foreground color (only bits 1-0 used in 640 mode)

Word—Color of arrows when highlighted (selected)
Bits 15–8 = 0 7–4 = Background color (only bits 1–0 used in 640 mode)
3–0 = 0

Word—Color of arrow box interior background
Bits 15–8 = 0 Bits 7–4 = Background color
Bits 3–0 = 0

Word—Thumb's interior color when not highlighted
Bits 15–8 = 0 Bits 7–4 = Background color
Bits 3–0 = 0

Word— Reserved for future use

Word—Page region's interior color
Bits 15–8 = 0 Bit 8 = 1 for dotted pattern, 0 for solid
Bits 7–4 = Background of pattern, or color if solid Bits 3–0 = Foreground of pattern

Word—Color of scroll bar's interior color when inactive
Bits 15–8 = 0 Bits 7–4 = Background color
Bits 3–0 = 0

Figure 11-5
List Manager scroll bar color table

An assembly-language example of a list record follows.

```
MyListRec  dc  i2'2,120,104,270'  ;listRect       Rectangle starts 2 pixels down, 120 in.
           dc  i2'4'              ;listSize       4 members
           dc  i2'10'             ;listView       10 members would be visible at once;
                                  ;               thus, 6 empty slots, scroll bar inactive
           dc  i2'0'              ;listType       Multiple-selection OK, Pascal strings
           dc  i2'1'              ;listStart      First member at top of view
           dc  i4'0'              ;listCtl        Space for list control's handle
           dc  i4'0'              ;listDraw       Default draw routine
           dc  i2'10'             ;listMemHeight  Each member 10 pixels tall
           dc  i2'5'              ;listMemSize    Each member record is 5 bytes
           dc  i4'MyList'         ;listPointer    Pointer to my list of member records
           dc  i4'0'              ;listRefCon     Application defined
           dc  i4'0'              ;listScrollClr  Default scroll bar color table
```

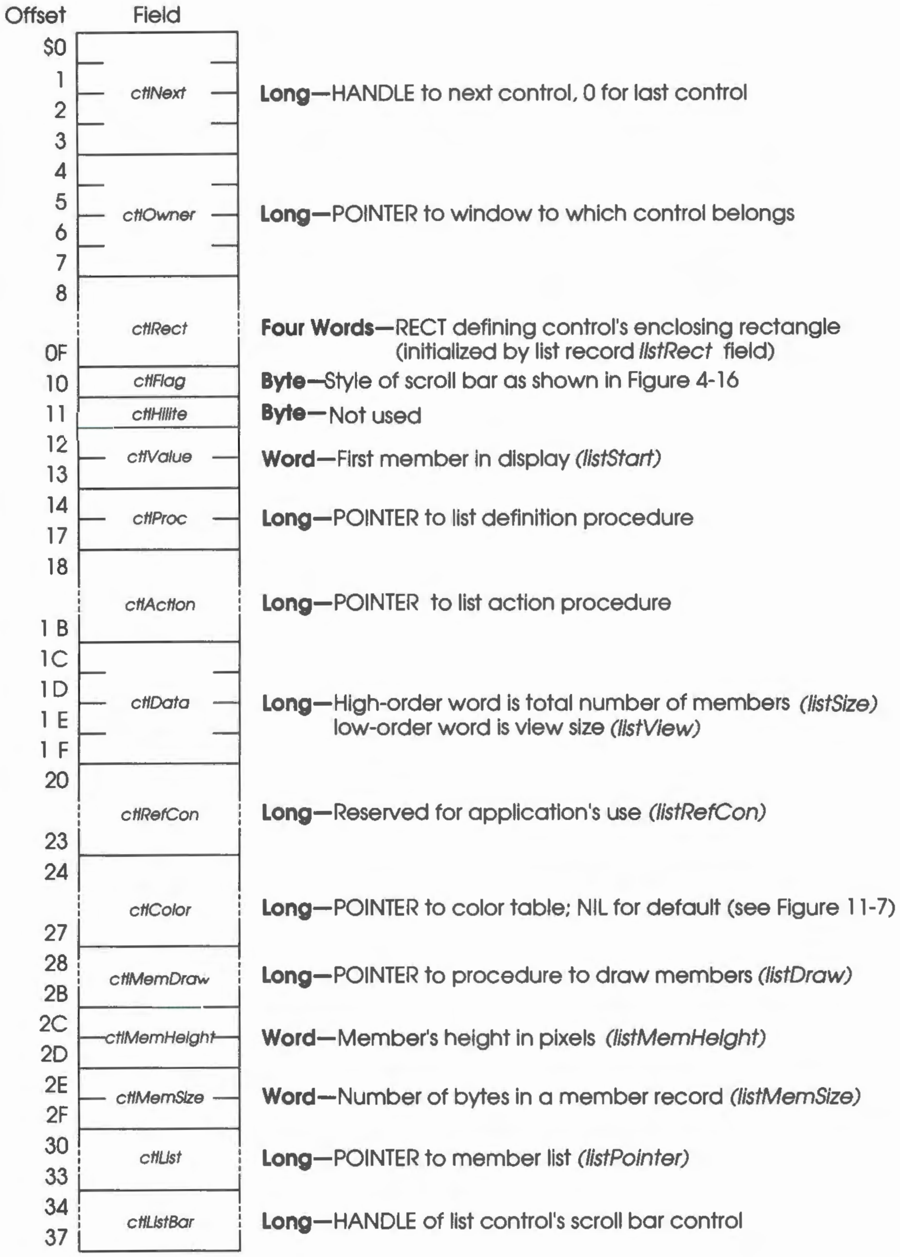
```
MyList     dc  i4'member1',i1'0'  ;Pointer to first member's string / memFlag byte
           dc  i4'member2',i1'0'  ;Pointer to first member's string / memFlag byte
           dc  i4'member3',i1'0'  ;Pointer to first member's string / memFlag byte
           dc  i4'member4',i1'0'  ;Pointer to first member's string / memFlag byte
;
member1    dc  i1'8',c'String 1'  ;String for member 1
member2    dc  i1'8',c'String 2'  ;String for member 2
member3    dc  i1'8',c'String 3'  ;String for member 3
member4    dc  i1'8',c'String 4'  ;String for member 4
```

List control records

The **list control record** is used to define the appearance of the control after the control has been created. Some fields in the list control record are initialized using corresponding values in the list record; those fields are identified in Figure 11-6.

Your application can add and delete members from the list after the list control is created by adding or subtracting from the list and then changing the low-order word of *ctlData*. The SortList routine can be used to realphabetize the list. If SortList is called after the list control is already visible (that is, after the list control record has already been created), then DrawMember should be called to redraw the entire list.

The complete definition of a list control record is shown in Figure 11-6.

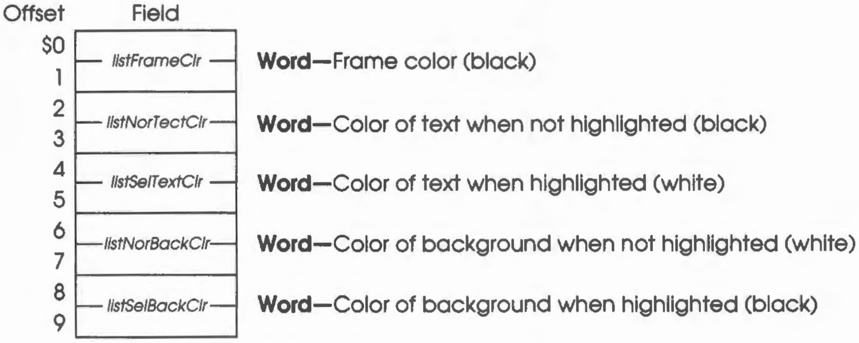


Offset	Field	
$0 1 2 3	*ctlNext*	**Long**—HANDLE to next control, 0 for last control
4 5 6 7	*ctlOwner*	**Long**—POINTER to window to which control belongs
8 0F	*ctlRect*	**Four Words**—RECT defining control's enclosing rectangle (initialized by list record *listRect* field)
10	*ctlFlag*	**Byte**—Style of scroll bar as shown in Figure 4-16
11	*ctlHilite*	**Byte**—Not used
12 13	*ctlValue*	**Word**—First member in display *(listStart)*
14 17	*ctlProc*	**Long**—POINTER to list definition procedure
18 1 B	*ctlAction*	**Long**—POINTER to list action procedure
1C 1D 1 E 1 F	*ctlData*	**Long**—High-order word is total number of members *(listSize)* low-order word is view size *(listView)*
20 23	*ctlRefCon*	**Long**—Reserved for application's use *(listRefCon)*
24 27	*ctlColor*	**Long**—POINTER to color table; NIL for default (see Figure 11-7)
28 2B	*ctlMemDraw*	**Long**—POINTER to procedure to draw members *(listDraw)*
2C 2D	*ctlMemHeight*	**Word**—Member's height in pixels *(listMemHeight)*
2E 2F	*ctlMemSize*	**Word**—Number of bytes in a member record *(listMemSize)*
30 33	*ctlList*	**Long**—POINTER to member list *(listPointer)*
34 37	*ctlListBar*	**Long**—HANDLE of list control's scroll bar control

Figure 11-6
List control record

The address of a list control's color table is stored in the *ctlColor* field of the list control record and can be set after the list control is created by the Control Manager routine CreateControl. A default color table is used if the *ctlColor* field of a list control's record is NIL (actual colors depend on the color palette used).

The color table and its default values are shown in Figure 11-7.



Offset	Field	
$0 1	*listFrameClr*	**Word**—Frame color (black)
2 3	*listNorTectClr*	**Word**—Color of text when not highlighted (black)
4 5	*listSelTextClr*	**Word**—Color of text when highlighted (white)
6 7	*listNorBackClr*	**Word**—Color of background when not highlighted (white)
8 9	*listSelBackClr*	**Word**—Color of background when highlighted (black)

Figure 11-7
List control color table

Figure 11-8 shows a list using the default color values.

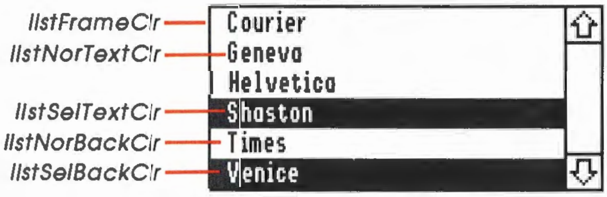


Figure 11-8
Color table and example list

Using the List Manager

This section discusses how the List Manager routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter. The List Manager depends upon the presence of the tool sets shown in Table 11-2 and requires that at least the indicated version of the tool set be present.

Table 11-2
List Manager—other tool sets required

Tool set number		Tool set name	Minimum version needed
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0
$03	#03	Miscellaneous Tool Set	1.0
$04	#04	QuickDraw II	1.0
$06	#06	Event Manager	1.0
$0E	#14	Window Manager	1.3
$10	#16	Control Manager	1.3

Your application must make the ListStartUp call before it makes any other List Manager calls. Conversely, when your application quits, it must make the ListShutDown call.

Your first step is to build a list. After the list is generated, you may want to call the SortList routine, which will alphabetize the list if the member strings are composed of ASCII characters.

Next, your application should create the list control by calling the CreateList routine, which in turn calls the Control Manager routine NewControl and passes information from the list record to that routine. CreateList also passes the address of a custom control definition procedure inside of the List Manager.

The list definition procedure also creates a vertical scroll bar on the right side of the list. Both the list control and scroll bar are added to the given window's control list. From that point on, the Control Manager routine DrawControls will draw both controls, and KillControls or CloseWindow will delete them.

Fields in the control record can be obtained and changed by various Control Manager calls. See the description of the list control record for field definitions.

Important
Do not use the following Control Manager routines with list controls: DisposeControl, DragControl, DrawOneCtl, EraseControl, HideControl, HiliteControl, MoveControl, NewControl, SetCtlAction, SetCtlParams, SetCtlTitle, SetCtlValue, ShowControl.

Selection modes

The List Manager allows the user to choose between three possible selection modes: single, arbitrary, and range. The Apple and Shift keys are used to choose the selection mode. The state of the Apple and shift keys is checked only when the user first presses the mouse button. After that, the user can release the key, and the selection mode remains in effect until the user releases the mouse button. The three modes are as follows:

- **Single mode:** The user selects single mode by simply pressing the mouse button and not pressing either the Apple or Shift key. Any selection the user makes deselects all other selected members. Thus, when the user drags the mouse, the selection moves from one member to another.
- **Arbitrary mode:** If the user holds down the Apple key and then presses the mouse button, already selected members are not deselected. This allows unselected members to be between selected members in the list. Dragging is allowed in this mode, so any enabled member the mouse is dragged over will be selected. The arbitrary mode overrides the range mode if the user presses both the Apple and Shift keys.
- **Range mode:** If the user holds down the Shift key and then presses the mouse button, a range of members is selected, and all of the members outside the range are deselected. A range is defined as follows: The first selected member in the list is the beginning of the range. The end of the range is the current selection if it appears after the beginning of the range. If the current selection is the first selection in the list, and therefore the beginning of the range, then the end of the range is the last selected member in the list. This concept is illustrated in Figure 11-9.

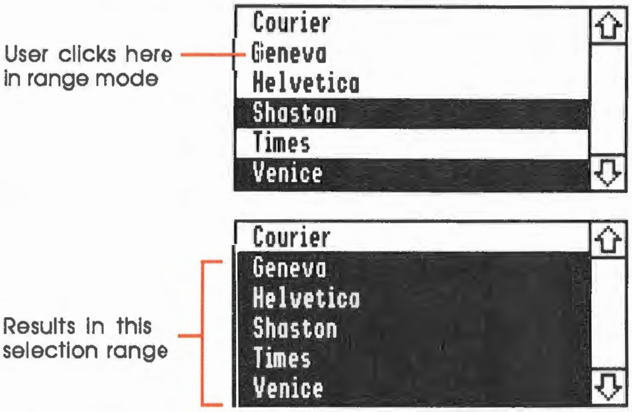


Figure 11-9
Range-mode selection

The current selection is both the beginning and end of the range if it is the only selection in the list.

The application can shut off arbitrary and range mode to allow only single selections by setting the *listSelect* bit (bit 1) in the *listType* field of the list record.

$011C ListBootInit

Initializes the List Manager; called only by the Tool Locator.

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 Call must not be made by an application.

$021C ListStartUp

Starts up the List Manager for use by an application.

Important
Your application must make this call before it makes any other List Manager calls.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void ListStartup()
```

$031C ListShutDown

Shuts down the List Manager.

Important

If your application has started up the List Manager, the application must make this call before it quits.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void ListShutDown()
```

$041C ListVersion

Returns the version number of the List Manager.

Parameters

Stack before call

```
|  previous contents  |
|---------------------|
|      wordspace      |    Word—Space for output
|---------------------|
|                     |<— SP
```

Stack after call

```
|  previous contents  |
|---------------------|
|     versionInfo     |    Word—Version number of the List Manager
|---------------------|
|                     |<— SP
```

Errors None

C

```
extern pascal Word ListVersion()
```

$051C ListReset

Resets the List Manager; called only when a system reset occurs.

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

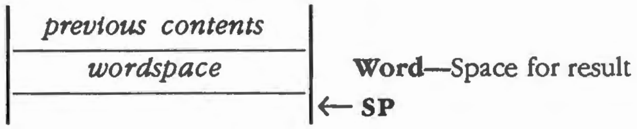
C Call must not be made by an application.

$061C ListStatus

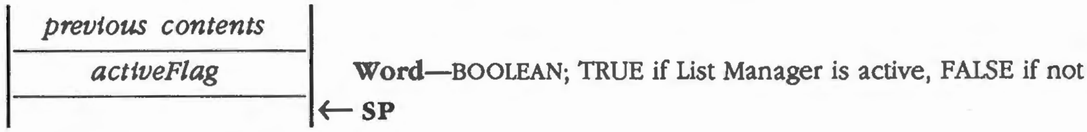
Indicates whether the List Manager is active.

Parameters

Stack before call



Stack after call


previous contents
activeFlag
Word—BOOLEAN; TRUE if List Manager is active, FALSE if not
← **SP**

Errors None

C

```
extern pascal Boolean ListStatus()
```

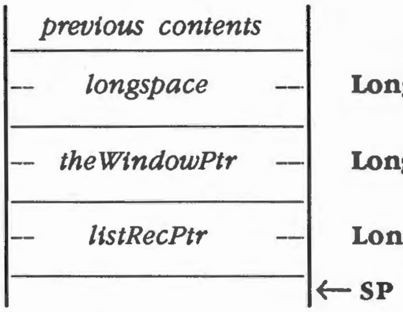
$091C CreateList

Calls the Control Manager routine NewControl to create a list control, using a specified list record. The routine also stores the list control's handle in the list record's *listCtl* field and passes the address of the List Manager's list control definition procedure. That definition procedure then creates the list control's vertical scroll bar and stores the scroll bar's handle in the list control's *ctlListBar* field.

To dispose of a list control, use the Control Manager routine KillControls or the Window Manager routine CloseWindow.

Parameters

Stack before call

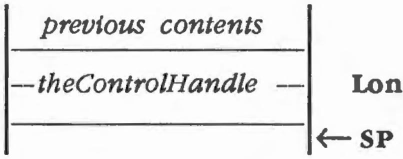


Long—Space for result

Long—POINTER to window in which the list should appear

Long—POINTER to list record

Stack after call



Long—HANDLE of list control

Errors

None

C

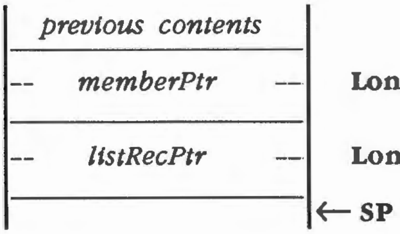
```
extern pascal ListCtlRecHndl CreateList(theWindowPtr,listRecPtr)

GrafPortPtr     theWindowPtr;

ListRecPtr     listRecPtr;
```

$0C1C DrawMember

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.

Parameters

Stack before call



Long—POINTER to member record in list to draw; NIL for all

Long—POINTER to list record

Stack after call

previous contents
← SP

Errors

None

C

```
extern pascal void DrawMember(memberPtr,listRecPtr)

MemRecPtr      memberPtr;

ListRecPtr      listRecPtr;
```

$0E1C GetListDefProc

Returns a pointer to the list control's definition procedure. Normally, you will not need to use this call.

❖ *Note:* The List Manager is basically a custom control; thus, you may need to access the control's definition procedure. See Chapter 4, "Control Manager," for information about control definition procedures.

Parameters

Stack before call

```
|  previous contents  |
|---------------------|
|--   longspace     --|   Long—Space for result
|---------------------|
|                     |← SP
```

Stack after call

```
|  previous contents  |
|---------------------|
|--   defProcPtr    --|   Long—POINTER to list control definition procedure
|---------------------|
|                     |← SP
```

Errors

None

C

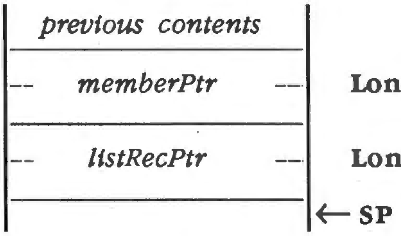
```
extern pascal LongProcPtr GetListDefProc()
```

$101C NewList

Resets the list control according to a specified list record. The routine uses the *listSize, listStart,* and *listPointer* fields of the list record pointed to by *listRecPtr* to reset the list control. The list control's scroll bar is also readjusted, and the list is redrawn with the new list and the member pointed to by *memberPtr* selected and in view.

Parameters

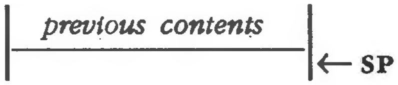
Stack before call



Long—POINTER to member to be selected

Long—POINTER to list record

Stack after call



Errors

None

C

```
extern pascal void NewList(memberPtr,listRecPtr)

MemRecPtr     memberPtr;

ListRecPtr     listRecPtr;
```

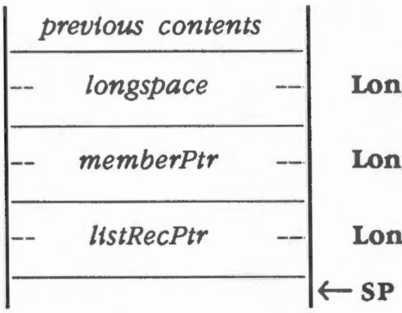
$0B1C NextMember

Searches a specified list record, starting with a specified member, and returns a pointer to the member record of the next selected member found.

A member is considered selected if the *memSelect* field (bits 7–6) of the member's *memFlag* field is set to 10. The NextMember routine does not change that bit.

Parameters

Stack before call

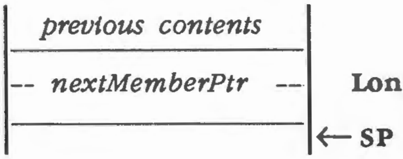


Long—Space for result

Long—POINTER to member at which to begin search; NIL for first

Long—POINTER to list record

Stack after call



Long—POINTER to member record of next selected member; NIL if no more

Errors

None

C

```
extern pascal memRecPtr NextMember(memberPtr,listRecPtr)

MemRecPtr     memberPtr;

ListRecPtr     listRecPtr;
```

$0F1C ResetMember

Searches a specified list record, starting with the first member, and returns a pointer to the member record of the first selected member found.

A member is considered selected if bit 7 of the member's *memFlag* field is set to 1. That bit will be cleared to 0 (the other bits of *memFlag* will not change) before ResetMember exits. However, the member is not redrawn in its new state (see the section "DrawMember" in this chapter).

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *listRecPtr* --	**Long**—POINTER to list record
	← **SP**

Stack after call

previous contents	
-- *nextMemberPtr* --	**Long**—POINTER to member record of first selected member; NIL if no more
	← **SP**

Errors

None

C

```
extern pascal memRecPtr ResetMember(listRecPtr)

ListRecPtr     listRecPtr;
```

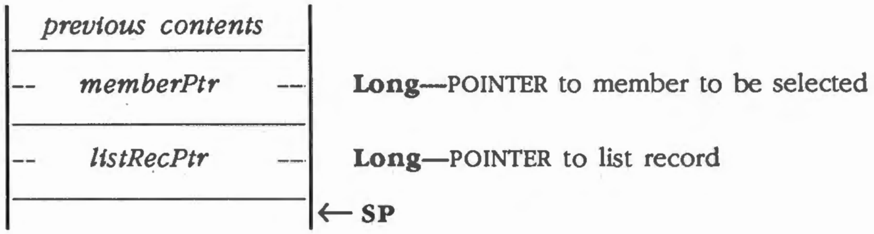
$0D1C SelectMember

Selects a specified member, deselects any other selected members in the list, and scrolls the list so that the specified member is at the top of the list display.

Important
The specified member is not selected if it is disabled.

Parameters

Stack before call



Long—POINTER to member to be selected

Long—POINTER to list record

Stack after call

previous contents
← SP

Errors

None

C

```
extern pascal void SelectMember(memberPtr,listRecPtr)

MemRecPtr      memberPtr;

ListRecPtr      listRecPtr;
```

$0A1C SortList

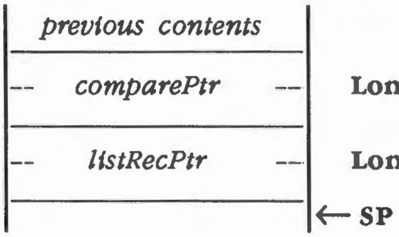
Alphabetizes a specified list by rearranging the array of member records.

If *comparePtr* is NIL, an internal string comparison routine is used that sorts standard alphabetical strings in ascending ASCII order. If your application needs a routine to sort members that are not standard strings, use *comparePtr* to point to that routine.

If SortList is called before the CreateList routine, the list will be drawn correctly when CreateList is called; no DrawMember call is necessary. If CreateList is called before SortList, DrawMember must be called to draw the list in its new order.

Parameters

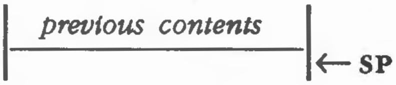
Stack before call



Long—POINTER to comparison routine; NIL for standard comparison

Long—POINTER to list record

Stack after call



Errors

None

C

```
extern pascal void SortList(comparePtr,listRecPtr)

VoidProcPtr     comparePtr;

ListRecPtr     listRecPtr;
```

(continued)

Custom comparison routine example

An example of a custom comparison routine follows.

```
;   IN: memberA = pointer to member A
;       memberB = pointer to member B
;
Compare START
;
memberA equ    4
memberB equu   memberA +4
;       (Your comparison code would go here. How the routine does the
;        comparison is up to you.  For example:)
;       if memberA >= memberB
;              sec                    Carry set if memberA is greater
;                                     than or equal to memberB
;       else
               clc                    Carry clear if memberA is less than memberB
;
        lda    0,s    Remove input parameters and return
        sta    8,s
        lda    2,s
        sta    10,s
        pla
        pla
        pla
        pla
        rtl
```

List Manager summary

This section briefly summarizes the constants and data structures contained in the List Manager. There are no tool set error codes for the List Manager.

Important

These definitions are provided in the appropriate interface file.

Table 11-3
List Manager constants

Name	Value	Description
Bit mask for listType		
`cString`	$0001	Null terminated string type
`selectOnlyOne`	$0002	Only one selection allowed
memFlag		
`memDisabled`	$40	Sets member flag to disabled
`memSelected`	$80	Sets member flag to selected

Table 11-4
List Manager data structures

Name	Offset	Type	Definition
List record (ListRec)			
listRect	$0	Rect	Enclosing rectangle
listSize	$8	Word	Number of members in the list
listView	$A	Word	Maximum number of members visible to the user at one time
listType	$C	Word	Bit flag
listStart	$E	Word	First member in view
listCtl	$10	CtlRecHndl	List control's handle
listDraw	$14	VoidProcPtr	Pointer to custom drawing routine
listMemHeight	$18	Word	Height of list members
listMemSize	$1A	Word	Size of member records
listPointer	$1C	MemRecPtr	Pointer to first element in MemRec
listRefCon	$20	Long	Becomes control's refCon
listScrollClr	$24	BarColorsPtr	Color table for list's scroll bar

(continued)

Table 11-4 (continued)
List Manager data structures

Name	Offset	Type	Definition
List control record (ListCtlRec)*			
ctlMemDraw	$28	VoidProcPtr	Pointer to routine to draw member
ctlMemHeight	$2C	Word	Member's height in pixels
ctlMemSize	$2E	Word	Bytes in member record
ctlList	$30	MemRecPtr	Pointer to first member record in array
ctlListBar	$34	CtlRecHndl	Handle to list control's scroll bar control
Member record (MemRec)			
memPtr	$0	Pointer	Pointer to string
memFlag	$4	Byte	Bit flag defining selected or disabled status
List color table (LColorTable)			
listFrameClr	$00	Word	Frame color
listNorTextClr	$2	Word	Unhighlighted text color
listSelTextClr	$4	Word	Highlighted text color
listNorBackClr	$6	Word	Unhighlighted background color
listSelBackClr	$8	Word	Highlighted background color

Note: The actual assembly-language equates have a lowercase *o* (the letter) in front of all of the names given in this table.

* The first few fields of the list control record are the same as those of a control record. Those fields are followed by the listed fields.

Chapter 12

Memory Manager

The **Memory Manager** controls the allocation, deallocation, and repositioning of memory blocks. The manager keeps track of how much memory is free and what parts are allocated to whom. Memory is allocated in **blocks** of arbitrary length; each block possesses several attributes that describe how the Memory Manager may modify it (such as by moving it or deleting it), how it must be aligned in memory (for example, on a page boundary), and what program owns it.

The Memory Manager works closely with ProDOS 16 and the System Loader to provide the needed memory spaces for loading programs and data and for providing buffers for input and output. All Apple IIGS software, including the System Loader and ProDOS 16, must obtain needed memory space by making Memory Manager calls.

When an application makes a ProDOS 16 call that requires allocation of memory (such as opening a file or writing from a file to a memory location), ProDOS 16 first obtains any needed memory blocks from the Memory Manager and then performs its tasks. Likewise, the System Loader requests any needed memory either directly or indirectly (through ProDOS 16 calls) from the Memory Manager.

Conversely, when an application informs the operating system that it no longer needs certain blocks of memory, that information is passed to the Memory Manager, which in turn frees those blocks. In all of these cases memory allocation and deallocation is completely automatic, as far as the application is concerned.

For more information about how the Memory Manager communicates with ProDOS 16 and the System Loader, see the *Apple IIGS ProDOS 16 Reference.*

❖ *Macintosh programmers:* The Apple IIGS Memory Manager is conceptually very similar to the Macintosh Memory Manager. However, because of the 65C816 microprocessor and the architecture of the Apple IIGS, its calls are very different, and its internal data structures are totally different than those of the Macintosh.

A preview of the Memory Manager routines

To introduce you to the capabilities of the Memory Manager, all Memory Manager routines are grouped by function and briefly described in Table 12-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the Memory Manager routines (discussed in alphabetical order).

Table 12-1
Memory Manager routines and their functions

Routine	Description
Housekeeping routines	
MMBootInit	Called only by the Tool Locator when the Memory Manager is initialized—must not be made by an application
MMStartUp	Starts up the Memory Manager for use by an application
MMShutDown	Shuts down the Memory Manager when the application quits
MMVersion	Returns the version number of the Memory Manager
MMReset	Called only when the system is Reset—must not be made by an application
MMStatus	Indicates whether or not the Memory Manager is active
Memory allocation routines	
NewHandle	Creates a new block and returns the handle to the block
ReAllocHandle	Reallocates a purged block using new attributes
RestoreHandle	Reallocates a purged block using the same attributes, user ID, and size that were associated with the purged handle
DisposeHandle	Discards a specified block and deallocates its handle
DisposeAll	Discards all of the handles and blocks belonging to a specified user ID
PurgeHandle	Purges a specified purgeable, unlocked block, but does not deallocate the handle
PurgeAll	Purges all of the purgeable, unlocked blocks for a specified user ID
Block information and free space routines	
FindHandle	Returns the handle of a block containing a specified address
CheckHandle	Checks a specified handle to see whether it is valid
GetHandleSize	Returns the size of a block
SetHandleSize	Changes the size of a specified block
CompactMem	Compacts memory space
FreeMem	Returns the total number of free bytes in memory
MaxBlock	Returns the size of the largest free block in memory, not counting memory that can be freed by purging or compacting
TotalMem	Returns the size of all memory, including the main 256K

Table 12-1 (continued)
Memory Manager routines and their functions

Routine	Description
Locking and purge-level routines	
HLock	Locks a specified block
HLockAll	Locks all of the blocks belonging to a specified user ID
HUnlock	Unlocks a specified block
HUnlockAll	Unlocks all of the blocks belonging to a specified user ID
SetPurge	Sets the purge level of a specified block
SetPurgeAll	Sets the purge level of all blocks belonging to a specified user ID
Other routines	
BlockMove	Copies a specified number of bytes from a source to a destination
PtrToHand	Copies a specified number of bytes from a source to a destination, with the source specified by a pointer and the destination specified by a handle
HandToPtr	Copies a specified number of bytes from a source to a destination, with the source specified by a handle and the destination specified by a pointer
HandToHand	Copies a specified number of bytes from a source to a destination, with the source specified by a handle and the destination specified by a handle

Apple IIGS memory map

The memory in the Apple IIGS is divided into three categories, as follows:

- **Nonspecial or normal memory:** memory that has no special restrictions on it. This memory includes banks $2–$DF and parts of banks $E0 and $E1.
- **Special memory:** memory that has restrictions on its use because it is memory that appears in the Apple IIe. Special memory may not be used by desk accessories, tool sets, and other routines that might be called while running old applications. This memory includes banks $0–$1 and parts of banks $E0 and $E1.
- **Reserved memory:** memory not managed by the Memory Manager. This includes the language cards, addresses $0000–$0800 in banks $0 and $1, and addresses $0000–$2000 in banks $E0 and $E1. This memory is marked as *busy* in the Memory Manager at startup time.

Figure 12-1 shows which parts of memory can be allocated through requests to the Memory Manager.

❖ *Note:* At the time of publication, the Apple IIGS can address up to 8 megabytes, which means that normal memory contains only banks $2–$7F and parts of banks $E0 and $E1. In the future, this mayexpand; that expansion will be transparent to your application if you follow the rules for using the Memory Manager.

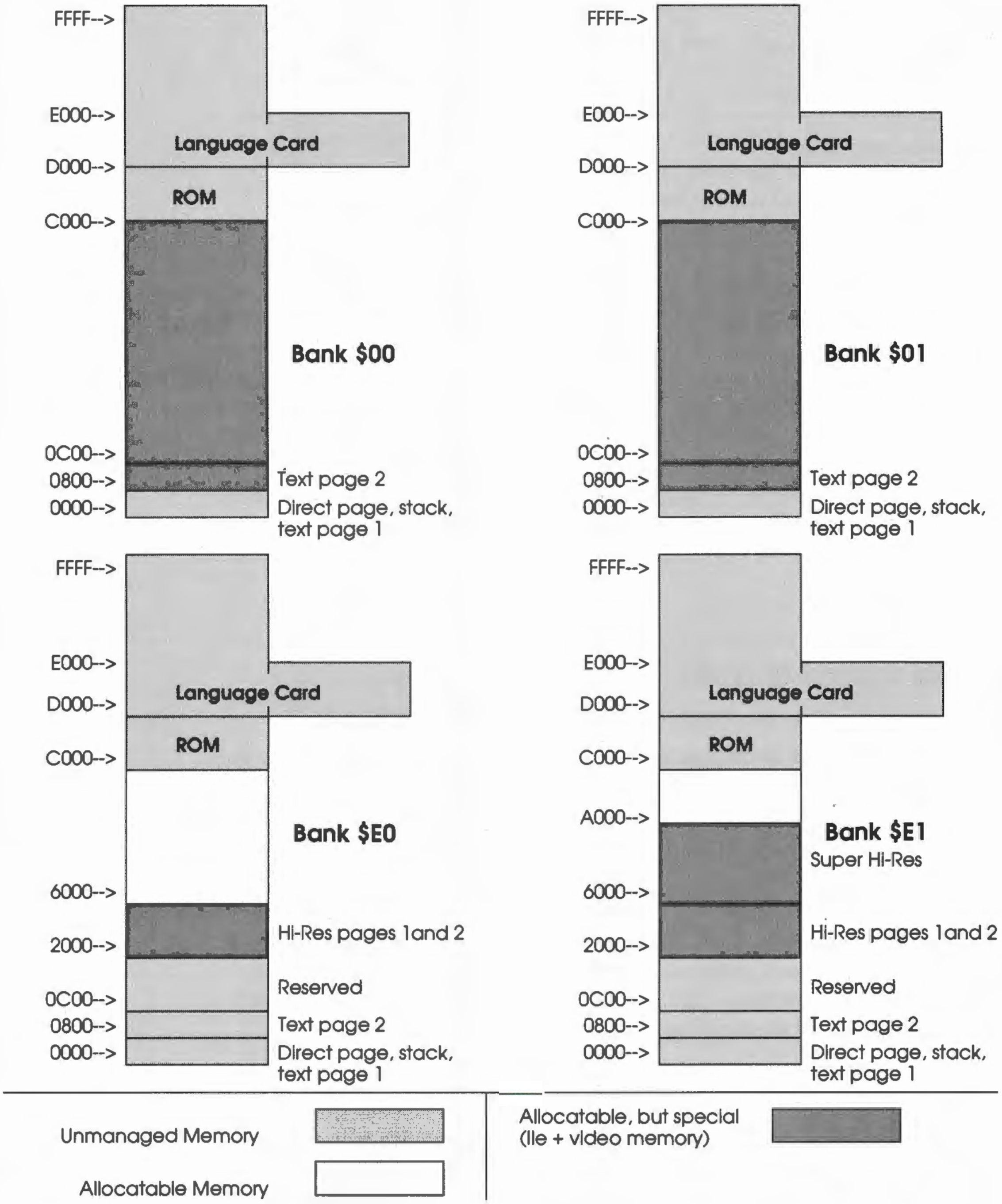


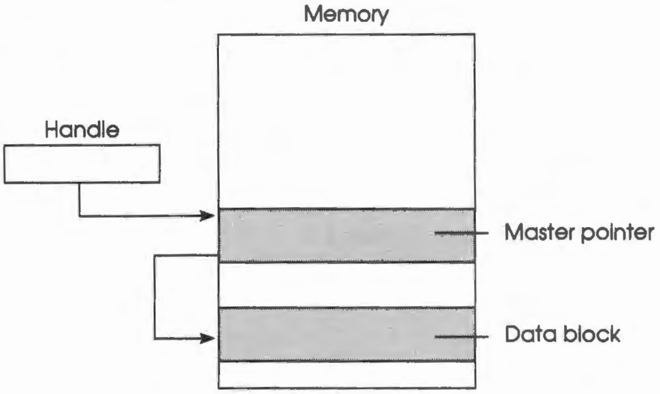
Figure 12-1
Memory Manager memory use

For more information about the memory layout, see the *Apple IIGS Hardware Reference*.

Pointers and handles

Because the Memory Manager on the Apple IIGS can move memory blocks, an application cannot use a simple pointer to access the blocks. Instead, each time the Memory Manager allocates a memory block, it returns to the requesting application a handle referencing that block.

A **handle** is a pointer to a **master pointer;** it is the address of a fixed (immovable) location that contains the address of the block. If the Memory Manager changes the location of the block, it updates the address in the fixed location; the value of the handle itself is not changed. Thus, the application can use the handle to access the block, no matter how often the block itself is moved in memory. Figure 12-2 illustrates a handle.



Handle points to master pointer, which points to actual block.

Figure 12-2
Memory handle

If a segment will always be fixed in memory (locked or nonmovable), it may be referenced by a pointer instead of a handle. To obtain a pointer to a particular block or location, an application can **dereference** the block's handle. The application reads the address stored in the location pointed to by the handle; that address is the pointer to the block. Of course, if the Memory Manager moves the block, the pointer is no longer valid.

Important

After a handle has been dereferenced, certain system calls or toolbox calls might cause memory compaction, which would then make the pointer to the block invalid. In that case, your application must dereference the handle again. However, compaction won't occur unexpectedly; that is, toolbox calls made from interrupt handlers cannot force compaction. This means that unlocked blocks can safely be dereferenced so long as no system or toolbox calls are made between the dereference and the use of the handle.

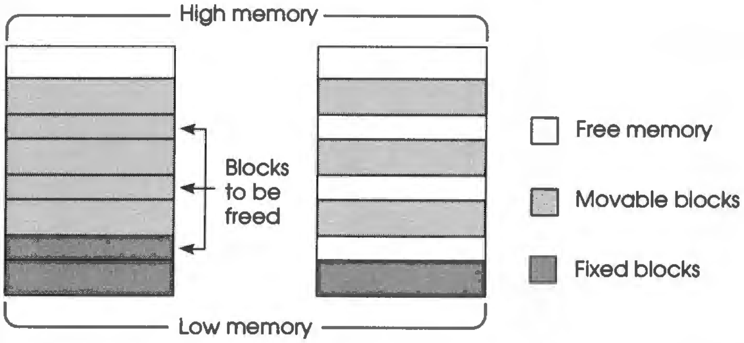
Pointers and handles must be at least 3 bytes long to access the full range of Apple IIGS memory. However, all pointers and handles used by the Apple IIGS must be 4 bytes long for ease of manipulation by the 16-bit registers in the 65C816 microprocessor.

Important

Because the address space of the IIGS requires only 24 bits, the high-order byte of the 32-bit address is always $0. Do not attempt to store other information in that high-order byte! If you do, the Memory Manager and other tool sets may not work properly.

Memory fragmentation and compaction

Memory blocks can be allocated and deallocated in any order, so memory tends to become fragmented into a jumble of free and allocated memory blocks. When **fragmentation** happens, the Memory Manager may not be able to allocate a requested block, even if there is enough free memory available, because the space is broken into small, isolated blocks. Figure 12-3 illustrates fragmented memory.



1. Before deallocation

2. After deallocation—fragmented memory

Figure 12-3
Memory fragmentation

When the Memory Manager is unable to allocate a block, it tries to compact memory. **Compaction** moves all movable blocks to consolidate free space into a single region in memory. Figure 12-4 illustrates memory compaction.

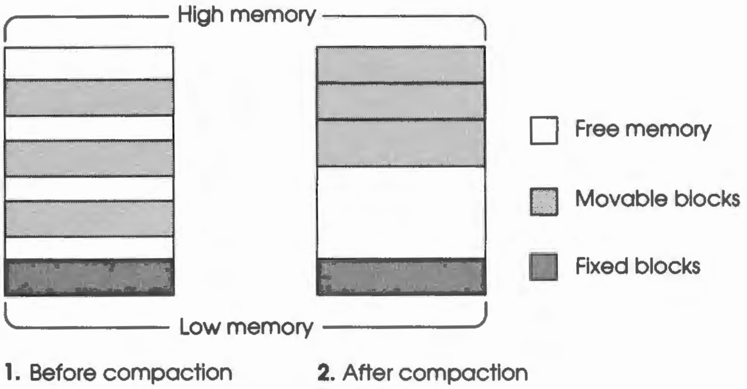


Figure 12-4
Memory compaction

Fixed blocks and locked movable blocks interfere with compaction by forming immovable islands in memory. Such blocks can prevent the Memory Manager from collecting together the free blocks and can leave memory fragmented after compaction, as shown in Figure 12-5.

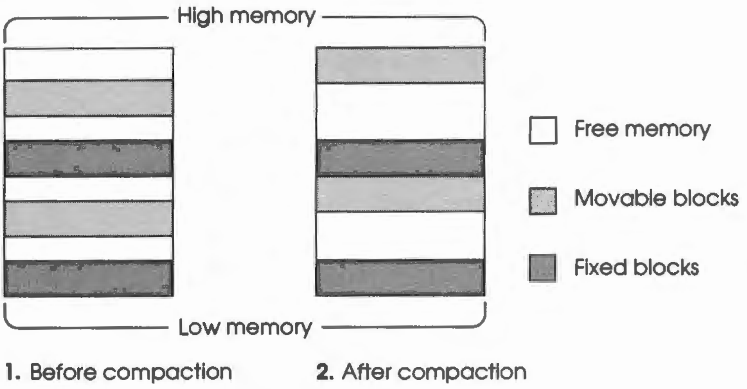


Figure 12-5
Fragmentation after compaction

The Memory Manager never moves a movable block around a nonmovable one. To minimize this problem, the Memory Manager tries to allocate fixed blocks toward the bottom of memory and movable blocks toward the top of memory.

Important

To help prevent fragmentation, your application should use movable blocks whenever possible and leave blocks unlocked whenever possible.

Purging memory

If the Memory Manager is still unable to allocate a block after compacting memory, it tries to **purge** blocks.

❖ *Note:* Only blocks that are marked *purgeable* and *unlocked* can be purged. See the section "Assigning Memory Block Attributes" later in this chapter for more information.

Purging throws out the contents of the block and frees it. The block's master pointer remains allocated, and its value is set to NIL.

❖ *Note:* **NIL** is equivalent to 0 (zero).

A handle pointing to a NIL master pointer is called an **empty handle.** If you want your application to refer to the purged block, the application must detect that the handle is empty and ask the Memory Manager to reallocate the block.

Important

Even if the block is reallocated, the data in a purged block has been lost and must be restored by your application.

Figure 12-6 shows a block being purged and reallocated.

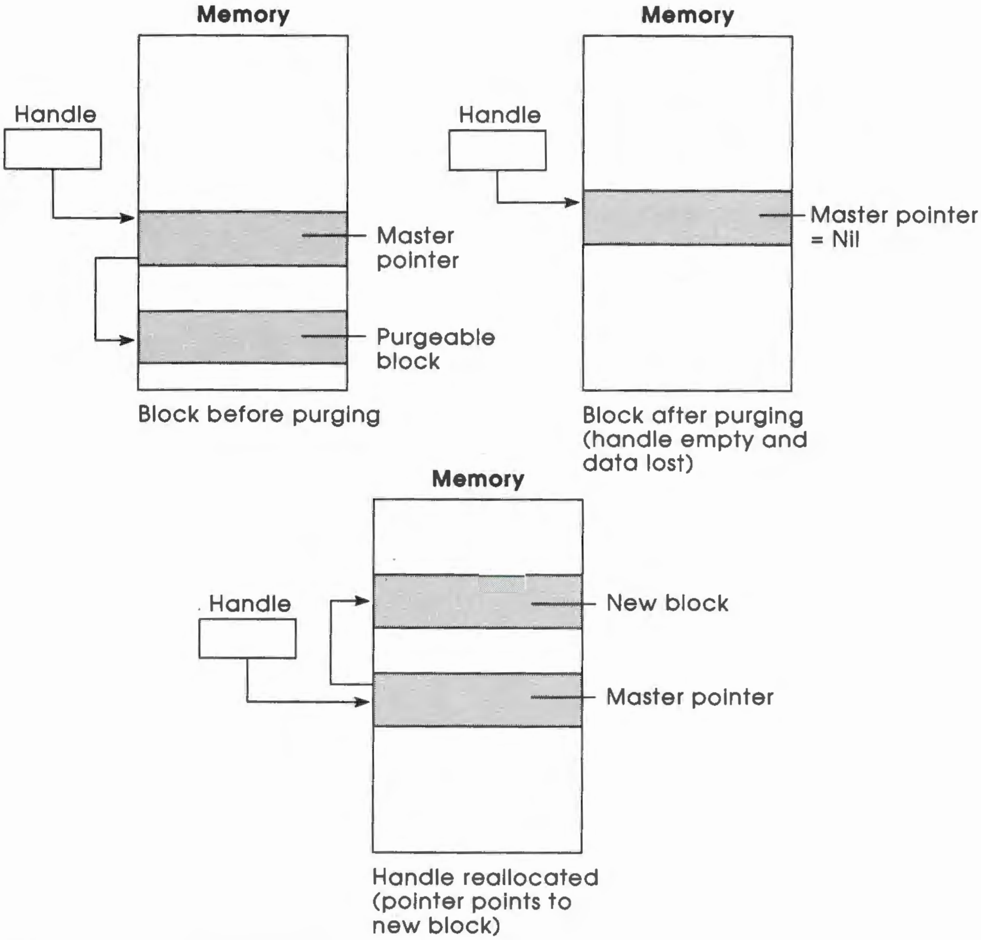


Figure 12-6
Memory block being purged and reallocated

When the Memory Manager runs out of memory, it starts purging purgeable blocks in an attempt to make more room. The order of the purging is based on the purge level of the block. The **purge level** is a 2-bit number in the attributes word specifying the purging priority of the block. The values are as follows:

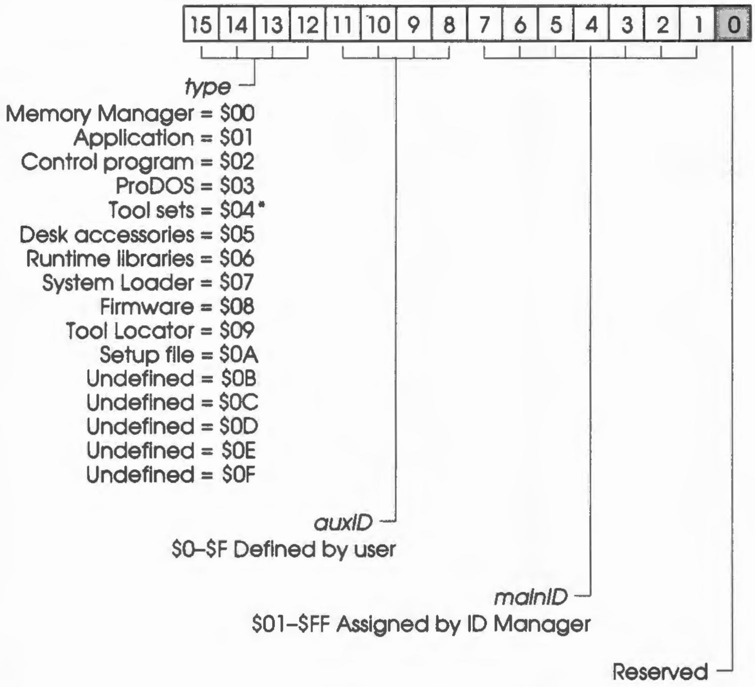
3	Most purgeable; used by the System Loader
2	Next most purgeable
1	Least purgeable
0	Not purgeable

Your application should use only levels 0–2; level 3 is reserved for the System Loader. When some applications exit, the memory is not freed, but all of the application's blocks are set to level 3. The old application can thus be restarted without accessing the disk if the new application did not need the space. If the Memory Manager purges any blocks of an application exited this way, it will purge all of them.

User IDs

When you start up the Memory Manager for use by your application, the operating system has already assigned a master user ID for that execution of the application. The Memory Manager uses this **master user ID** to reserve the memory it needs for the program's code and static data. The operating system gives the master user ID number to the Memory Manager, which in turn passes that ID to the application in the MMStartUp call. You must save that ID for use when you shut down the application.

User IDs are made up of three fields—*type, auxID,* and *mainID*—contained in a word parameter, as shown in Figure 12-7. The value in the *mainID* field is assigned by the **User ID Manager** and will always be a nonzero value. You must provide the value for the *type* field, which has fixed assignments as shown in Figure 12-7.



* Apple controls assignment of IDs in this class. At the time of publication, the only assignments were $41xx = Miscellaneous Tool Set and $42xx = Scrap Manager.

Figure 12-7
User ID fields

For your application, you will be most interested in the *auxID* field; you can use it to allocate any additional, private memory that your application needs. Since the field's valid values are from $1 to $F, you can create up to 15 new and distinct user IDs, each of which can be used to allocate memory in a NewHandle call.

Important

Do not use an *auxID* of 0. The Memory Manager routines PurgeAll and DisposeAll treat an *auxID* field with 0 in it as a wildcard that matches all values.

When your application is through using all of the memory for a particular *auxID,* it can discard the ID all at once by calling DisposeAll with the fully specified ID. An example of this technique is shown in the following assembly-language code fragment:

```
pushword #0           ; Space for master user ID

_MMStartUp

pla                   ; Retrieve master user ID

sta MasterID          ; Store master user ID

ora #$0100            ; Create user ID with auxID = 1

sta MyID              ; Store ID for use with private memory

....

; (Your code here)

...

; (Ready to exit program)

pushword MyID

_DisposeAll           ; Discard all of my private memory

; (Continue with termination processing)
```

Using this technique, you don't have to deallocate the new user ID you obtained by modifying the *auxID* field. When the master user ID is deallocated when the application quits, any derivatives of the master ID are automatically deleted.

There are two other methods of assigning the user ID, although we recommend that you modify the *auxID*. The options and their disadvantages are:

1. Simply use the master user ID to obtain new private memory. This is the simplest method; however, your application must then individually discard any blocks it obtains, rather than using a DisposeAll call to discard them all at once.
2. Obtain an entirely new user ID by calling the Miscellaneous Tool Set GetNewID routine. This method allows you to discard all private memory at once; however, you must be sure that your application discards the memory and deletes the new user ID before the application quits.

Assigning memory block attributes

Your application controls the properties of a particular block by specifying values in a **memory attributes** word. This attribute word determines how the blocks are allocated and maintained. When a block is allocated, several of the bits in the attributes word determine how the block is allocated. These attributes can be set only when the block is allocated. However, the attributes that determine whether a block can be moved or purged (*attrLocked* and *attrPurge*) can be changed by your application after a block is created. Figure 12-8 illustrates the memory attributes word. In all of the bits except bits 9–8, a value of 1 means that the attribute applies to the block; you might think of setting the bit to 1 as applying a restriction to the block.

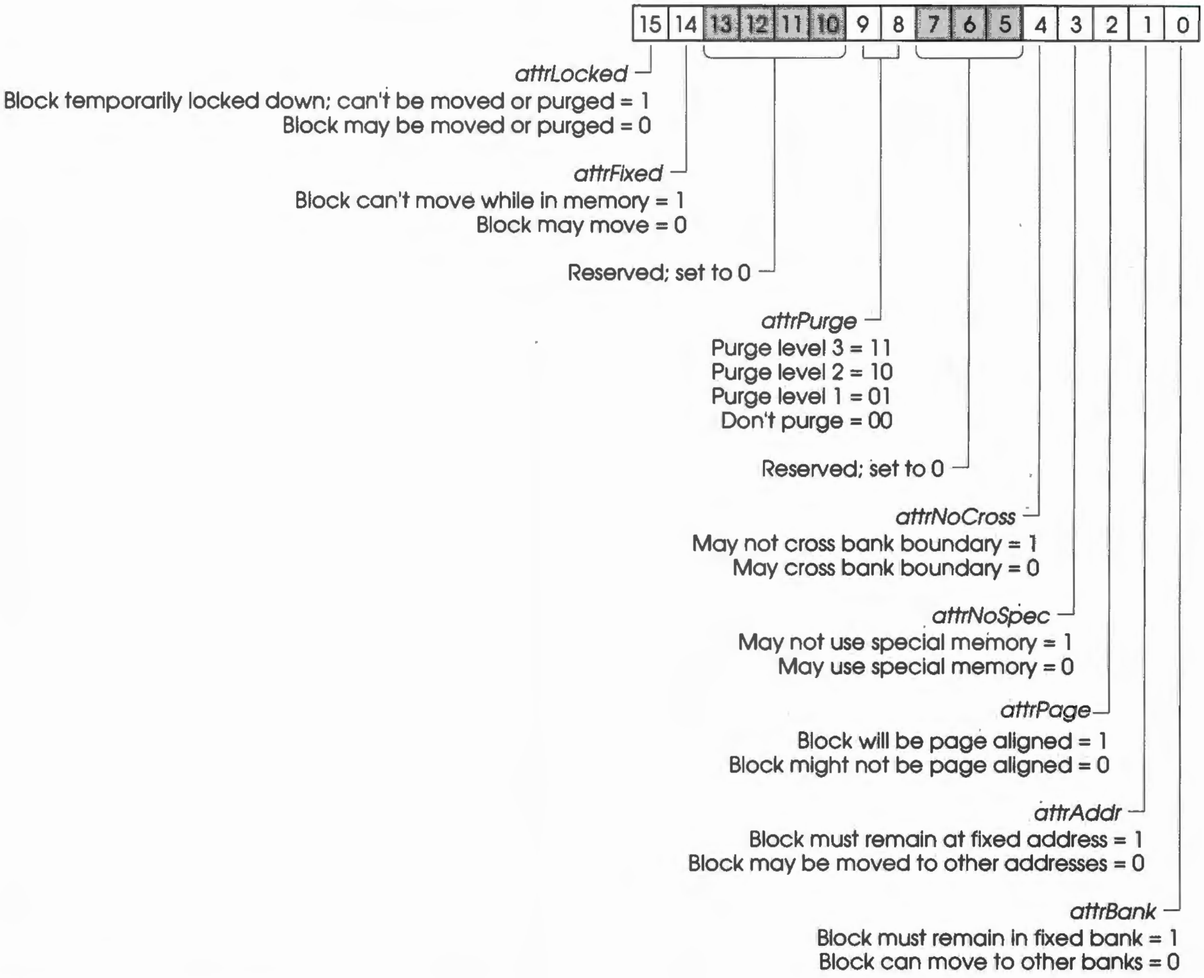

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
attrLocked
Block temporarily locked down; can't be moved or purged = 1
Block may be moved or purged = 0
attrFixed
Block can't move while in memory = 1
Block may move = 0
Reserved; set to 0
attrPurge
Purge level 3 = 11
Purge level 2 = 10
Purge level 1 = 01
Don't purge = 00
Reserved; set to 0
attrNoCross
May not cross bank boundary = 1
May cross bank boundary = 0
attrNoSpec
May not use special memory = 1
May use special memory = 0
attrPage
Block will be page aligned = 1
Block might not be page aligned = 0
attrAddr
Block must remain at fixed address = 1
Block may be moved to other addresses = 0
attrBank
Block must remain in fixed bank = 1
Block can move to other banks = 0

Figure 12-8
Memory attributes flag

Table 12-2 provides the attribute information in a different format.

Table 12-2
Memory block attributes

Bit	Constant	Attribute	Description
15	`attrLocked`	Locked	If the block is locked, it is immovable and unpurgeable, regardless of the fixed or purge-level attributes. A block can thus be temporarily locked in place while it is being executed or referenced. This attribute can be changed after allocation.
14	`attrFixed`	Fixed	If a block is fixed, it cannot be moved when compacting memory. Code blocks should usually be fixed, but data blocks should usually not be fixed.
9–8	`attrPurge`	Purge level	This 2-bit number defines the purge priority of a block: 0 means the block cannot be purged, 3 means the block will be the first block purged. This attribute can be changed after allocation. See the section "Purging Memory" in this chapter for more information.
4	`attrNoCross`	Bank-boundary limited	Specifies that a block must not cross banks. Code blocks, for example, may never cross banks.
3	`attrNoSpec`	Special memory not usable	Specifies that the block may not be allocated in special memory. This is memory that was used in the Apple IIe. It includes banks $00 and $01 and the video screens.
2	`attrPage`	Page aligned	Specifies that the block must start on a page; that is, the block must start at a location that is a multiple of $100. For timing reasons, code or data may need to be page aligned; for example, the direct-page space for the managers and tool sets should be page aligned.
1	`attrAddr`	Fixed address	Specifies that the block must be at a specified address when allocated. An example is allocating the graphics screen.
0	`attrBank`	Fixed bank	Specifies that the block must start in a specified bank. An example is allocating a block to be used as a direct page.

Cleaning up memory

When your application quits, it must explicitly dispose of any memory that it acquired; if it doesn't, the memory management system can become clogged.

If, as recommended, you modified the *auxID* field of the master user ID to create a unique user ID, you can simply use the DisposeAll routine to dispose of the memory all at once for that specific user ID.

If you used the master user ID to allocate private memory, you must dispose of the private memory on a handle-by-handle basis. In this case, you can't use the DisposeAll routine to dispose of your private memory all at once, since that would also mark the application's code space as available for reallocation.

If you asked the User ID Manager for an entirely new ID, you can use the DisposeAll routine to dispose of the memory all at once for that specific user ID.

For more information about memory management, see the *Programmer's Introduction to the Apple IIGS* and the *Apple IIGS ProDOS 16 Reference*.

Using the Memory Manager

This section discusses how the Memory Manager routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

The Memory Manager depends upon the presence of the tool set shown in Table 12-3 and requires that at least the indicated version of the tool set be present.

Table 12-3
Memory Manager—other tool sets required

Tool set number	Tool set name	Minimum version needed
$01 #01	Tool Locator	1.0

The first Memory Manager call that your application must make is MMStartUp. Conversely, when you quit your application, you must make the MMShutDown call.

Remember that the Memory Manager works closely with ProDOS 16 and the System Loader to provide the needed memory spaces for initally loading your application and its data. Thus, your application's use of the Memory Manager, after you have started it up, will be to ask it for more memory for private purposes.

To allocate memory for the application's own purposes, you must use one or more Memory Manager NewHandle calls and assign to the memory blocks you acquire the appropriate user ID and allocation attributes. After you have acquired blocks of memory, you can temporarily lock a single block by using the HLock routine or lock all of the blocks associated with a particular user ID by using the HLockAll routine.

When your application no longer needs the blocks to be locked, you should unlock them with the HUnlock or HUnlockAll routine, so that the Memory Manager may move them during compaction. If you are temporarily through using the blocks, you can set their purge levels by using the SetPurge or SetPurgeAll routine. If the Memory Manager needs more space, it can then purge those blocks.

If the Memory Manager does purge the blocks, you can restore the block with the same attributes, user ID, and size by using the RestoreHandle routine, or you can reallocate the block with new characteristics by using the ReAllocHandle routine.

Important
If the Memory Manager purges the blocks, the data in them is lost. Your application has the responsibility to save and restore the data appropriately.

When your application is completely done with its own private memory, it should call the DisposeAll routine and specify the appropriate user ID (usually an ID with a modified *auxID* field) to discard all of that memory.

Important
Do not call DisposeAll with the unmodified master user ID.

$0102 MMBootInit

Initializes the Memory Manager; called only by the Tool Locator.

Warning

An application must never make this call, because it will destroy all currently allocated blocks.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$0202 MMStartUp

Starts up the Memory Manager for use by an application.

Important

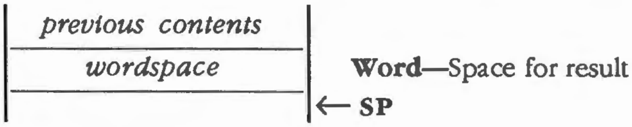
Your application must make this call before it makes any other Memory Manager calls.

You must save the value returned as *userID* to use when your application quits.

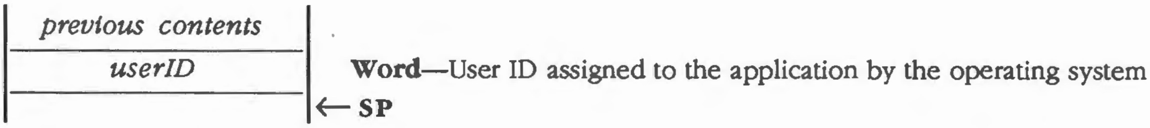
❖ *Note:* If you're writing an application for operating systems such as ProDOS 8 or DOS 3.3 and the call is not made from a valid block, an ID error occurs. If this happens, your application should call the User ID Manager for an ID number and then call the Memory Manager to allocate the application's program segments. For more information about the User ID Manager, see the section "User IDs" in this chapter.

Parameters

Stack before call



Stack after call


previous contents
userID
Word—User ID assigned to the application by the operating system
← **SP**

Errors

$0207	`idErr`	Invalid user ID

C

```
extern pascal Word MMStartUp()
```

$0302 MMShutDown

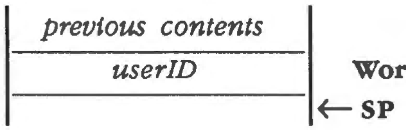
Shuts down the Memory Manager when an application quits. The *userID* must be the same as the one returned by the Memory Manager in the MMStartUp call.

Important

If your application has started up the Memory Manager, the application must make this call before it quits. Also, your application must discard all memory blocks it has allocated before it makes the MMShutDown call. See the sections "Cleaning Up Memory" and "Using the Memory Manager" in this chapter for more information.

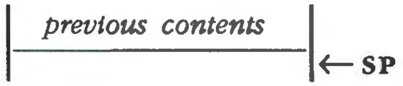
Parameters

Stack before call



Word—User ID of the application to be shut down

Stack after call



Errors

None

C

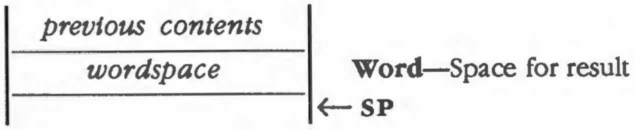
```
extern pascal void MMShutDown(userID)

Word      userID;
```

$0402 MMVersion

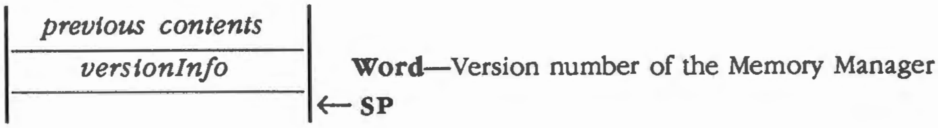
Returns the version number of the Memory Manager.

Parameters

Stack before call



Stack after call



Errors

None

C

```
extern pascal Word MMVersion()
```

$0502 MMReset

Reset the Memory Manager; called 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

$0201	`memErr`	Unable to reset

C

Call must not be made by an application.

$0602 MMStatus

Indicates whether the Memory Manager is active.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
activeFlag	**Word**—BOOLEAN; TRUE if Memory Manager active, FALSE if inactive
	← **SP**

Errors

None

C

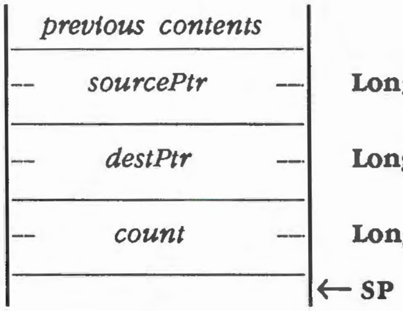
```
extern pascal Boolean MMStatus()
```

$2B02 BlockMove

Copies a specified number of bytes from a source to a destination. BlockMove works correctly even if source and destination overlap or cross bank boundaries. No address validation is performed; BlockMove simply overwrites everything.

Parameters

Stack before call

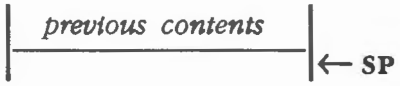


Long—POINTER to start of bytes to be copied

Long—POINTER to starting address where bytes will be written

Long—Number of bytes to be copied

Stack after call



Errors

None

C

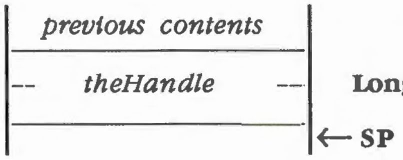
```
extern pascal void BlockMove(sourcePtr,destPtr,count)

Pointer     sourcePtr;

Pointer     destPtr;

LongWord     count;
```

$1E02 CheckHandle

Checks a handle to see whether it is valid. If the Memory Manager does not recognize *theHandle*, an error occurs. The routine is intended primarily as a debugging aid.

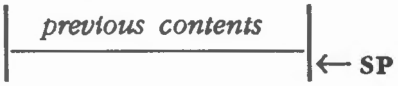
Parameters

Stack before call



Long—HANDLE whose validity will be checked

Stack after call



Errors

$0206	`handleErr`	Invalid handle

C

```
extern pascal void CheckHandle(theHandle)

Handle      theHandle;
```

$1F02 CompactMem

Compacts memory space.

❖ *Note:* CompactMem has no effect if it is called during an interrupt. Thus, if you're writing an interrupt handler, you can't use CompactMem to force the Memory Manager to compact memory space.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

None

C

```
extern pascal void CompactMem()
```

$1102 DisposeAll

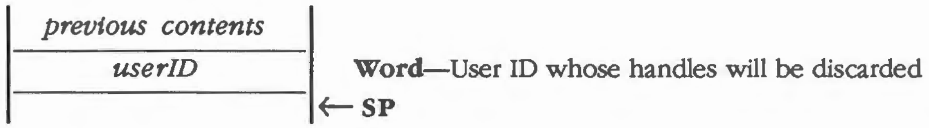
Discards all of the handles and blocks belonging to a specified user ID.

Important

Your application must discard all memory blocks it has allocated for itself before making the MMShutDown call. However, if you used the unmodified master user ID to allocate your own private memory, you must call DisposeHandle to deallocate that memory handle by handle. If you use DisposeAll in that instance, you will also deallocate the very space your application is using for execution. See the section "Using the Memory Manager" in this chapter for more details.

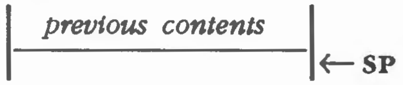
Parameters

Stack before call



Word—User ID whose handles will be discarded

Stack after call



Errors

$0207	`idErr`	Invalid user ID

C

```
extern pascal void DisposeAll(userID)

Word      userID;
```

$1002 DisposeHandle

Discards a specified block and deallocates its handle. The block is purged regardless of locked status and purge level.

Parameters

Stack before call

```
|  previous contents  |
|---------------------|
|--    theHandle    --|   Long—HANDLE of the block to be discarded
|---------------------|
|                     |← SP
```

Stack after call

```
|  previous contents  |
|---------------------|
|                     |← SP
```

Errors

$0206 `handleErr` Invalid handle

C

```
extern pascal void DisposeHandle(theHandle)

Handle      theHandle;
```

$1A02 FindHandle

Returns the handle of the block containing a specified address. If the block is not locked, it may later move. If *locationPtr* is not in any handle, then NIL is returned.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *locationPtr* --	**Long**—POINTER whose handle will be found
	← **SP**

Stack after call

previous contents	
-- *theHandle* --	**Long**—HANDLE of *locationPtr;* NIL if not found
	← **SP**

Errors

None

C

```
extern pascal Handle FindHandle(locationPtr)

Pointer     locationPtr;
```

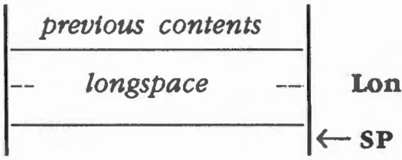
$1B02 FreeMem

Returns the total number of free bytes in memory, not counting memory that can be freed by purging. However, because of memory fragmentation, it might not be possible to allocate a block as large as the size indicated by this routine.

❖ *Note:* Use the MaxBlock routine to find the size of the largest block that can be allocated.

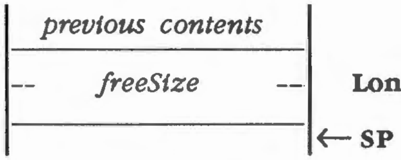
Parameters

Stack before call



Long—Space for result

Stack after call



Long—Total number of free bytes in memory

Errors

None

C

```
extern pascal LongWord FreeMem()
```

$1802 GetHandleSize

Returns the size of a specified block in bytes.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *theHandle* --	**Long**—HANDLE of block whose size is to be retrieved
	← **SP**

Stack after call

previous contents	
-- *blockSize* --	**Long**—Size of block, in bytes
	← **SP**

Errors

$0206	`handleErr`	Invalid handle

C

```
extern pascal LongWord GetHandleSize(theHandle)
Handle     theHandle;
```

$2A02 HandToHand

Copies a specified number of bytes from a source to a destination, using handles. HandToHand works correctly even if source and destination overlap or cross bank boundaries. No address validation is performed; HandToHand simply overwrites everything.

Parameters

Stack before call

previous contents	
— *sourceHandle* —	**Long**—HANDLE specifying start of bytes to be copied
— *destHandle* —	**Long**—HANDLE specifying starting address where bytes will be written
— *count* —	**Long**—Number of bytes to be copied
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0202	`emptyErr`	Illegal operation on an empty handle
$0206	`handleErr`	Invalid handle

C

```
extern pascal void HandToHand(sourceHandle,destHandle,count)

Handle      sourceHandle;

Handle      destHandle;

LongWord      count;
```

$2902 HandToPtr

Copies a specified number of bytes from a source to a destination, with the source specified by a handle and the destination specified by a pointer. HandToPtr works correctly even if source and destination overlap or cross bank boundaries. No address validation is performed; HandToPtr simply overwrites everything.

Parameters

Stack before call

previous contents	
sourceHandle	**Long**—HANDLE specifying start of bytes to be copied
destPtr	**Long**—POINTER to starting address where bytes will be written
count	**Long**—Number of bytes to be copied
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0202	`emptyErr`	Illegal operation on an empty handle
$0206	`handleErr`	Invalid handle

C

```
extern pascal void HandToPtr(sourceHandle,destPtr,count)
Handle      sourceHandle;
Pointer      destPtr;
LongWord      count;
```

$2002 HLock

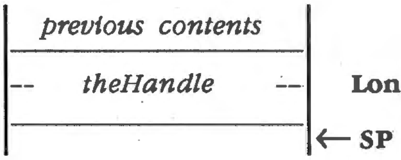
Locks a block specified by a handle. A locked block cannot be relocated or purged during memory compaction.

❖ *Note:* If you need to quickly lock a block, you can directly set bit 15 of the attributes word to 1. Apple guarantees that the locked attribute will stay at that location. In 65C816 assembly language, you could set the bit as in the following fragment:

```
LDY      #4
LDA      [myhandle],y
ORA      #AttrLocked
STA      [myhandle],y
```

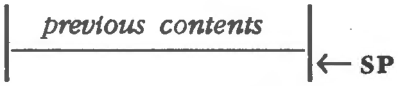
Parameters

Stack before call



Long—HANDLE of block to be locked

Stack after call



Errors

$0206	`handleErr`	Invalid handle

C

```
extern pascal void HLock(theHandle)
Handle     theHandle;
```

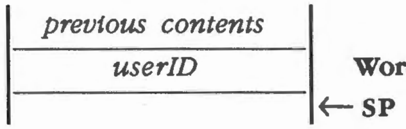
$2102 HLockAll

Locks all of the blocks belonging to a specified user ID. Locked blocks cannot be moved or purged during memory compaction.

❖ *Note:* See the section “User IDs” in this chapter for more information about user IDs.

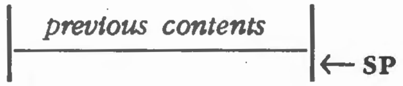
Parameters

Stack before call



Word—User ID whose blocks are to be locked

Stack after call



Errors

$0207	idErr	Invalid user ID

C

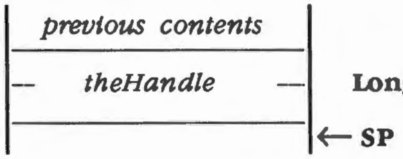
```
extern pascal void HLockAll(userID)

Word     userID;
```

$2202 HUnlock

Unlocks a specified block. Unlocked blocks can be moved or purged during memory compaction.

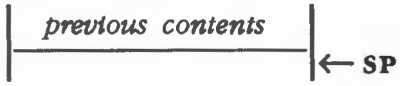
Parameters

Stack before call



Long—HANDLE of block to be unlocked

Stack after call



Errors

$0206 `handleErr` Invalid handle

C

```
extern pascal void HUnlock(theHandle)

Handle     theHandle;
```

$2302 HUnlockAll

Unlocks all of the blocks for a specified user ID. Unlocked blocks can be moved or purged during memory compaction.

Parameters

Stack before call

previous contents	
userID	**Word**—User ID of the blocks to be unlocked
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0207	`idErr`	Invalid user ID

C

```
extern pascal void HUnlockAll(userID)

Word      userID;
```

$1C02 MaxBlock

Returns the size of the largest free block in memory, not counting memory that can be freed by purging or compacting.

Parameters

Stack before call

```
|  previous contents  |
|---------------------|
|--    longspace    --|    Long—Space for result
|---------------------|
|                     |<— SP
```

Stack after call

```
|  previous contents  |
|---------------------|
|--    blockSize    --|    Long—Size in bytes of largest free block in memory
|---------------------|
|                     |<— SP
```

Errors

None

C

```
extern pascal LongWord MaxBlock()
```

$0902 NewHandle

Creates a new block and returns the handle to the block. The *userID* parameter marks the block as belonging to a specific owner. You can specify the user ID as

- A new ID obtained by modifying the *auxID* field of the master user ID
- The master user ID the Memory Manager returned to you in the MMStartUp call
- An entirely new user ID you obtained by calling the Miscellaneous Tool Set GetNewID routine

For more information about the ramifications of these different kinds of IDs, see the sections "User IDs" and "Using the Memory Manager" in this chapter.

The block is located and its attributes are set according to the flags set in the *attributes* parameter, as shown in Figure 12-9.

The *locationPtr* points to the starting address of the block only if the *attributes* parameter specifies fixed address or fixed bank. Otherwise, *locationPtr* is ignored. If a block of 0 bytes is created, *theHandle* becomes an empty handle.

Parameters

Stack before call

previous contents	
— *longspace* —	**Long**—Space for result
— *blockSize* —	**Long**—Size in bytes of block to create
userID	**Word**—User ID to be associated with the block
attributes	**Word**—Attributes (see Figure 12-9)
— *locationPtr* —	**Long**—POINTER to where in memory the block is to begin
	← **SP**

Stack after call

previous contents	
— *theHandle* —	**Long**—HANDLE of new block; empty if block of 0 bytes created
	← **SP**

Errors

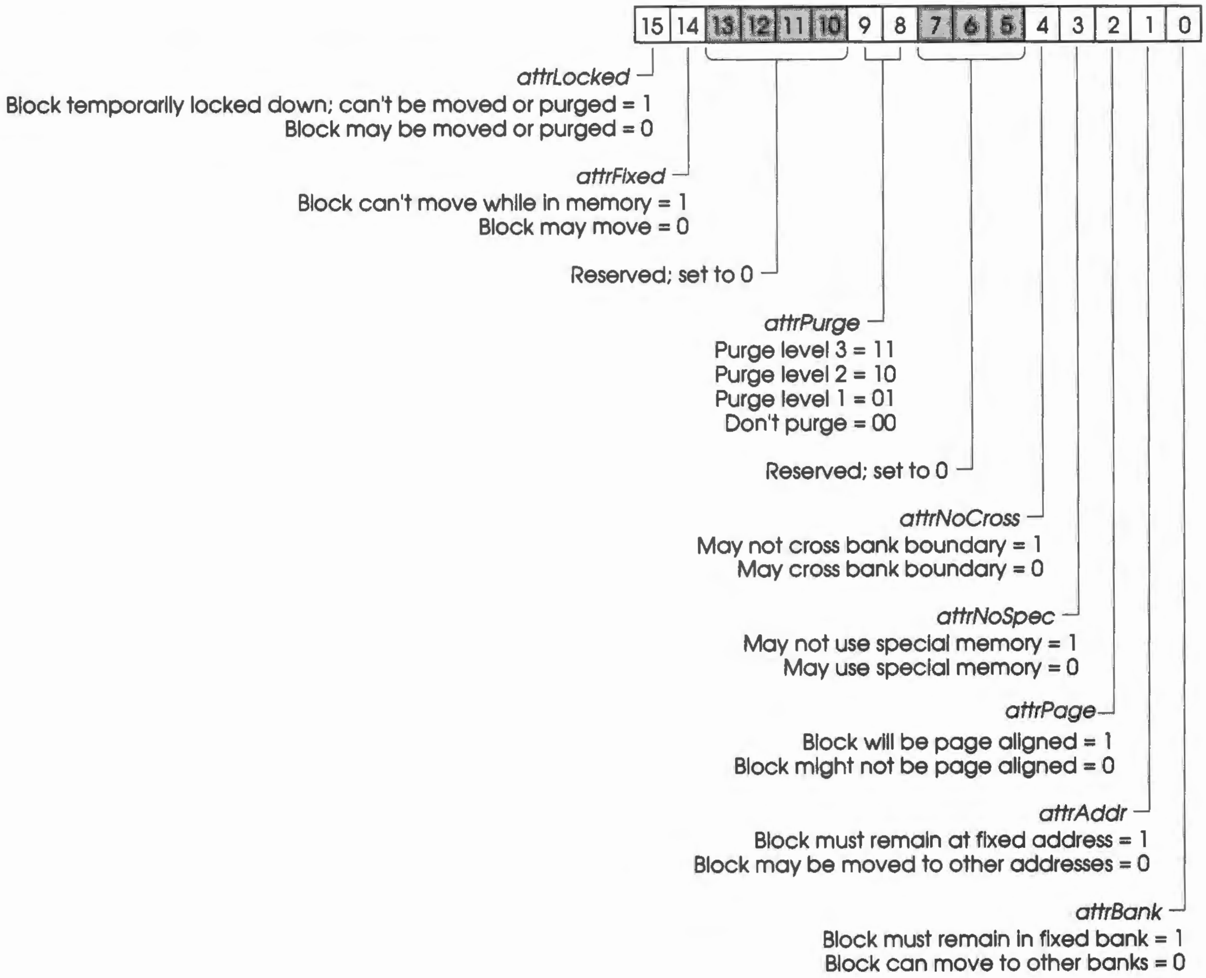
$0201	`memErr`	Unable to allocate block
$0204	`lockErr`	Illegal operation on a locked or immovable block
$0207	`idErr`	Invalid user ID

C

```
extern pascal Handle NewHandle(blockSize,userID,attributes,locationPtr)

LongWord     blockSize;

Word     userID;

Word     attributes;

Pointer     locationPtr;
```

Memory attributes word

Figure 12-9 illustrates the memory attributes word. In all of the bits, a value of 1 means that the attribute applies to the block; you might think of setting the bit to 1 as applying a restriction to the block.



15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
attrLocked
Block temporarily locked down; can't be moved or purged = 1
Block may be moved or purged = 0
attrFixed
Block can't move while in memory = 1
Block may move = 0
Reserved; set to 0
attrPurge
Purge level 3 = 11
Purge level 2 = 10
Purge level 1 = 01
Don't purge = 00
Reserved; set to 0
attrNoCross
May not cross bank boundary = 1
May cross bank boundary = 0
attrNoSpec
May not use special memory = 1
May use special memory = 0
attrPage
Block will be page aligned = 1
Block might not be page aligned = 0
attrAddr
Block must remain at fixed address = 1
Block may be moved to other addresses = 0
attrBank
Block must remain in fixed bank = 1
Block can move to other banks = 0

Figure 12-9
Memory attributes word

$2802 PtrToHand

Copies a specified number of bytes from a source to a destination, with the source specified by a pointer and the destination by a handle. PtrToHand works correctly even if source and destination overlap or cross bank boundaries. No address validation is performed; PtrToHand simply overwrites everything.

Parameters

Stack before call

previous contents	
-- *sourcePtr* --	**Long**—POINTER specifying start of bytes to be copied
-- *destHandle* --	**Long**—HANDLE specifying starting address where bytes will be written
-- *count* --	**Long**—Number of bytes to be copied
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0202	`emptyErr`	Illegal operation on an empty handle
$0206	`handleErr`	Invalid handle

C

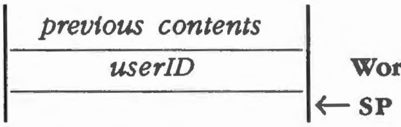
```
extern pascal void PtrToHand(sourcePtr,destHandle,count)

Pointer      sourcePtr;

Handle      destHandle;

LongWord      count;
```

$1302 PurgeAll

Purges all of the purgeable blocks for a specified user ID. Only purgeable, unlocked blocks are purged. If any of the blocks were not purgeable, an error occurs, but any purgeable blocks are purged anyway.

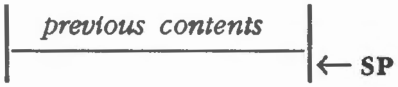
Parameters

Stack before call



Word—User ID whose blocks will be purged

Stack after call



Errors

$0204	`lockErr`	Illegal operation on a locked or immovable block
$0205	`purgeErr`	Attempt to purge an unpurgeable block
$0207	`idErr`	Invalid user ID

C

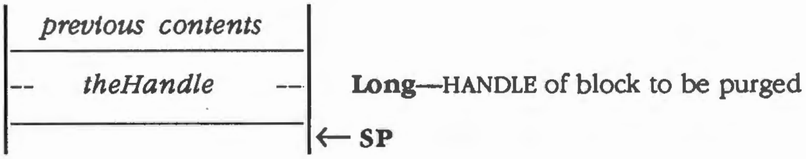
```
extern pascal void PurgeAll(userID)

Word     userID;
```

$1202 PurgeHandle

Purges a specified purgeable, unlocked block, but does not deallocate the handle; that is, *theHandle* itself remains allocated but is empty.

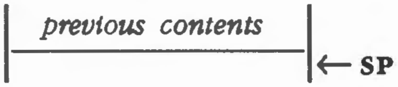
Parameters

Stack before call



Long—HANDLE of block to be purged

Stack after call



Errors

$0204	`lockErr`	Illegal operation on a locked or immovable block
$0205	`purgeErr`	Attempt to purge an unpurgeable block
$0206	`handleErr`	Invalid handle

C

```
extern pascal void PurgeHandle(theHandle)

Handle     theHandle;
```

$0A02 ReAllocHandle

Reallocates a purged block, using new attributes.

The *locationPtr* points to the starting address of the block only if the *attributes* parameter specifies fixed address or fixed bank. Otherwise, *locationPtr* is ignored.

Parameters

Stack before call

previous contents	
blockSize	**Long**—Size in bytes of block to create
userID	**Word**—User ID associated with the block
attributes	**Word**—Attributes (see Figure 12-9 in the section "NewHandle")
locationPtr	**Long**—POINTER to where in memory the block is to begin
theHandle	**Long**—HANDLE of the block to reallocate
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0201	`memErr`	Unable to allocate block
$0203	`notEmptyErr`	Empty handle expected for this operation
$0204	`lockErr`	Illegal operation on a locked or immovable block
$0206	`handleErr`	Invalid handle
$0207	`idErr`	Invalid user ID

C

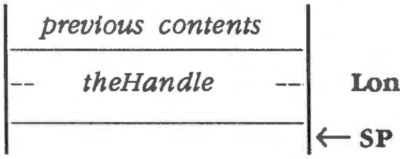
```
extern pascal void ReallocHandle(blockSize,userID,attributes,
locationPtr,theHandle)

LongWord      blockSize;

Word      userID;

Word      attributes;

Pointer      locationPtr;

Handle      theHandle;
```

$0B02 RestoreHandle

Reallocates a purged block using the same *attributes, userID,* and *blockSize* that were associated with the purged handle. The block must not have had a fixed address or fixed bank attribute. Any information in the purged block was lost.

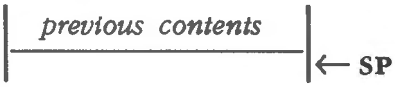
Parameters

Stack before call



Long—HANDLE of the block to restore

Stack after call



Errors

$0201	`memErr`	Unable to allocate block
$0203	`notEmptyErr`	Empty handle expected for this operation
$0206	`handleErr`	Invalid handle
$0208	`attrErr`	Illegal operation with specified attributes; block cannot have fixed address or fixed bank attribute

C

```
extern pascal void RestoreHandle(theHandle)

Handle     theHandle;
```

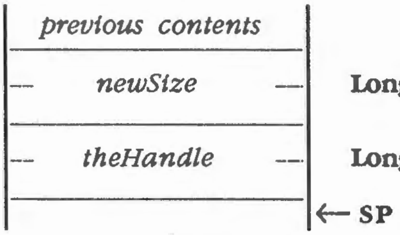
$1902 SetHandleSize

Changes the size of a specified block. The block can be made larger or smaller. If you need more room to lengthen a block, you may compact memory or purge blocks.

You should unlock *theHandle* before making the SetHandleSize call, because the block may have to move to change size. If *newSize* is set to 0, *theHandle* becomes an empty handle.

Parameters

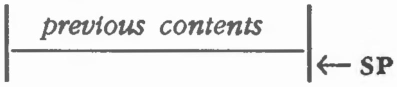
Stack before call



Long—New size of block in bytes

Long—HANDLE of block whose size is to be set

Stack after call



Errors

$0201	`memErr`	Unable to allocate block
$0202	`emptyErr`	Illegal operation on an empty handle, usually caused by attempting to resize a purged block
$0204	`lockErr`	Illegal operation on a locked or immovable block
$0206	`handleErr`	Invalid handle

C

```
extern pascal void SetHandleSize(newSize,theHandle)

LongWord      newSize;

Handle      theHandle;
```

$2402 SetPurge

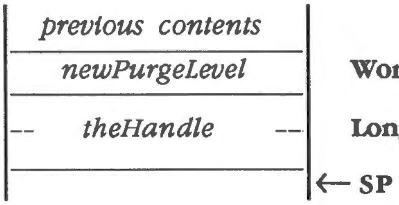
Sets the purge level of a specified block. The purge level is a 2-bit number in the attributes word specifying the purging priority of the block. The values are

3	Most purgeable; used only by the System Loader
2	Next most purgeable
1	Least purgeable
0	Not purgeable

Your application should use only levels 0 to 2; level 3 is reserved for the System Loader.

Parameters

Stack before call



Word—New purge level of block (bits 1-0 only; set others to 0)

Long—HANDLE of block whose purge level is to be set

Stack after call

previous contents
← SP

Errors

$0206	`handleErr`	Invalid handle

C

```
extern pascal void SetPurge(newPurgeLevel,theHandle)

Word     newPurgeLevel;

Handle    theHandle;
```

$2502 SetPurgeAll

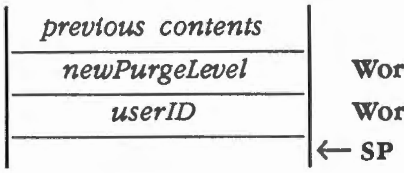
Sets the purge level of all blocks associated with a specified user ID. The purge level is a 2-bit number in the attributes word specifying the purging priority of the block. The values are

3	Most purgeable; used only by the System Loader
2	Next most purgeable
1	Least purgeable
0	Not purgeable

Your application should use only levels 0 to 2; level 3 is reserved for the System Loader.

Parameters

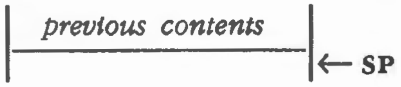
Stack before call



Word—New purge level of block (bits 1-0 only; set others to 0)

Word—User ID whose blocks are to be set to *newPurgeLevel*

Stack after call



Errors

$0207	idErr	Invalid user ID

C

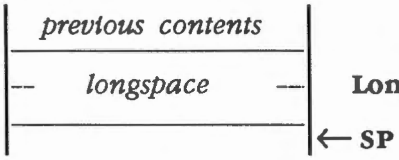
```
extern pascal void SetPurgeAll(newPurgeLevel,userID)

Word     newPurgeLevel;

Word     userID;
```

$1D02 TotalMem

Returns the size of all memory, including the main 256K.

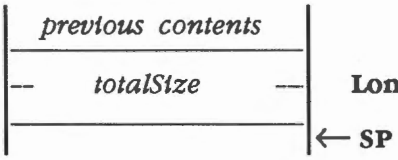
Parameters

Stack before call



Long—Space for result

Stack after call



Long—Total size in bytes of memory, including main 256K

Errors

None

C

```
extern pascal LongWord TotalMem()
```

Memory Manager summary

This section briefly summarizes the constants and tool set error codes contained in the Memory Manager. There are no predefined data structures for the Memory Manager.

Important
These definitions are provided in the appropriate interface file.

Table 12-4
Memory Manager constants

Name	Value	Description
Attribute bits		
attrNoPurge	$0000	Not purgeable
attrBank	$0001	Fixed bank
attrAddr	$0002	Fixed address
attrPage	$0004	Page aligned
attrNoSpec	$0008	May not use special memory
attrNoCross	$0010	May not cross banks
attrPurge1	$0100	Purge level 1
attrPurge2	$0200	Purge level 2
attrPurge3	$0300	Purge level 3
attrPurge	$0300	Test or set both purge bits
attrHandle	$1000	Block of handles (reserved for Memory Manager)
attrSystem	$2000	System handle (reserved for Memory Manager)
attrFixed	$4000	Fixed block
attrLocked	$8000	Locked block

Table 12-5
Memory Manager error codes

Code	Name	Description
$0201	memErr	Unable to allocate block
$0202	emptyErr	Illegal operation on an empty handle
$0203	notEmptyErr	Empty handle expected for this operation
$0204	lockErr	Illegal operation on a locked or immovable block
$0205	purgeErr	Attempt to purge an unpurgeable block
$0206	handleErr	Invalid handle
$0207	idErr	Invalid user ID
$0208	attrErr	Illegal operation with specified attributes

Chapter 13

Menu Manager

The **Menu Manager** supports the use of the style of menus approved by the *Human Interface Guidelines: The Apple Desktop Interface.* **Menus** allow users to examine all choices available to them at any time without being forced to choose one of them and without having to remember command words or special keys. The Apple IIGS user simply positions the cursor in the menu bar and presses the mouse button over a menu title. The application then calls the Menu Manager, which highlights the selected title (by redrawing it in inverted colors) and "pulls down" the menu below it; such menus are thus called **pull-down menus.** So long as the mouse button is held down, the menu is displayed. Dragging through the menu causes each of the menu items (commands) to be highlighted in turn. If the mouse button is released over an item, that item is "chosen." The item blinks briefly to confirm the choice, and the menu disappears.

When the user chooses an item, the Menu Manager tells the application which item was chosen, and the application performs the corresponding action. When the application completes the action, it removes the highlighting from the menu title, indicating to the user that the operation is complete.

If the user moves the cursor out of the menu with the mouse button held down, the menu remains visible, though no menu items are highlighted. If the mouse button is released outside the menu, no choice is made: The menu just disappears, and the application takes no action. The user can always look at a menu without causing any changes in the document or on the screen.

A preview of the Menu Manager routines

To introduce you to the capabilities of the Menu Manager, all Menu Manager routines are grouped by function and briefly described in Table 13-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the Menu Manager routines (discussed in alphabetical order).

Table 13-1
Menu Manager routines and their functions

Routine	Description
Housekeeping routines	
MenuBootInit	Initializes the Menu Manager; called only by the Tool Locator—must not be called by an application
MenuStartUp	Starts up the Menu Manager for use by an application
MenuShutDown	Shuts down the Menu Manager when an application quits
MenuVersion	Returns the version number of the Menu Manager
MenuReset	Resets the Menu Manager; called only when the system is reset—must not be called by an application
MenuStatus	Indicates whether or not the Menu Manager is active
Initialization and termination routines	
NewMenuBar	Creates a default menu bar with no menus
NewMenu	Allocates space for a menu list and its items
DisposeMenu	Frees the memory allocated by NewMenu
FixMenuBar	Computes standard sizes for the menu bar and menus
CalcMenuSize	Sets menu dimensions, either manually or automatically
User interaction routines	
MenuSelect	Draws highlighted titles, pulls down menus, and handles user interaction when a mouse button is clicked on a menu bar
MenuKey	Maps a character to the associated menu and item for that character
MenuRefresh	Attempts to refresh the screen
MenuGlobal	Turns menu help on or off or returns the current state of menu help; your application can use menu help to let users find out how inactive items can be made active
Menu drawing routines	
DrawMenuBar	Draws the current menu bar, along with any menu titles on the bar
HiliteMenu	Highlights or unhighlights the title of a specified menu
FlashMenuBar	Flashes the entire current menu bar
Menu and item shuffling routines	
InsertMenu	Inserts a specified menu into the menu list after a specified menu item or at the front of the list
DeleteMenu	Removes a specified menu from the menu list
InsertMItem	Inserts a menu item into a menu after a specified menu item or at the front of the list
DeleteMItem	Removes a specified item from the current menu

Table 13-1 (continued)
Menu Manager routines and their functions

Routine	Description
Menu bar access routines	
SetSysBar	Sets a new system bar; the system menu bar becomes the current menu bar
GetSysBar	Returns the handle of the current system menu bar
SetMenuBar	Sets the current menu bar
GetMenuBar	Returns the handle of the current menu bar
SetBarColors	Sets the normal, inverse, and outline colors of the current menu bar
GetBarColors	Returns the colors for the current menu bar
SetMTitleStart	Sets the starting position for the leftmost title within the current menu bar
GetMTitleStart	Returns the starting position for the leftmost title within the current menu bar
CountMItems	Returns the number of items, including any dividing lines, in a specified menu
Menu record access routines	
GetMHandle	Returns a handle to a menu record
SetMTitleWidth	Sets the width of a title
GetMTitleWidth	Returns the width of a menu title
SetMenuFlag	Sets the menu to a specified state
GetMenuFlag	Returns the menu flag for a specified menu
SetMenuTitle	Specifies the title for a menu
GetMenuTitle	Returns a pointer to the title of a menu
SetMenuID	Specifies a new menu number
Item record access routines	
SetMItem	Specifies the name for a menu item by pointing to an item line
GetMItem	Returns a pointer to the name of an item
SetMItemName	Specifies the name of a menu item by pointing to a Pascal-type string
EnableMItem	Sets a specified menu item to display normally and allows it to be selected
DisableMItem	Sets a specified menu item to display in dimmed characters and does not allow it to be selected
CheckMItem	Sets a specified menu item to display or to not display a check mark to the left of the item
SetMItemMark	Sets a specified menu item to display or to not display a specified character to the left of the item
GetMItemMark	Returns the current character that is displayed to the left of a specified menu item
SetMItemStyle	Sets the text style for a specified menu item
GetMItemStyle	Returns the text style for a specified menu item
SetMItemFlag	Sets a specified item number to be underlined or not underlined and sets the highlighting style
GetMItemFlag	Returns the values for a specified item, such as whether it is disabled, underlined, or highlighted
SetMItemID	Specifies the ID number of a menu item
SetMItemBlink	Determines how many times all menu items should blink when selected

(continued)

Table 13-1 (continued)
Menu Manager routines and their functions

Routine	Description
Miscellaneous Menu Manager routines	
GetMenuMgrPort	Returns a pointer to the Menu Manager's port
MenuNewRes	Adjusts screen resolution and redraws the current system menu bar
InitPalette	Reinitializes the palettes needed for the colored Apple logo in the system menu bar

Menu bars

A **menu bar** is an outlined rectangle that holds the titles of all the menus associated with the bar. A menu may be enabled or temporarily disabled. A disabled menu can still be pulled down, but its title and all the items in it are dimmed and not selectable.

❖ *Note:* If your program is likely to be translated into other languages, the menu titles may take up more space. If you're having trouble fitting your menus into the menu bar, you should review your menu organization and menu titles.

System menu bar

There is one special type of menu bar called the **system menu bar,** shown in Figure 13-1.

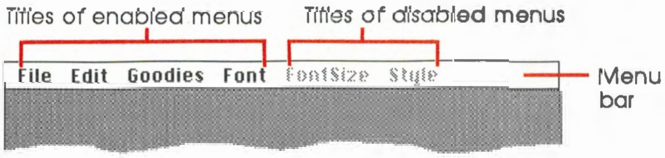


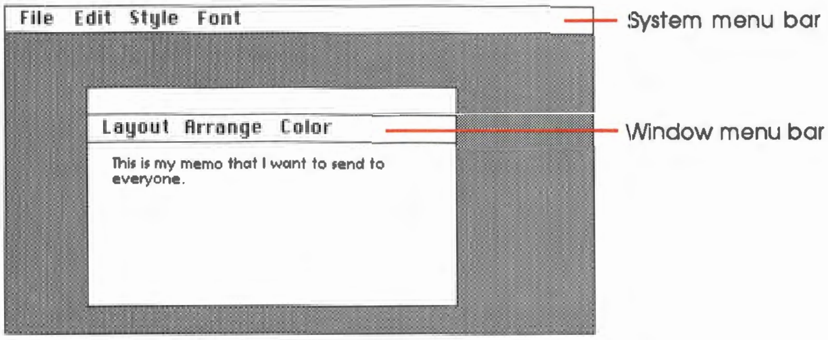
Figure 13-1
System menu bar

Only one system menu bar at a time can appear on the screen. The system menu bar always appears at the top of the Apple IIGS screen; nothing but the cursor ever appears in front of it. In applications that support desk accessories, the first menu should be the **Desk Accessory menu** (the menu whose title is a colored Apple symbol). The Desk Accessory menu contains the names of all available desk accessories. When the user chooses a desk accessory from the menu, the title of a menu belonging to the desk accessory may appear in the menu bar for as long as the accessory is active, or the entire menu bar may be replaced by menus belonging to the particular desk accessory.

Color number 1 is reserved for drawing the Apple logo as the title for the Desk Accessory menu. Therefore, color number 1 should not be used as the color for the bar, as the inverse colors for the bar, or as the outline color. The color can be used for menus, items, window menu bars, and the rest of the screen.

Window menu bars

In addition to the system menu bar, your application can have various **window menu bars** (see Figure 13-2). These can appear anywhere on the screen and in windows.



This is my memo that I want to send to everyone.

Figure 13-2
Window menu bar

Window menu bars are provided to give you more flexibility and to address the limited resolution in 320 mode. However, be careful when using window menu bars; your application may be better served by a revised system menu bar. In addition, there are several factors which you must consider, including the following:

- TaskMaster doesn't handle events in the content region of a window; thus, your application must handle events coming from a window menu bar.
- To affect a window menu bar, you must first install as the current GrafPort the port of the window that owns the menu bar.
- Because the menus will be drawn in the window's port, they will be clipped to that port's clip region; thus, the menus must fit inside the content region of the window or parts of the menus will not be visible to the user.

Menu appearance

A standard menu consists of a number of menu items listed vertically inside a shadowed rectangle. A **menu item** may be the text of a command or just a line dividing groups of choices. Menus always appear in front of everything else, except the cursor. The menu in Figure 13-3 is a menu with six items, including one dividing line.

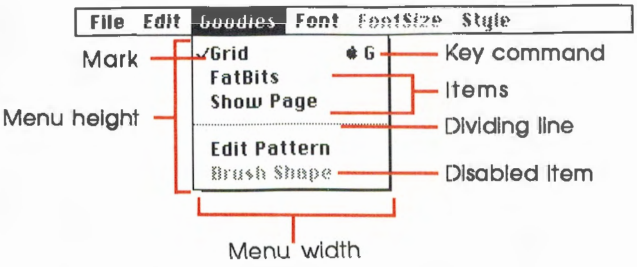


Figure 13-3
Standard menu

Each item can vary slightly from the standard appearance:

- A mark may appear on the left side of the item to denote the status of the item or of the mode it controls. See the sections "SetMItemMark," "GetMItemMark," and "CheckMItem" in this chapter.
- An Apple logo may appear on the right side of the item to show that the item may be invoked from the keyboard (that is, the item has a keyboard equivalent), followed by a capital letter. If the user presses this key while holding down the Apple key, the menu item is invoked just as if it had been chosen from the menu. See the section "MenuKey" in this chapter.
- Each item's text may have its own style. See the sections "SetMItemStyle" and "GetMItemStyle" in this chapter.
- An item can be dimmed to indicate that it is disabled and can't be chosen (dividing lines should always be disabled). See the sections "DisableMItem" and "EnableMItem" in this chapter.
- Any menu can be drawn directly by the application and can contain anything (see the section "Defining Your Own Menus" in this chapter).

❖ *Note:* If an item without any text is dimmed, the item appears the same as the menu background.

If the standard menu doesn't suit your needs—for example, if you want more graphics or a nonlinear text arrangement—you can define a custom menu that, although visibly different to the user, responds to your application's Menu Manager calls just like a standard menu (see the section "Defining Your Own Menus" in this chapter).

Keyboard equivalents for commands

Your program can set up a **keyboard equivalent** for any of its menu commands to allow the user to invoke the command from the keyboard. The character you specify for a keyboard equivalent will usually be a letter that the user can type in either uppercase or lowercase. For example, typing either *C* or *c* while holding down the Apple key invokes the command whose equivalent is *C.*

❖ *Note:* For consistency with the Apple *Human Interface Guidelines,* you should specify the letter in uppercase in the menu.

Using the Menu Manager

This section discusses how the Menu Manager routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

The Menu Manager depends upon the presence of the tool set shown in Table 13-2 and requires that at least the indicated version of the tool set be present.

Table 13-2
Menu Manager—other tool sets required

Tool set number		Tool set name	Minimum version needed
$01	#01	Tool Locator	1.2
$02	#02	Memory Manager	1.2
$04	#04	QuickDraw II	1.2
$06	#06	Event Manager	1.0
$0E	#14	Window Manager	1.3
$10	#16	Control Manager	1.3

The first Menu Manager call that your application must make is MenuStartUp. When you quit your application, you must call MenuShutDown.

The procedure described on the following pages presents the steps necessary to set up the menu bar using 65C816 assembly language. The procedure basically consists of the following steps (the next few sections explain these steps in detail):

1. Initialize the Menu Manager.
2. Define the menus and items.
3. Set the sizes of the menu bar and menus.
4. Draw the menu bar.
5. Accept input from the user, either with or without the Window Manager routine TaskMaster.

Initializing the Menu Manager

Enter the following code to initialize the Menu Manager:

```
lda     MyID                  ; ID number returned by Memory Manager MMStartUp
pha                           ; Pass ID to MenuStartUp
lda     MenuZPage             ; Direct page for Menu Manager use
pha                           ; Pass direct page
_MenuStartUp                  ; Creates and draws an empty menu bar; no output
```

You now have a system menu bar that contains no menus and is the current menu bar. If you are only using one menu, the system menu bar will always be the current menu bar. You can also create several menu bars by changing the current menu bar. See the section "SetSysBar" in this chapter.

Defining menus and items

Your program then must define menus and items by providing a list of menu and item lines to NewMenu for each menu (see the section "Menu Lists: Menu Lines and Item Lines" in this chapter) and by using the InsertMenu routine to add them to the system menu bar. FixMenuBar may be useful in setting default sizes.

```
pha                           ; Space for returned handle
pha
pea     Menu1|-16             ; Pass pointer to menu/item lines
pea     Menu1
_NewMenu                      ; Allocate a menu record and initialize it
pla                           ; Get returned menu handle
sta     menuHandle            ; and save it
pla
sta     menuHandle+2
ora     menuHandle            ; Check for bad handle
beq     error                 ; Unable to allocate handle or bad menu/item lines
lda     menuHandle+2          ; Pass handle of menu to insert (just allocated)
pha
lda     menuHandle
pha
pea     0                     ; Insert menu as first menu flag
_InsertMenu                   ; Insert menu in data structures, not drawn
```

In the preceding example, the menu and item line data might look like this:

```
DATA:

Menu1   dc      c'>>Title\N1',i1'0'           ; Menu title

        dc      c'--Item 1\N256',i1'0'        ; Item text

        dc      c'.'                          ; Termination character
```

Setting the sizes of the menu bar and items

After you have created and inserted all the menus you want, you must set the sizes of the menu bar and menus.

```
        pha                         ;Space for returned bar height

        _FixMenuBar                 ;Initialize menu bar and menu sizes

        pla                         ;Get the height of the menu bar; generally not useful
```

Drawing the new menu bar

After the menu bars have been created and initialized, you should draw them. If you need to change the color of the menu bar and the menus, use the SetBarColors routine before drawing the menu.

```
        _DrawMenuBar                ;Draws the menu bar and the menu titles
```

The menu bar is displayed, and initialization is complete.

Important

Remember to consult the Apple *Human Interface Guidelines* before changing the color of the menu bar and the menu titles.

Accepting input from the user

Now you are ready to accept input from the user. You can do so either with or without the Window Manager routine TaskMaster. The routine is described in detail in Chapter 25, "Window Manager," in Volume 2; this section describes how it is used with the Menu Manager.

Whether or not you use TaskMaster, you need to know about the task record, which looks like this in assembly language:

```
TaskRec         anop
what            ds       2            ; First part is event record, defined in Event Manager
message         ds       4
when            ds       4
where           ds       4
modifiers       ds       2
TaskData        ds       4            ; Place for returned values
TaskMask        ds       4            ; Used by Window Manager calls, not really needed here
```

With TaskMaster

The following example uses TaskMaster:

```
poll            pha                   ;Space for returned value
                pea     $002E         ;Accept at least key and mouse events
                pea     TaskRec|-16   ;Pass pointer to task record
                pea     TaskRec
                _TaskMaster           ;Task Master will call GetNextEvent for you
                pla                   ;Get return flag
                beq     poll          ;Was there an event?

                cmp     #wInMenuBar   ;Was there a menu event?
                bne     poll          ;If not, continue to poll (or check other events)

                lda     TaskData      ;Get ID of item selected
                and     #$00FF        ;Table shortcut, if all item IDs must be 256 to 511
                asl     a
                tax
                jmp     (itemProcs,x) ;Jump to item handler

itemProcs       dc      i'item1'      ;Address for item handler (only one item here)
```

```
item1                                   ;(Perform whatever action the item dictates)
            pea     0                   ;Pass FALSE to unhighlight the menu title
            lda     TaskData+2          ;Get menu's ID, returned from MenuKey or MenuSelect
            pha                         ;Pass the menu's ID number
            _HiliteMenu                 ;Unhighlight the menu's title after item task action

            jmp     poll                ;Return to polling
```

Without TaskMaster

The following example does not use TaskMaster:

```
poll        pha                         ;Space for returned value
            pea     $002E               ;Accept at least key and mouse events
            pea     TaskRec|-16         ;Pass pointer to event record
            pea     TaskRec
            _GetNextEvent
            pla                         ;Get return flag
            beq     poll                ;Was there an event?

            lda     what                ;Get event code
            cmp     #3                  ;Keypress event?
            beq     ck_key
            cmp     #5                  ;Autokey event?
            bne     ck_button

ck_key      pea     TaskRec|-16         ;Pass TaskRec, which contains key pressed in 'message'
            pea     TaskRec
            pea     0                   ;Use current menu bar flag
            pea     0
            _MenuKey
            bra     ck_Menu             ;Execute common menu selection routine

ck_button   cmp     #1                  ;Mouse down event?
            bne     poll                ;If not key or button, continue to poll
```

You should first determine whether the button was pressed in the menu bar. If you are using the Window Manager, call the FindWindow routine, which returns a value that indicates whether the specified point is in the system menu bar. If you are not using the Window Manager, your application must determine how to tell whether the button was pressed in the menu bar (the height of the menu bar, returned from FixMenuBar, could be useful).

Continue if the point is in the menu bar:

```
                pea     TaskRec|-16    ;Pass TaskRec, which contains where mouse was pressed
                pea     TaskRec
                pea     0              ;Use current menu bar flag
                pea     0
                _MenuSelect            ;Menu select will wait for button up before returning

ck_Menu         lda     TaskData       ;Get ID of item selected
                beq     poll           ;Was an item selected?  If not, return to polling

                cmp     #256           ;Item IDs 1-255 are reserved for desk accessory items
                bcc     call_deskmgr   ;Do what is necessary for desk accessories

                and     #$00FF         ;Shortcut for table, if all item IDs between 256-511
                asl     a
                tax
                jmp     (itemProcs,x)  ;Jump to item handler

itemProcs       dc      i'item1'       ;Address for item handler (only one item here)

item1                                  ;(Perform whatever action the item dictates)
                pea     0              ;Pass FALSE to unhighlight the menu title
                lda     TaskData+2     ;Get menu's ID, returned from MenuKey or MenuSelect
                pha                    ;Pass the menu's ID number
                _HiliteMenu            ;Unhighlight the menu's title after item action

                jmp     poll           ;Return to polling
```

❖ *Note:* The Menu Manager tries to automatically save and restore the screen behind the menu or tells the Window Manager to update the screen. However, if you are not using the Window Manager, and the Menu Manager cannot allocate a buffer large enough to save the screen behind the menu, your application must update the screen area after a menu has been pulled down. See the section "MenuRefresh" in this chapter.

If you want your menu bar or menu items to change while on the screen, you can use SetMenuTitle, InsertMenu, DeleteMenu, SetMItem, InsertMItem, and DeleteMItem to rearrange the menus and items.

Several miscellaneous Menu Manager routines may be useful. CalcMenuSize calculates the dimensions of a menu and is called by FixMenuBar. CountMItems counts the number of items in a menu. FlashMenuBar inverts the menu bar or menu title. SetMItemBlink controls the number of times a menu item blinks when chosen.

Menu lists: menu lines and item lines

Menus can be created by passing a pointer to a list of menu lines and item lines to the NewMenu routine. NewMenu parses the **menu lines** and **item lines,** allocates enough memory for necessary records, and initializes those records. The list can be edited using a word processor, thus allowing users to easily customize their own menus. An example of a list is

```
>>Title 1\N1
--Item string 1\N256
--Item string 2\N257
--Item string 3\N258
;
```

This is a simple list of one menu line and three item lines. The first character on the first line is the **title character,** which denotes the start of a menu. The second character simply repeats the first character, since the second character is changed by the Menu Manager for internal reasons. Each line is terminated by a return (decimal 13) or a null byte (0). The first character on subsequent lines that is different from the title character is the **item character.** The second character on subsequent lines is also changed by the Menu Manager for its own purposes. Finally, a **termination character,** different from the menu and item character, denotes the end of the list.

In the preceding example, the > character is the title character, the - character is the item character, and the ; character is the termination character. However, you may use any characters, so long as the title and item characters are different and the termination character is different from the item character. (The title and termination characters may be the same.)

Important
Menu lines and item lines must remain in memory until the menu is disposed of.

Backslash characters are also included in the preceding example. A **backslash character** denotes the end of a title's text and the beginning of special characters. The *N* immediately following the backslash in the example precedes an ID number, which is a decimal, unsigned ASCII number. Every menu and title item, even dividing lines, must have an ID number. The ID number for each menu title must be different from the ID number for every other menu title on the menu bar, and the ID number of each item must be different from the ID number for every other item. Items that are dividing lines, and thus always disabled, can have the same ID number.

The special characters are shown in Table 13-3.

Table 13-3
Menu special characters

Character	Description
\	Denotes beginning of special characters
*	Followed by the character to be displayed on the screen as a keyboard equivalent (usually an uppercase letter) followed by the alternate keyboard character (usually the corresponding lowercase letter)
B	Make the text bold
C	Followed by a character to be used to mark the item
D	Dim (disable) the selection
H	Hexadecimal, non-ASCII ID number follows; low-order byte/high-order byte
I	Italicize the text
N	Decimal ASCII ID number follows; any length between 1 and 65535
U	Underscore the text
V	Place an underline under the item without using a separate item
X	Use color replace, and not XOR, highlighting

❖ *Note:* You cannot include a backslash (\) in a text string. It will always be seen as the beginning of special characters.

All special characters can be used for items, but the characters *, B, C, I, U, and V cannot be used for menu titles.

An example of a menu and item lines using multiple special characters and different title, item, and terminating characters is as follows:

`$$Title 1\N1`	Title character = $, ID = 1; ID can be same as item ID
`--Item-string 1\N256*Xx`	Item character = dash, ID = 256, key equivalents = X and x
`--Item-string 2\BC√UN257`	Item character = dash, bold, checked, underscored; ID = 257
`--Item-string 3\IN258`	Item character = dash, text is italicized, ID = 258
`$`	Terminating character; can be same as title character

❖ *Note:* Because you cannot type a √ character from the keyboard, you must write a line of code that looks something like the following fragment of 65816 assembly language (the ASCII equivalent for √ is 18):

```
dc   c'--Item-string 2\BC',i1'18'c'UN257',i1'0'
```

Using just the @ symbol in a title provides the Apple logo. The @ must follow the character denoting a menu title and then be followed by an end-of-line mark (carriage return). Do not place a space before or after the @, as you should with other menu titles.

An example of an Apple logo menu title is as follows:

`$$@\N1X`   Apple logo title; ID = 1, color replace highlighting

A single dash (-) as an item's text denotes a dividing line. Special characters apply to dividing lines. Dividing lines should be marked as dimmed with *D*. For example, if the > character denotes an item line, the code

```
>>-\N256D
```

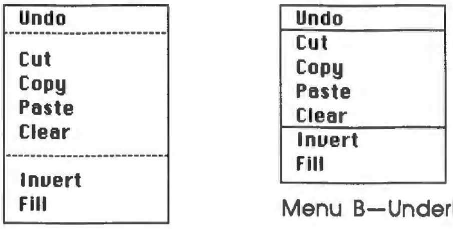
will produce a dashed, dimmed dividing line.

Dividing lines and underlines

There are two standard ways to separate groups of items. The first is to use a **dividing line,** created with an item string that has only a place-holding character and a single dash. It uses the space of an entire item and an entire item record. The second way is to use an **underline,** set either in the menu line or SetMItemFlag. This will draw a solid line on the bottom line of the item. The underlined item doesn't use any more space, on the screen or in memory, than the item does without the underline.

The advantages of underlines are that they save memory, the menu draws faster, the user has to move the mouse less to move from the menu's title to the last item in the menu, and you can put more items in the menu and still have dividing lines. The disadvantage of underlines is that there isn't as much space separating items, which is the dividing line's function.

Figure 13-4 shows two menus with the same information. Menu A uses dividing lines and has 9 items (dividing lines count as items). Menu B uses underlines and has 7 items (underlines don't count as items). Menu B looks crowded and would look even worse if one of the underlined items had descending lowercase letters.



Menu A—Dividing Lines

Menu B—Underlines

Figure 13-4
Dividing lines and underlines

ID number assignment

ID numbers must be assigned in the menu and item line list. The ID numbers must be allocated as shown in Table 13-4.

Warning
A menu ID must be unique for each menu; that is, no two menus can have the same ID or the system will fail. Similarly, no two items can have the same item ID.

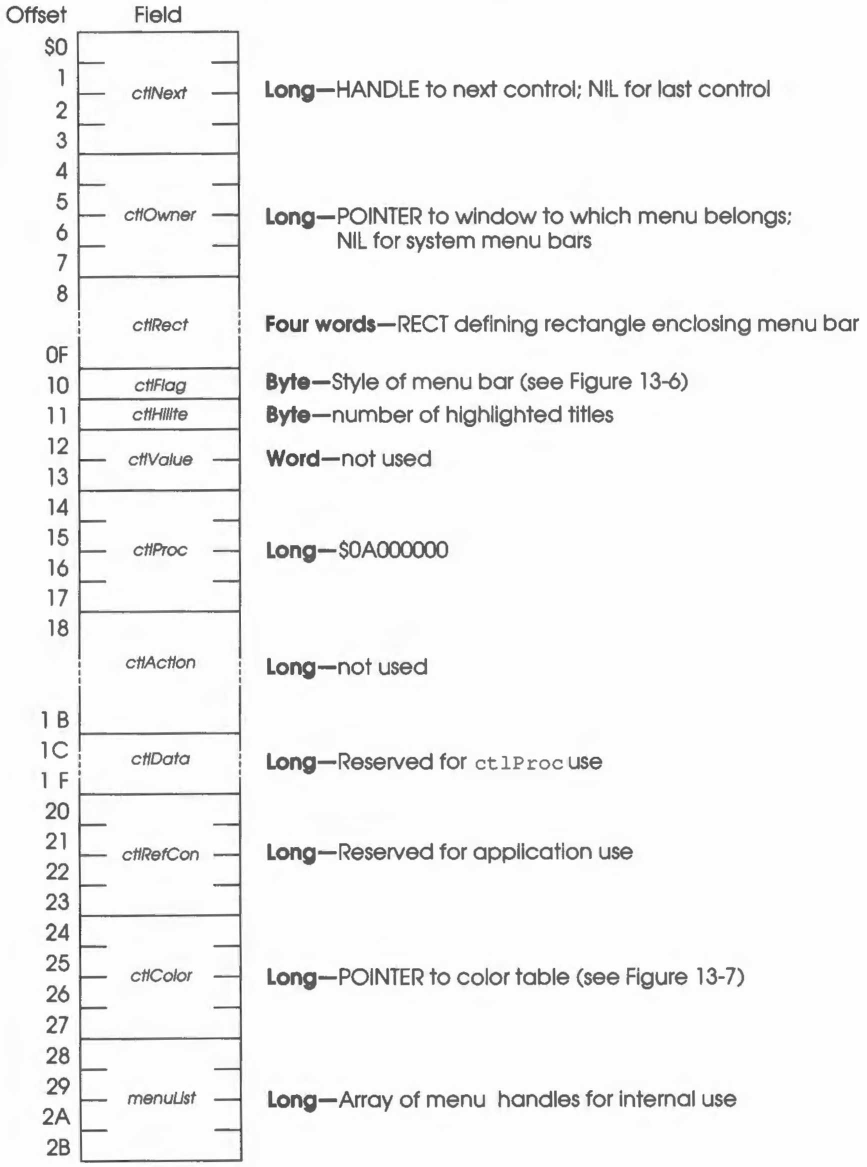
Table 13-4
ID number assignment

Hex number	Decimal number	Description
Menu ID numbers		
$0000	0	Reserved for internal use; usually indicates first menu in bar
$0001–$FFFE	1–65534	Reserved for application use
$FFFF	65535	Reserved for internal use; usually indicates last menu in bar
Item ID numbers		
$0000	0	Reserved for internal use; usually indicates first item in menu
$0001–$00F9	1–249	Reserved for desk accessory items
$00FA	250	Reserved for Undo edit item
$00FB	251	Reserved for Cut edit item
$00FC	252	Reserved for Copy edit item
$00FD	253	Reserved for Paste edit item
$00FE	254	Reserved for Clear edit item
$00FF	255	Reserved for Close command item
$0100–$FFFE	256–65534	Reserved for application use
$FFFF	65535	Reserved for internal use; usually indicates last item in menu

After the NewMenu call has been completed, you can change the ID numbers with the SetMItemID, GetMItem, and SetMenuID routines.

Menu bar records

The Menu Manager keeps the information required for performing operations on a menu bar in a **menu bar record,** which contains the menu position, color, menu lists, item lists, and other flags the Menu Manager needs to manage menus. The menu bar record is similar to a control record (see Chapter 4, "Control Manager"), with the addition of a menu list at the end. The record is illustrated in Figure 13-5.



Offset	Field	
$0 1 2 3	*ctlNext*	**Long**—HANDLE to next control; NIL for last control
4 5 6 7	*ctlOwner*	**Long**—POINTER to window to which menu belongs; NIL for system menu bars
8 0F	*ctlRect*	**Four words**—RECT defining rectangle enclosing menu bar
10	*ctlFlag*	**Byte**—Style of menu bar (see Figure 13-6)
11	*ctlHilite*	**Byte**—number of highlighted titles
12 13	*ctlValue*	**Word**—not used
14 15 16 17	*ctlProc*	**Long**—$0A000000
18 1 B	*ctlAction*	**Long**—not used
1C 1 F	*ctlData*	**Long**—Reserved for `ctlProc` use
20 21 22 23	*ctlRefCon*	**Long**—Reserved for application use
24 25 26 27	*ctlColor*	**Long**—POINTER to color table (see Figure 13-7)
28 29 2A 2B	*menuList*	**Long**—Array of menu handles for internal use

Figure 13-5
Menu bar record

The bit positions in the *ctlFlag* are shown in Figure 13-6.

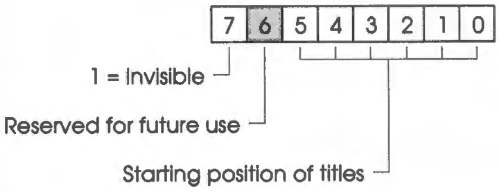


Figure 13-6
Menu bar *ctlFlag* values (*Note:* At the time of publication, the invisible flag was not yet implemented.)

The bit positions in the color table pointed to by *ctlColor* are shown in Figure 13-7.

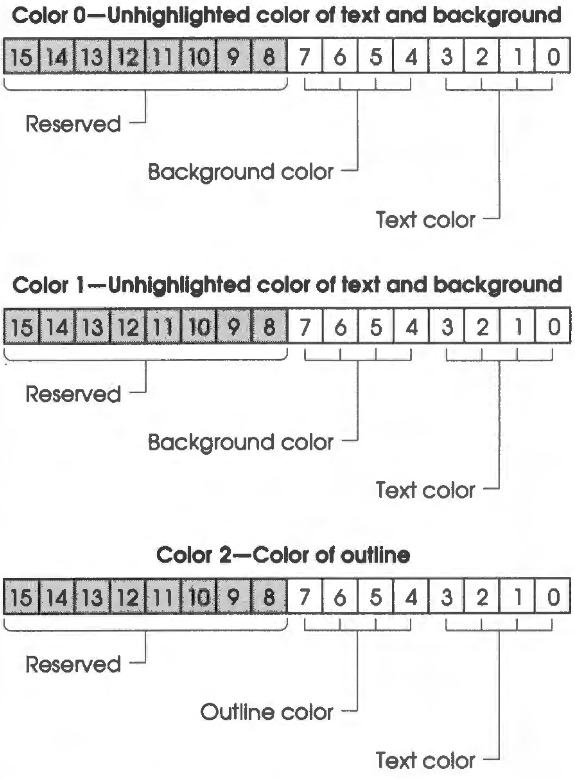
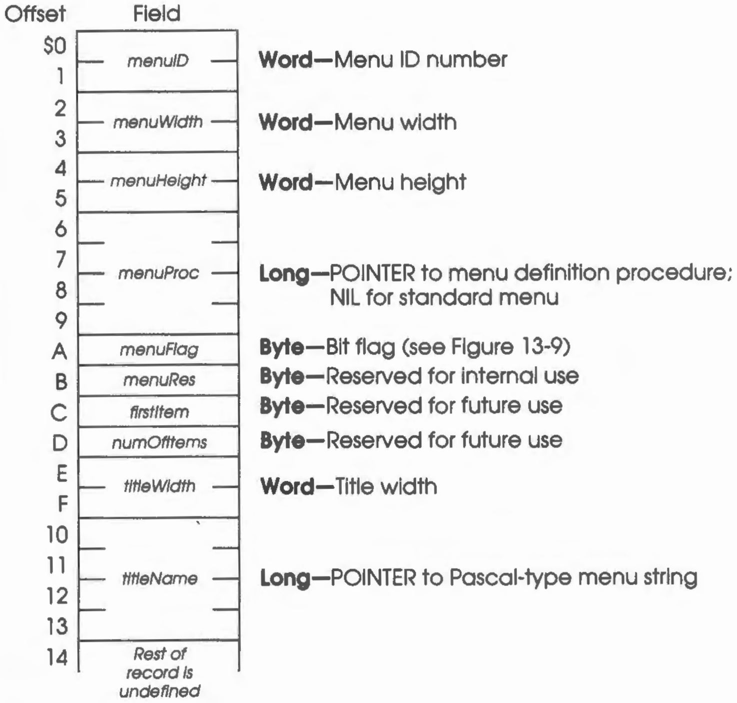


Figure 13-7
Menu bar color table

The *menuList* in the menu bar record is an array of handles to the menu records. Menu records are explained in the next section.

Menu records

A **menu record** provides information about one of the menus in a menu bar. Only the first part of the menu record is defined. The defined part of the record is illustrated in Figure 13-8, and each field is explained in detail following the figure.



Offset	Field	
$0	*menuID*	**Word**—Menu ID number
1		
2	*menuWidth*	**Word**—Menu width
3		
4	*menuHeight*	**Word**—Menu height
5		
6	*menuProc*	**Long**—POINTER to menu definition procedure; NIL for standard menu
7		
8		
9		
A	*menuFlag*	**Byte**—Bit flag (see Figure 13-9)
B	*menuRes*	**Byte**—Reserved for internal use
C	*firstItem*	**Byte**—Reserved for future use
D	*numOfItems*	**Byte**—Reserved for future use
E	*titleWidth*	**Word**—Title width
F		
10	*titleName*	**Long**—POINTER to Pascal-type menu string
11		
12		
13		
14	*Rest of record is undefined*	

Figure 13-8
Menu record

menuID: See the section "ID Number Assignment" in this chapter.

menuWidth: The width of the menu is usually set by the CalcMenuSize routine. However, this value can be changed by the application. If the width is set to be too narrow, text will be drawn outside of the menu and thus will alter other elements on the screen.

menuHeight: The height of the menu is generally set by CalcMenuSize. However, this value can be changed by the application. If the height is set to be too short, text will be drawn outside of the menu and thus will alter other elements on the screen.

menuProc: A value of 0 to 255 in this field means that a standard definition procedure is being used. Values greater than 255 are considered to be the address of a custom menu definition procedure and are called accordingly. See the section "Defining Your Own Menus" in this chapter for more information about custom menus.

menuFlag: This is a bit flag controlling various menu attributes, as shown in Figure 13-9.

Warning

The undefined bits in Figure 13-9 are reserved for internal use by the Menu Manager; their values must not be changed by an application.

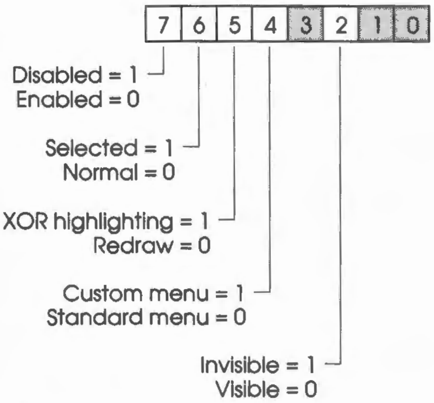


Figure 13-9
The *menuFlag* values (*Note*: At the time of publication, the invisible flag was not yet implemented.)

menuRes: This byte is reserved and should not be relied on for information nor modified by an application.

firstItem: This byte is reserved for the future implementation of menus that can be scrolled. This field should not be relied on for information nor modified by applications written before the implementation is defined.

numOfItems: This byte is reserved for the future implementation of scrollable menus. This field should not be relied on for information, nor modified by applications written before the implementation is defined.

titleWidth: The width of the title can be set to any value between 1 and $FFFF (unsigned). The value is used to highlight the title and compute where the next menu title should start. The title's text is left justified.

titleName: This field indicates a pointer to the string to be used for the menu title. The first byte of the string should be the length of the string followed by the ASCII text. Custom menus may store any type of value desired, so long as the title is drawn by the custom definition procedure.

Defining your own menus

The standard type of menu is predefined. However, you may want to define your own type of menu—perhaps one with more graphics or a nonlinear text arrangement. QuickDraw and the Menu Manager make it possible for you to do this.

To define your own type of menu, you write a custom **menu definition procedure.** The Menu Manager calls the menu definition procedure to perform basic operations, such as drawing the menu.

Warning

Do not make any Menu Manager calls from within a custom definition procedure.

To create a custom menu record you must allocate a block of memory large enough for your menu record. Only the defined part (see the section "Menu Records" in this chapter) of the menu record has to follow Menu Manager form; the format of the rest of the record is up to you. You can also pass a menu line with no items to the NewMenu routine and then resize the allocated block to your needs. Fields in the menu record that need to be initialized are

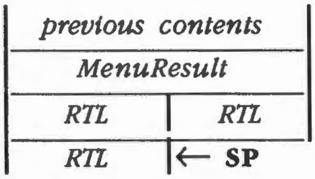
menuID	Menu ID number
menuWidth	Width of menu, or you can wait for the mSize operation
menuHeight	Height of menu, or you can wait for the mSize operation
menuProc	Pointer to menu definition procedure
menuFlag	In addition to other flags, bit 4 must be set
titleWidth	Width of title
titleName	Pointer to title text (first byte of text is a length byte) or to any other desired data

You may choose any name you wish for the menu definition procedure. The inputs and outputs are as follows:

Stack upon entry to custom routine

previous contents		
wordspace		**Word**—Space for result
menuMessage		**Word**—Operation to perform
— *theMenuHandle* —		**Long**—HANDLE to menu
— *rectPtr* —		**Long**—POINTER to RECT of rectangle enclosing menu
xHitPt		**Word**—X coordinate of point to check
yHitPt		**Word**—Y coordinate of point to check
menuParam		**Word**—Additional parameter for each operation
RTL	*RTL*	**3 bytes**—RTL address
RTL	← **SP**	

Stack before RTL



Word—Depends on operation

3 bytes—RTL address

Important
Do not change the original RECT pointed to by *rectPtr*. If you need to change something, first make a copy of the RECT and then change that copy.

❖ *Note:* The term *item number* referred to in the following sections of this document does not refer to the same thing as an *item ID number*. The *item number* can be any value, although bit 15 and bit 0 each have a specific meaning. The Menu Manager passes this value back to the definition procedure and compares the value to other item numbers returned by the same definition procedure during a previous call. The definition procedure can use any numbering system desired, but it should be the same system for each definition procedure function.

The *menuMessage* parameter identifies the operation to be performed. It has one of the values shown in Table 13-5.

Table 13-5
Menu messages

Value	Name	Description
0	`mDrawMsg`	Draw the menu
1	`mChooseMsg`	Tell which item was chosen and highlight it
2	`mSizeMsg`	Calculate the menu dimensions
3	`mDrawTitle`	Draw the menu title
4	`mDrawMItem`	Highlight or unhighlight an item
5	`mGetMItemID`	Return the item ID number

The mDrawMenu operation

The message `mDrawMsg` tells the menu definition procedure to draw the menu inside the rectangle whose RECT data structure is pointed to by *rectPtr.* The RECT passed specifies the coordinates of the menu's interior; that is, the menu minus its frame. The current GrafPort will be the Menu Manager port.

The standard menu definition procedure determines how to draw the menu items by looking in the menu record at the data that defines them. For menus of your own definition, you may set up the data defining the menu items any way you like. You should also check the *menuFlag* field of the menu record to see whether the menu is disabled (or whether any of the menu items are disabled, if you're using all the flags), and if the menu is disabled, draw it in gray. You may even print the items in a different font, so long as you restore the original font when you finish. The returned value is not used.

Stack upon entry

previous contents	
wordspace	**Word**—Space for result
menuMessage	**Word**—`mDrawMsg` message
— *theMenuHandle* —	**Long**—HANDLE to menu
— *rectPtr* —	**Long**—POINTER to RECT that defines the menu's interior
xHitPt	**Word**—Undefined
yHitPt	**Word**—Undefined
menuParam	**Word**—Undefined
RTL *RTL*	**3 bytes**—RTL address
RTL ← **SP**	

Stack before RTL

previous contents	
menuResult	**Word**—Not used; may be any value
RTL *RTL*	**3 bytes**—RTL address
RTL ← **SP**	

The mChoose operation

When the menu definition procedure receives the message `mChooseMsg`, the *yHitPt* and *xHitPt* parameters specify the mouse location in global coordinates. The procedure should determine whether the given point is within an enabled menu item. Before calling this routine, the Menu Manager checks that the point is within the rectangle whose data structure is pointed to by *rectPtr,* and that the menu is enabled.

If the mouse location is in an enabled menu item, return the item number in *menuResult,* with the high-order bit of the item set. If the mouse location isn't in an enabled item, return 0.

Stack upon entry

previous contents		
wordspace		**Word**—Space for result
menuMessage		**Word**—`mChooseMsg`
-- *theMenuHandle* --		**Long**—HANDLE to menu
-- *rectPtr* --		**Long**—POINTER to RECT specifying rectangle enclosing menu
xHitPt		**Word**—X coordinate of point to check
yHitPt		**Word**—Y coordinate of point to check
menuParam		**Word**—Undefined
RTL	*RTL*	**3 bytes**—RTL address
RTL	← **SP**	

Stack before RTL

previous contents		
menuResult		**Word**—0 if no item at point; otherwise, item number with high bit set
RTL	*RTL*	**3 bytes**—RTL address
RTL	← **SP**	

The mSize operation

The message `mSizeMsg` tells the menu definition procedure to calculate the horizontal and vertical dimensions of the menu. The menu width should be computed and stored in the *menuWidth* field of the menu record if *menuWidth* was 0 on entry; the same is true for *menuHeight*. By the same token, don't replace the value in the *menuWidth* field if it is nonzero on entry or the value in the *menuHeight* field if it is nonzero on entry.

Stack upon entry

previous contents	
wordspace	**Word**—Space for result
menuMessage	**Word**—`mSizeMsg`
-- *theMenuHandle* --	**Long**—HANDLE of menu
-- *rectPtr* --	**Long**—Undefined
xHitPt	**Word**—Undefined
yHitPt	**Word**—Undefined
menuParam	**Word**—Undefined
RTL *RTL*	**3 bytes**—RTL address
RTL ← **SP**	

Stack before RTL

previous contents	
menuResult	**Word**—Undefined
RTL *RTL*	**3 bytes**—RTL address
RTL ← **SP**	

The mDrawTitle operation

When the menu definition procedure receives the message `mDrawTitle`, the title of the menu must be drawn. The *menuParam* is as follows:

$0000	The title should be completely drawn; called this way from DrawMenuBar
$0001	The title should be drawn as highlighted without special highlighting
$800x	Negative value to use special highlighting

If the menu's *menuFlag* indicates that XOR highlighting is not to be performed, *menuParam* will never be negative. Your definition procedure must define what is special highlighting. In the standard definition procedure, the title rectangle is `XORed`.

Return FALSE in *menuResult* to have the Menu Manager draw the title (the *titleName* field of the menu record must contain a pointer to a text string for the title); otherwise, return TRUE.

Stack upon entry

previous contents	
wordspace	**Word**—Space for result
menuMessage	**Word**—`mDrawTitle`
-- *theMenuHandle* --	**Long**—HANDLE to menu
-- *rectPtr* --	**Long**—POINTER to RECT of rectangle enclosing title area
xHitPt	**Word**—Undefined
yHitPt	**Word**—Undefined
menuParam	**Word**—0 = draw normal, 1 = draw inverted, bit 15 set to invert
RTL *RTL*	**3 bytes**—RTL address
RTL ← **SP**	

Stack before RTL

previous contents	
menuResult	**Word**—FALSE to draw default title, TRUE if definition procedure drew title
RTL *RTL*	**3 bytes**—RTL address
RTL ← **SP**	

The mDrawMItem operation

The `mDrawMItem` command is a request to draw an item in its highlighted or unhighlighted state. If *menuParam* is positive, it is the item number and should be drawn unhighlighted. If *menuParam* is negative, it should be drawn highlighted. Bits 14 to 0 of *menuParam* are the same as the item number returned by the mChoose operation of your definition procedure.

Stack upon entry

previous contents		
wordspace		**Word**—Space for result
menuMessage		**Word**—`mDrawMItem`
-- *theMenuHandle* --		**Long**—HANDLE to menu
-- *rectPtr* --		**Long**—POINTER to RECT of rectangle enclosing menu
xHitPt		**Word**—Undefined
yHitPt		**Word**—Undefined
menuParam		**Word**—Item number; high bit set to highlight, clear for unhighlighted
RTL	*RTL*	**3 bytes**—RTL address
RTL	← **SP**	

Stack before RTL

previous contents		
menuResult		**Word**—Undefined
RTL	*RTL*	**3 bytes**—RTL address
RTL	← **SP**	

The mGetItemID operation

The *menuParam* parameter equals the item's number, and the definition procedure is asked to return the item ID number. The item number is the value returned by the mChoose operation with the high-order bit masked off.

Stack upon entry

previous contents		
wordspace		**Word**—Space for result
menuMessage		**Word**—`mGetItemID`
-- *theMenuHandle* --		**Long**—HANDLE to menu
-- *rectPtr* --		**Long**—Undefined
xHitPt		**Word**—Undefined
yHitPt		**Word**—Undefined
menuParam		**Word**—Item number
RTL	*RTL*	**3 bytes**—RTL address
RTL	← **SP**	

Stack before RTL

previous contents		
menuResult		**Word**—Item ID number
RTL	*RTL*	**3 bytes**—RTL address
RTL	← **SP**	

$010F MenuBootInit

Initializes the Menu Manager; called only by the Tool Locator.

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 Call must not be made by an application.

$020F MenuStartUp

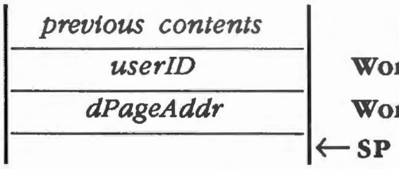
Starts up the Menu Manager for use by an application by taking the following actions:

- Making a system menu bar with no menus the current menu bar
- Calling Desktop in the Window Manager to reserve space for the menu bar
- Opening a GrafPort
- Calling DrawMenuBar to draw the empty system menu bar

Important
Your application must make this call before it makes any other Menu Manager calls.

Parameters

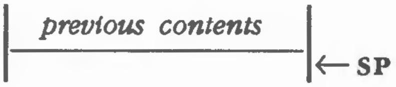
Stack before call



Word—ID number of the application

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

Stack after call



Errors

None

C

```
extern pascal void MenuStartUp(userID,dPageAddr)

Word      userID;

Word      dPageAddr;
```

$030F MenuShutDown

Shuts down the Menu Manager when an application quits. The routine closes the Menu Manager's port and frees any allocated menus.

Important

If your application has started up the Menu Manager, the application must make this call before it quits.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

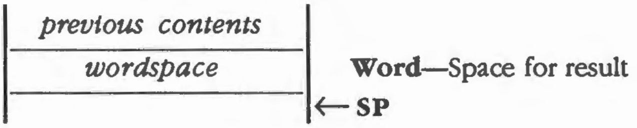
```
extern pascal void MenuShutDown()
```

$040F MenuVersion

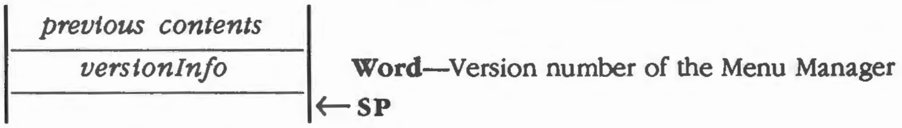
Returns the version number of the Menu Manager.

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal Word MenuVersion()
```

$050F MenuReset

Resets the Menu Manager; called 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

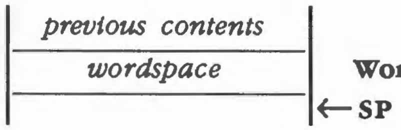
C Call must not be made by an application.

$060F MenuStatus

Indicates whether the Menu Manager is active.

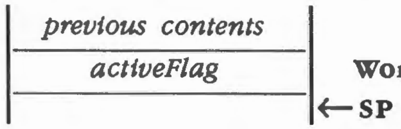
Parameters

Stack before call



Word—Space for result

Stack after call



Word—BOOLEAN; TRUE if Menu Manager active, FALSE if inactive

Errors None

C

```
extern pascal Boolean MenuStatus()
```

$1C0F CalcMenuSize

Sets menu dimensions, either manually or automatically. The Menu Manager will calculate the width if *newWidth* is 0 and the height if *newHeight* is 0.

To compute the width, the Menu Manager finds the widest item in the menu and then adds room for a mark and a command key. A default width is used if the menu does not contain any item text.

To compute the height, the Menu Manager adds together the font height of each item and then adds 4, or uses the value found in the font index of *itemFlag* if bit 14 of *itemFlag* is set.

❖ *Note:* When FixMenuBar is called, the Menu Manager calls CalcMenuSize for each menu with a negative *newWidth* and *newHeight*.

Parameters

Stack before call

previous contents	
newWidth	**Word**—Menu width in pixels, or 0 for automatic calculation
newHeight	**Word**—Menu height in pixels, or 0 for automatic calculation
menuNum	**Word**—ID number of the menu whose width and height will be calculated
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

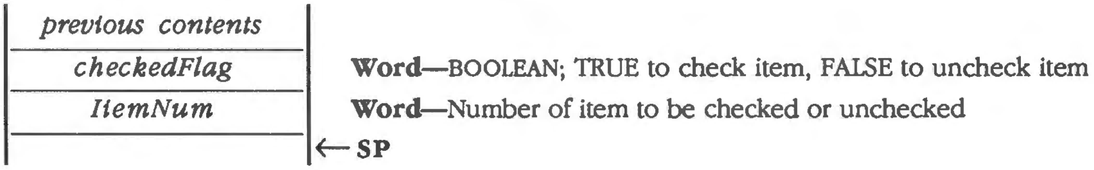
```
extern pascal void CalcMenuSize(newWidth,newHeight,menuNum)

Word      newWidth;

Word      newHeight;

Word      menuNum;
```

$320F CheckMItem

Sets a specified menu item to display or to not display a check mark to the left of the item.

Parameters

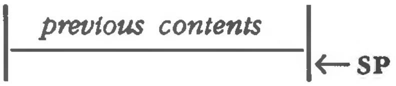
Stack before call



Word—BOOLEAN; TRUE to check item, FALSE to uncheck item

Word—Number of item to be checked or unchecked

Stack after call



Errors

None

C

```
extern pascal void CheckMItem(checkedFlag,itemNum)

Boolean     checkedFlag;

Word     itemNum;
```

$140F CountMItems

Returns the number of items, including any dividing lines, in a specified menu.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
menuNum	**Word**—Menu ID of menu whose items will be counted
	← **SP**

Stack after call

previous contents	
numberItems	**Word**—Number of items in the menu
	← **SP**

Errors

None

C

```
extern pascal Word CountMItems(menuNum)

Word     menuNum;
```

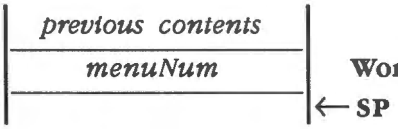
$0E0F DeleteMenu

Removes a specified menu from the menu list. The memory for the menu is not deallocated; if you want to deallocate the memory, call the DisposeMenu routine.

You should also call the DrawMenuBar routine to redraw the new menu bar.

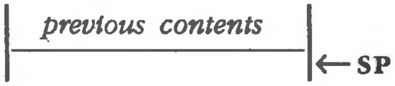
Parameters

Stack before call



Word—ID of menu to be deleted

Stack after call



Errors

None

C

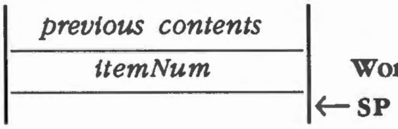
```
extern pascal void DeleteMenu(menuNum)

Word     menuNum;
```

$100F DeleteMItem

Removes a specified item from the current menu. Call CalcMenuSize after DeleteMItem to resize the menu if necessary.

Parameters

Stack before call

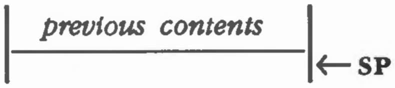

previous contents

itemNum

Word—Item ID of item to be deleted

← **SP**

Stack after call


previous contents

← **SP**

Errors

None

C

```
extern pascal void DeleteMItem(itemNum)

Word     itemNum;
```

$310F DisableMItem

Sets a specified item to display in dimmed characters and does not allow it to be selected.

Parameters

Stack before call

previous contents	
itemNum	**Word**—Number of item to be disabled
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

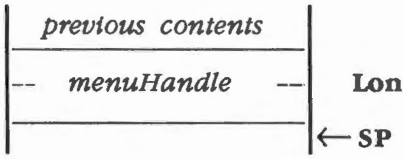
```
extern pascal void DisableMItem(itemNum)

Word     itemNum;
```

$2E0F DisposeMenu

Disposes of the memory allocated for a specified menu. The menu list will no longer be usable. For more information about how memory is disposed of, see Chapter 12, "Memory Manager."

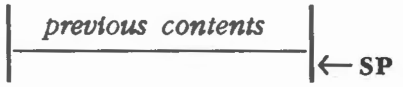
Parameters

Stack before call



Long—HANDLE to menu list to be discarded

Stack after call



Errors

None

C

```
extern pascal void DisposeMenu(menuHandle)

CtlRecHndl     menuHandle;
```

Assembly-language example

To delete a menu from the menu list and free its memory, you can enter something like this:

```
pha                                      ; Space for returned handle
pha
pea           MenuID                     ; ID of menu to delete
_GetMHandle                              ; Get the handle of the menu
                                         ; Leave menu handle on stack

pea           MenuID                     ; ID of menu to delete from list
_DeleteMenu                              ; Delete menu from list
                                         ; Handle still on stack
_DisposeMenu                             ; Deallocate menu record
```

$2A0F DrawMenuBar

Draws the current menu bar, along with any menu titles on the bar.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

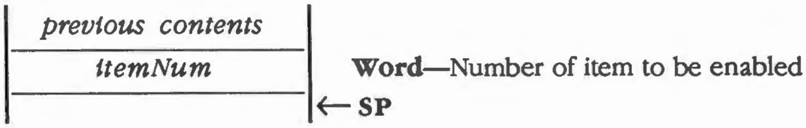
```
extern pascal void DrawMenuBar()
```

$300F EnableMItem

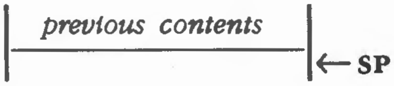
Sets a specified item to display normally and allows it to be selected.

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal void EnableMItem(itemNum)
Word     itemNum;
```

$130F FixMenuBar

Computes standard sizes for the menu bar and menus.

The routine searches all the menu title fonts and uses the tallest one to compute the height of the menu bar, adds that value to the top of the bar, and stores that value in the bottom of the bar. It also sets the *titleWidth* field in the menu record for every *titleWidth* specified as 0 and calls CalcMenuSize for each menu in the menu bar.

Parameters

Stack before call

```
|  previous contents  |
|---------------------|
|      wordspace      |    Word—Space for result
|---------------------|
|                     |<— SP
```

Stack after call

```
|  previous contents  |
|---------------------|
|      menuHght       |    Word—Height of the menu bar
|---------------------|
|                     |<— SP
```

Errors

None

C

```
extern pascal Word FixMenuBar()
```

$0C0F FlashMenuBar

Flashes the entire current menu bar by first redrawing it using the colors specified as *newInvertColor* by the SetBarColors routine and then redrawing it again using normal colors.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

None

C

```
extern pascal void FlashMenuBar()
```

$180F GetBarColors

Returns the colors for the current menu bar.

Parameters

Stack before call

```
|  previous contents  |
|---------------------|
|--    longspace    --|   Long—Space for result
|---------------------|
|                     |<— SP
```

Stack after call

```
|  previous contents  |
|---------------------|
|--  menuBarColor   --|   Long—Current menu bar colors (see Figure 13-10)
|---------------------|
|                     |<— SP
```

Errors

None

C

```
extern pascal LongWord GetBarColors()
```

Returned colors

The values returned in *menuBarColor* are shown in Figure 13-10.

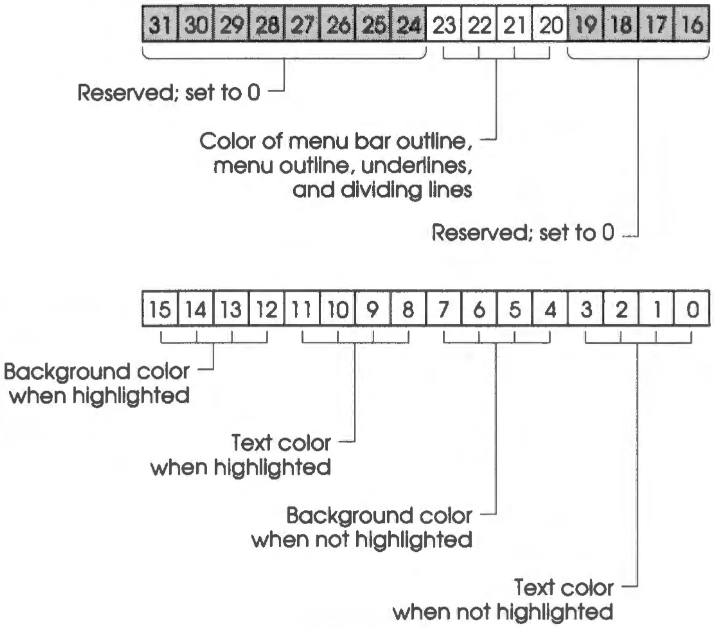

31
30
29
28
27
26
25
24
23
22
21
20
19
18
17
16
Reserved; set to 0
Color of menu bar outline, menu outline, underlines, and dividing lines
Reserved; set to 0
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Background color when highlighted
Text color when highlighted
Background color when not highlighted
Text color when not highlighted

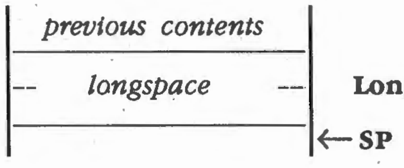
Figure 13-10
Menu bar color table for GetBarColors

$0A0F GetMenuBar

Returns the handle of the current menu bar.

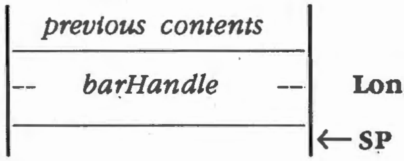
Parameters

Stack before call



Long—Space for result

Stack after call



Long—HANDLE to current menu bar

Errors

None

C

```
extern pascal CtlRecHndl GetMenuBar()
```

$200F GetMenuFlag

Returns the menu flag for a specified menu (see the section "Menu Records" in this chapter for further definition).

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
menuNum	**Word**—Number of menu whose state will be returned
	← **SP**

Stack after call

previous contents	
menuState	**Word**—Current state of menu bar *ctlFlag*
	← **SP**

Errors

None

C

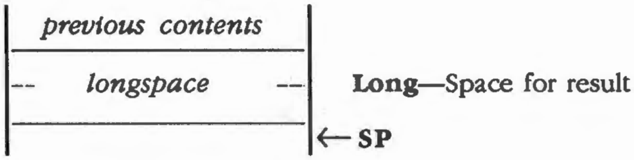
```
extern pascal Word GetMenuFlag(menuNum)

Word     menuNum;
```

$1B0F GetMenuMgrPort

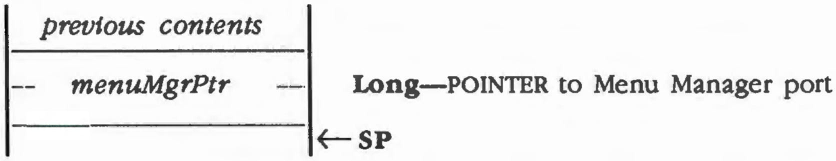
Returns a pointer to the Menu Manager's port. For example, you need the port if you want to change its font.

Parameters

Stack before call



Stack after call



Errors

None

C

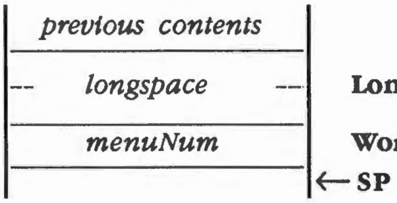
```
extern pascal GrafPortPtr GetMenuMgrPort()
```

$220F GetMenuTitle

Returns a pointer to the title of a menu.

Parameters

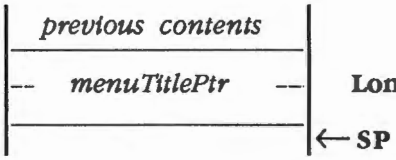
Stack before call



Long—Space for result

Word—Number of menu whose title pointer will be returned

Stack after call



Long—POINTER to the title of the menu

Errors

None

C

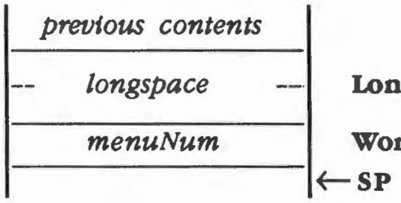
```
extern pascal Pointer GetMenuTitle(menuNum)
Word     menuNum;
```

$160F GetMHandle

Returns a handle to a menu record.

Parameters

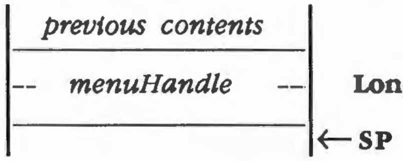
Stack before call



Long—Space for result

Word—Menu whose handle will be returned

Stack after call



Long—HANDLE to specified menu; 0 if menu not found

Errors

None

C

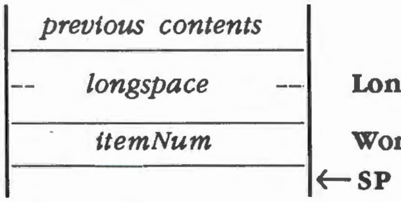
```
extern pascal CtlRecHndl GetMHandle(menuNum)

Word     menuNum;
```

$250F GetMItem

Returns a pointer to the name of an item.

Parameters

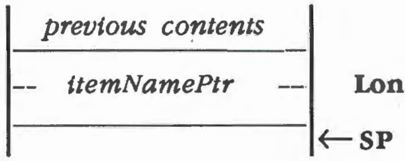
Stack before call



Long—Space for result

Word—Number of item whose name pointer will be returned

Stack after call



Long—POINTER to the name of the item

Errors

None

C

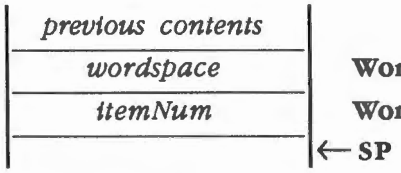
```
extern pascal Pointer GetMItem(itemNum)

Word     itemNum;
```

$270F GetMItemFlag

Returns the values for a specified item, such as whether it is disabled, underlined, or highlighted. The values are shown in Table 13-9 in the section "SetMItemFlag" in this chapter.

Parameters

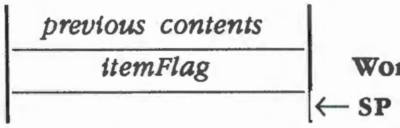
Stack before call



Word—Space for result

Word—Number of item whose flag will be returned

Stack after call



Word—Values for flag (see Table 13-9 in the section "SetMItemFlag")

Errors

None

C

```
extern pascal Word GetMItemFlag(itemNum)

Word     itemNum;
```

$340F GetMItemMark

Returns the current character that is displayed to the left of a specified menu item.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
itemNum	**Word**—Number of item whose mark character will be returned
	← **SP**

Stack after call

previous contents	
mark	**Word**—Character marking item; 0 if no mark
	← **SP**

Errors

None

C

```
extern pascal Word GetMItemMark(itemNum)
Word     itemNum;
```

$360F GetMItemStyle

Returns the text style for a specified menu item. The values are shown in Figure 13-12 in the section "SetMItemStyle" in this chapter.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
itemNum	**Word**—Number of item whose text style will be returned
	← **SP**

Stack after call

previous contents	
textStyle	**Word**—Text style (see Figure 13-12 in the section "SetMItemStyle")
	← **SP**

Errors

None

C

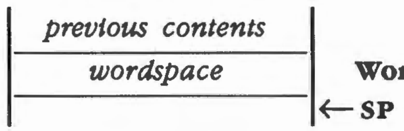
```
extern pascal TextStyle GetMItemStyle(itemNum)

Word     itemNum;
```

$1A0F GetMTitleStart

Returns the starting position for the leftmost title within the current menu bar.

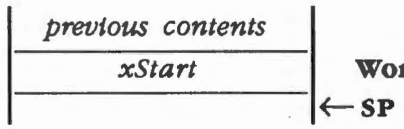
Parameters

Stack before call



Word—Space for result

Stack after call



Word—Starting position of first title in number of pixels from left

Errors

None

C

```
extern pascal Word GetMTitleStart()
```

$1E0F GetMTitleWidth

Returns the width of a menu title. The width defines the area where the user can select the menu and the area that is inverted when the title is highlighted.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
menuNum	**Word**—Number of menu whose title width will be returned
	← **SP**

Stack after call

previous contents	
currentWidth	**Word**—Width of the title in pixels
	← **SP**

Errors

None

C

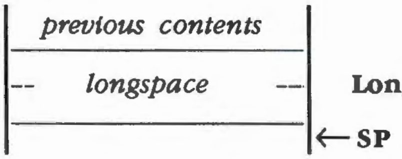
```
extern pascal Word GetMTitleWidth(menuNum)

Word     menuNum;
```

$110F GetSysBar

Returns the handle of the current system menu bar.

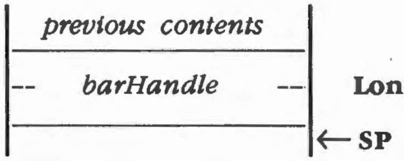
Parameters

Stack before call



Long—Space for result

Stack after call



Long—HANDLE to current system bar

Errors

None

C

```
extern pascal CtlRecHndl GetSysBar()
```

$2C0F HiliteMenu

Highlights or unhighlights the title of a specified menu. The routine should be called with *hiliteFlag* FALSE and the *menuNum* of the selected menu after your application has finished acting on a menu selection.

Parameters

Stack before call

previous contents	
hiliteFlag	**Word**—BOOLEAN; FALSE to draw normally, TRUE to highlight title
menuNum	**Word**—ID of menu
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void HiliteMenu(hiliteFlag,menuNum)

Boolean     hiliteFlag;

Word     menuNum;
```

$2F0F InitPalette

Reinitializes the palette needed for the color Apple logo in the system menu bar. The routine changes the scan-line byte for lines 2 through 9 to the first color from color tables 1 through 6.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

None

C

```
extern pascal void InitPalette()
```

$0D0F InsertMenu

Inserts a specified menu into the menu list after a specified menu or at the front of the list.

After you call this routine, you should call the DrawMenuBar routine to redraw the new menu bar.

Parameters

Stack before call

previous contents	
-- *addMenuHandle* --	**Long**—HANDLE to menu to insert
insertAfter	**Word**—Menu is inserted after this menu; if 0, menu inserted at front
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

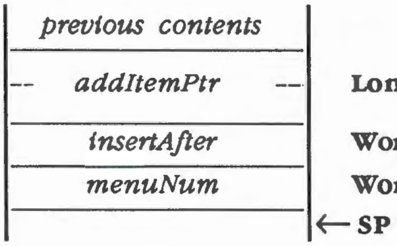
```
extern pascal void InsertMenu(addMenuHandle,insertAfter)
CtlRecHndl     addMenuHandle;
Word     insertAfter;
```

$0F0F InsertMItem

Inserts a menu item into a menu after a specified menu item or at the front of the list. Call CalcMenuSize to resize the menu, if necessary.

Parameters

Stack before call

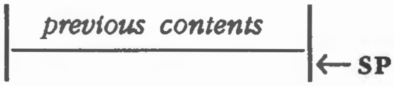


Long—POINTER to item to insert

Word—Item after which item is inserted; 0 to add to front, $FFFF at end

Word—Menu ID of menu to contain new item, 0 for first menu

Stack after call



Errors

None

C

```
extern pascal void InsertMItem(addItemPtr,insertAfter,menuNum)

Pointer     addItemPtr;

Word      insertAfter;

Word      menuNum;
```

$230F MenuGlobal

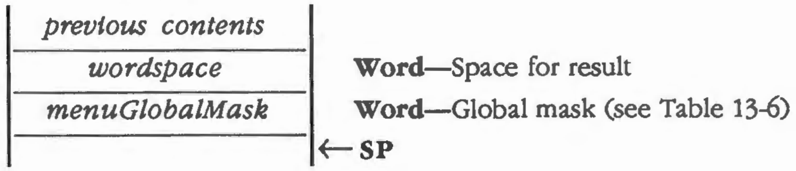
Specifies a mask that determines how the Menu Manager performs tasks. If *menuGlobalMask* has bit 15 set to 1, the mask will be ANDed with the global flag. If *menuGlobalMask* has bit 15 set to 0, the mask will be ORed with the global flag. MenuGlobal also returns the current state of the global flag in *menuGlobalFlag*.

Important
At the time of publication, all bits except 15 and 0 are reserved for future use.

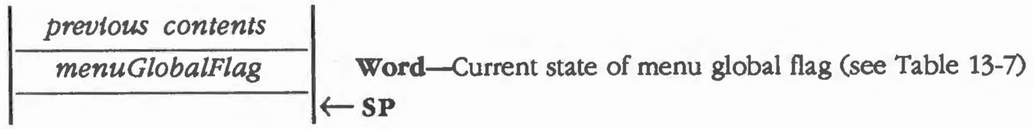
The MenuGlobal routine can use bit 0 to turn menu help on or off and return the current state of menu help. This feature allows users find out how inactive items can be made active.

Parameters

Stack before call



Stack after call



Errors

None

C

```
extern pascal Word MenuGlobal(menuGlobalMask)

Word      menuGlobalMask;
```

(continued)

Menu help

Using the standard desktop interface, a menu item is dimmed to indicate that the user can't choose the item. This presents a problem: The user wouldn't be able to select the item to determine how to make the menu item active. The MenuGlobal routine provides a menu help feature that enables an application to provide that information to the user. The values used to turn menu help on and off are shown in Table 13-6.

Table 13-6
Menu global mask values

Value	Description
$0000	Flag does not change (used to retrieve current state of flag)
$0001	Turn menu help on
$FFFE	Turn menu help off

When your application passes $0001 to the MenuGlobal routine, it indicates to the Menu Manager that the application can use the menu help feature. The Menu Manager will then

- Highlight a dimmed item when the user moves the cursor over the item
- Allow the MenuSelect routine to return the ID number of the selected, dimmed item in the high-order word of *taskData* in the task record passed to that routine

If you're using the Window Manager routine TaskMaster, you will need to set the *tmInactive* bit (bit 14) in the *taskMask* field of the task record. TaskMaster will then return `wInactMenu` ($001C) when the user chooses a dimmed menu item. The dimmed item's ID number can then be returned in the high-order word of *taskData*.

The value returned in *menuGlobalFlag* indicate the state of the flag after any changes have been made. The values at the time of publication are shown in Table 13-7.

Table 13-7
Menu global flag values

Value	Description
$0000	Menu help off
$0001	Menu help on

$090F MenuKey

Maps a character to the associated menu and item for that character. When your application receives a key-down event while the user is holding down the Apple key—or while the user is holding down the Apple key and another key, if the command being invoked can be repeated—the application should call MenuKey with the character that was typed. MenuKey highlights the appropriate menu title if the key matches and then returns the selection. The specified menu bar becomes the current menu bar.

If the user makes a selection, the low-order word of the *when* element in the event record contains ID number of the item selected, and the high-order word contains the menu's ID number. The selected menu's title will remain highlighted. Use the HiliteMenu routine to redraw the title normally. If the user doesn't make a selection, the low-order word of the *when* element in the event record will be 0.

There should generally be no more than one item in the menu list with the same keyboard equivalent; if there is, MenuKey returns the first one that it encounters.

Parameters

Stack before call

previous contents	
-- *taskRecPtr* --	**Long**—POINTER to task record containing character to check
-- *barHandle* --	**Long**—HANDLE to menu bar, or 0 for system menu bar
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void MenuKey(taskRecPtr,barHandle)

WmTaskRecPtr     taskRecPtr;

CtlRecHndl     barHandle;
```

(continued)

Keyboard equivalents and the MenuKey routine

For the returned key, bit 8 should be set. The application can then easily determine whether the Apple key was held down when the character key was pressed. When the Event Manager returns a key-down message, you can clear the low-order byte of the event modifier flag and then OR it with the event message (where the key is). Then MenuKey will check the flag for you.

Any lowercase key is changed to uppercase before being used. MenuKey also makes some other changes to accommodate the way the key is packed into the item's record.

$290F MenuNewRes

Adjusts screen resolution and redraws the current system menu bar. Call this routine when the screen resolution changes.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void MenuNewRes()
```

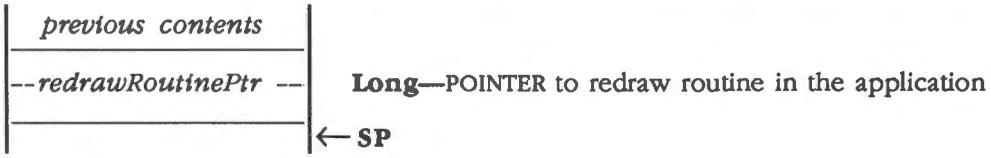
$0B0F MenuRefresh

Called when the application is not using the Window Manager, and the Menu Manager cannot restore the screen under a menu. In the attempt to recover, the Menu Manager takes the following steps:

1. It tries to allocate a buffer large enough to save the screen before it draws the menu.
2. If the buffer is allocated, the screen will be restored from it, and then the memory buffer will be deallocated. If the buffer cannot be allocated, the Menu Manager tries to call the Window Manager (via the call the Window Manager made to MenuRefresh during initialization) to refresh the screen when the menu goes away.
3. If no buffer can be allocated and the Window Manager isn't installed, the Menu Manager will call the routine pointed to by *redrawRoutinePtr* to refresh the screen under the menu.

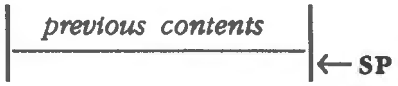
Parameters

Stack before call



Long—POINTER to redraw routine in the application

Stack after call



Errors

None

C

```
extern pascal void MenuRefresh(redrawRoutinePtr)

VoidProcPtr     redrawRoutinePtr;
```

Redraw routine example

The Redraw routine that MenuRefresh calls should look something like this:

```
              lda #$90
Refresh       START
;
rect_addr     equ 6                    ;Offset down stack to RECT pointer
              .
              .                        ;(These would be any needed operations
              .                        ; to redraw the screen inside the given RECT)
;
; Remove the given pointer from the stack:
;
              lda 0,s                  ;Move the return address down
              sta 4,s                  ;the stack
              lda 2,s
              sta 6,s
;
              pla                      ;Move the stack back to the return
              pla                      ;address
;
              rtl                      ;Return to the Menu Manager
```

$2B0F MenuSelect

Draws highlighted titles, pulls down menus, and handles user interaction when a mouse button is clicked on a menu bar (see the Window Manager FindWindow routine if you're using the Window Manager). These tasks are handled automatically for the system menu bar when TaskMaster is used in the Window Manager.

If the user makes a selection, the low-order word of the *when* element in the event record will contain the ID number of the item selected, and the high-order word will contain the menu ID number. The selected menu's title will remain highlighted. See the section "HiliteMenu" in this chapter to redraw the title normally.

If the user doesn't make a selection, the low-order word of the *when* element in the event record will be 0.

The specified menu bar becomes the current menu bar.

Parameters

Stack before call

```
|   previous contents   |
|-----------------------|
|--   taskRecPtr     --|     Long—POINTER to task record containing button-down point
|-----------------------|
|--   barHandle      --|     Long—HANDLE to menu bar; 0 for system menu bar
|-----------------------|
|                       |<— SP
```

Stack after call

```
|   previous contents   |
|-----------------------|
|                       |<— SP
```

Errors

None

C

```
extern pascal void MenuSelect(taskRecPtr,barHandle)

WmTaskRecPtr     taskRecPtr;

CtlRecHndl     barHandle;
```

$2D0F NewMenu

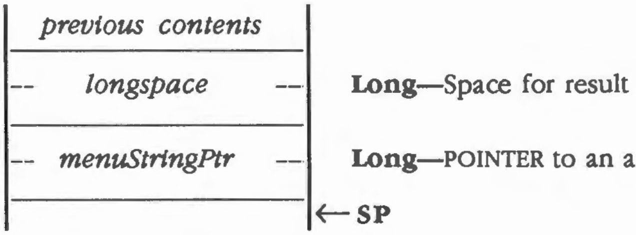
Allocates space for a menu list and its items. A menu string that describes menu titles, items, and special flags must be passed to the routine. See the section "Menu Lists: Menu Lines and Item Lines" in this chapter for the format needed.

The menu pointed to by *menuStringPtr* can then be inserted in the default system menu bar by an InsertMenu call.

Call DisposeMenu to deallocate the menu list when you are finished. To set the colors of the menu bar, see the section "SetBarColor" in this chapter.

Parameters

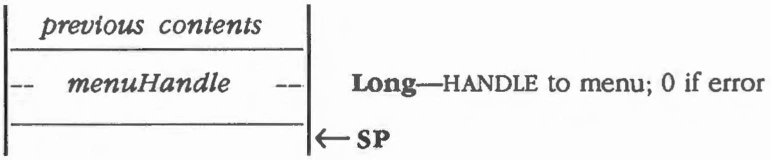
Stack before call



Long—Space for result

Long—POINTER to an array of menu lines and item lines

Stack after call



Long—HANDLE to menu; 0 if error

Errors

None

C

```
extern pascal CtlRecHndl NewMenu(menuStringPtr)

Pointer     menuStringPtr;
```

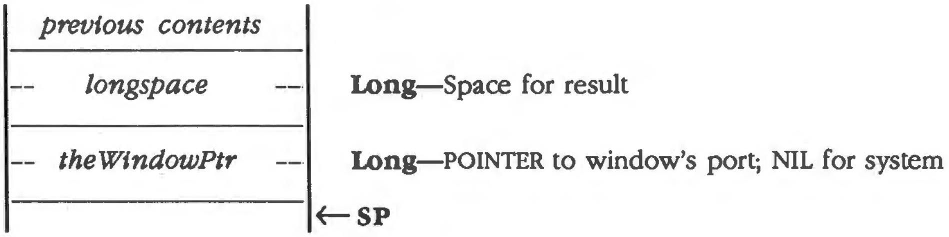
$150F NewMenuBar

Creates a default menu bar with no menus. MenuStartup calls NewMenuBar to create a default system menu bar. The upper-left corner of the default menu bar matches the port, and the width is the width of the screen. The height of the bar is 13 pixels. The menu bar is visible and has default colors of black text on a white background.

If you are going to use only one system menu bar, you don't have to call NewMenuBar.

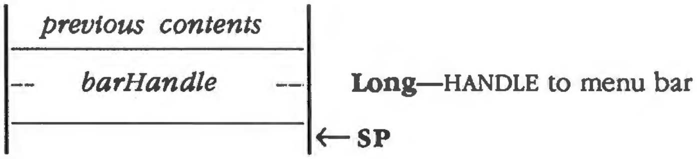
Parameters

Stack before call



previous contents
-- longspace --
-- theWindowPtr --
Long—Space for result
Long—POINTER to window's port; NIL for system
← SP

Stack after call



Errors

None

C

```
extern pascal CtlRecHndl NewMenuBar(theWindowPtr)
GrafPortPtr     theWindowPtr;
```

$170F SetBarColors

Sets the normal, inverse, and outline colors of the current menu bar. The *newInvertColor* value is the color of an item when selected; *newOutColor* is the color of the menu bar outline, menu outline, underlines, and dividing lines. Any negative values will not change the specified color.

Parameters

Stack before call

previous contents	
newBarColor	**Word**—Normal color (see Figure 13-11)
newInvertColor	**Word**—Selected color (see Figure 13-11)
newOutColor	**Word**—Outline color (see Figure 13-11)
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

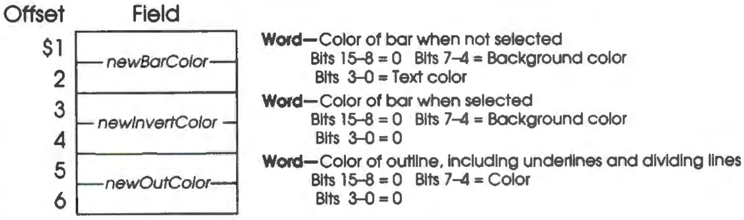
C

```
extern pascal void SetBarColors(newBarColor,newInvertColor,newOutColor)

Word     newBarColor;

Word     newInvertColor;

Word     newOutColor;
```

(continued)

Bar colors

Figure 13-11 illustrates the menu bar color table.



Word—Color of bar when not selected
Bits 15–8 = 0 Bits 7–4 = Background color
Bits 3–0 = Text color

Word—Color of bar when selected
Bits 15–8 = 0 Bits 7–4 = Background color
Bits 3–0 = 0

Word—Color of outline, including underlines and dividing lines
Bits 15–8 = 0 Bits 7–4 = Color
Bits 3–0 = 0

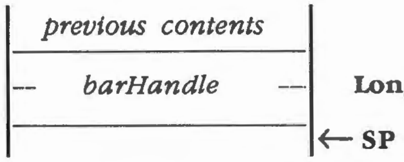
Figure 13-11
Menu bar color table for SetBarColors

$390F SetMenuBar

Sets the current menu bar. If you want the system menu bar to be the current menu bar, pass 0 for *barHandle.*

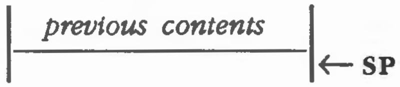
Parameters

Stack before call



Long—HANDLE to new current bar; 0 for system menu bar

Stack after call



Errors

None

C

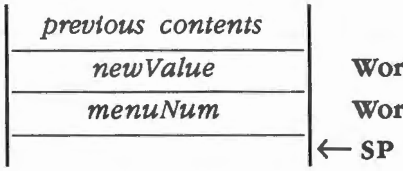
```
extern pascal void SetMenuBar(barHandle)

CtlRecHndl     barHandle;
```

$1F0F SetMenuFlag

Sets the menu to a specified state. If you change a flag that affects the appearance of a menu title, call the DrawMenuBar routine after the SetMenuFlag call.

Parameters

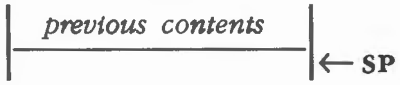
Stack before call



Word—New bit value to set (see Table 13-8)

Word—Number of menu whose state will be set

Stack after call



Errors

None

C

```
extern pascal void SetMenuFlag(newValue,menuNum)
Word     newValue;
Word     menuNum;
```

Table 13-8
Menu flag values

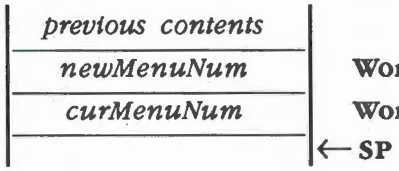
Name	Value	Description
`enableMenu`	$FF7F	Menu will not be dimmed and will be selectable
`disableMenu`	$0080	Menu will be dimmed and not selectable
`colorReplace`	$FFDF	Menu's title and background will be redrawn in highlighted state
`xorTitleHilite`	$0020	Menu's title will be XORed to highlighted state
`standardMenu`	$FFEF	Menu will be considered a standard menu
`customMenu`	$0010	Menu will be considered a custom menu

$370F SetMenuID

Specifies a new menu number.

Parameters

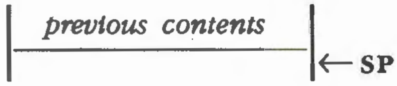
Stack before call



Word—New menu number for the menu

Word—Current menu number whose ID will be changed

Stack after call



Errors

None

C

```
extern pascal void SetMenuID(newMenuNum,curMenuNum)
Word     newMenuNum;
Word     curMenuNum;
```

$210F SetMenuTitle

Specifies the title for a menu.

Parameters

Stack before call

previous contents	
-- *newStrPtr* --	**Long**—POINTER to string to become new title
menuNum	**Word**—Number of menu whose title will be set
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetMenuTitle(newStrPtr,menuNum)

Pointer     newStrPtr;

Word     menuNum;
```

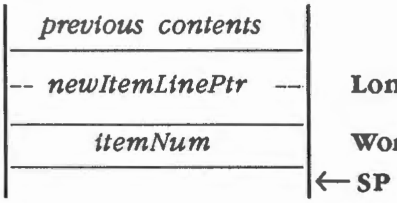
$240F SetMItem

Specifies the name for a menu item by pointing to an item line. The menu item with *itemNum* has its pointer replaced by *newItemLinePtr*. All other parameters of the item, such as ID number, text style, marked status, and keyboard equivalent, remain the same.

Important
The string pointed to by *newStrPtr* must remain in memory at that address.

Parameters

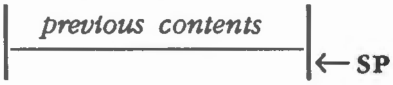
Stack before call



Long—POINTER to an item line

Word—Number of item whose name will be set

Stack after call



Errors

None

C

```
extern pascal void SetMItem(newItemLinePtr,itemNum)

Pointer     newItemLinePtr;

Word     itemNum;
```

Assembly-language example

An example of what *newItemLinePtr* should point to is as follows:

```
NewLine        dc      c'--New Name',il'0'
```

SetMItem replaces the second dash with the length of the name, and it replaces the item's name pointer with a pointer to the length byte, which produces a pointer to a Pascal string.

$280F SetMItemBlink

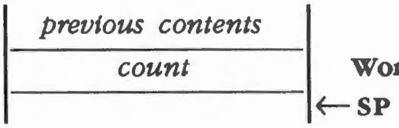
Determines how many times menu items should blink when selected.

When an enabled item is selected by the user, the item blinks briefly to confirm the choice. Normally, your application should simply allow the user to set the rate of blinking by using the Control Panel. However, if you're writing a desk accessory, you might wish to control the duration of the blinking.

SetMItemBlink affects all system and window menu bars.

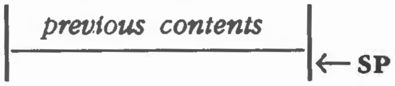
Parameters

Stack before call



Word—Number of times any item should blink when selected

Stack after call



Errors

None

C

```
extern pascal void SetMItemBlink(count)

Word     count;
```

$260F SetMItemFlag

Sets a specified item number to be underlined or not underlined and sets the highlighting style.

Parameters

Stack before call

previous contents	
newValue	**Word**—New bits to set (see Table 13-9)
itemNum	**Word**—Number of item whose text style will be set
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetMItemFlag(newValue,itemNum)

Word      newValue;

Word      itemNum;
```

(continued)

Item flag values

Table 13-9 shows the values for the item flags.

Table 13-9
Item flag values

Name	Value	Description
`underMItem`	$0040	Underline item
`noUnderMItem`	$FFBF	Don't underline item
`xorMItemHilite`	$0020	Use XOR highlighting
`colorMItemHilite`	$FFDF	Use redraw highlighting
`enableItem`	$FF7F	Enable menu item
`disableItem`	$0080	Disable menu item

Important

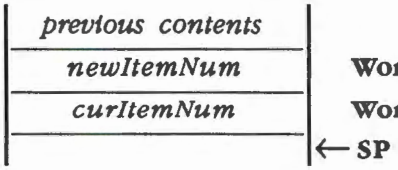
Fonts that have a descent value of less than 2, such as Shaston 8 (the system font at the time of publication), will not be underlined.

$380F SetMItemID

Specifies the ID number of a menu item.

Parameters

Stack before call



previous contents

newItemNum **Word**—New ID number to be assigned to the item

curItemNum **Word**—Current item whose ID will be changed

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

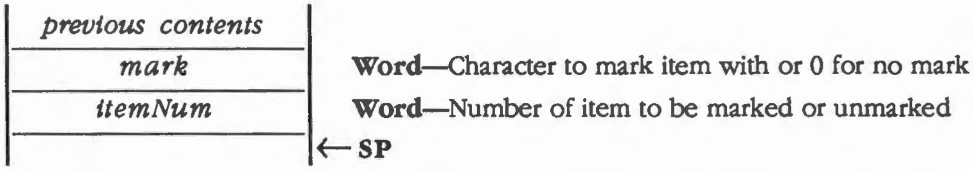
```
extern pascal void SetMItemID(newItemNum,curItemNum)
Word     newItemNum;
Word     curItemNum;
```

$330F SetMItemMark

Sets a menu item to display or not display a specified character to the left of the item. The character will appear to the left of the item's text; it will not appear, or will be removed if it already appears, if ***markItem*** is 0.

Parameters

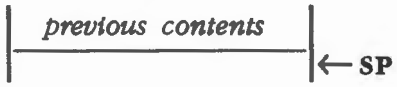
Stack before call



Word—Character to mark item with or 0 for no mark

Word—Number of item to be marked or unmarked

Stack after call



Errors

None

C

```
extern pascal void SetMItemMark(mark,itemNum)

Word     mark;

Word     itemNum;
```

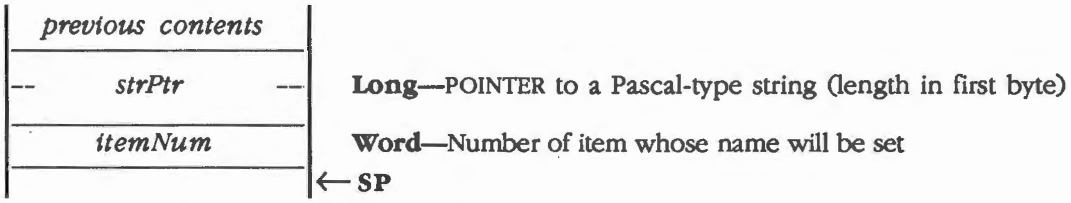
$3A0F SetMItemName

Specifies the name of a menu item by pointing to a Pascal string. The menu item with *itemNum* will use the string pointed to by *strPtr* whenever the menu is drawn. All other parameters of the item, such as ID number, text style, marked status, and keyboard equivalent, remain the same.

Important
The string pointed to by *strPtr* must remain in memory at that address.

Parameters

Stack before call



Long—POINTER to a Pascal-type string (length in first byte)

Word—Number of item whose name will be set

Stack after call

previous contents
← SP

Errors

None

C

```
extern pascal void SetMItemName(strPtr,itemNum)

Pointer    strPtr;

Word     itemNum;
```

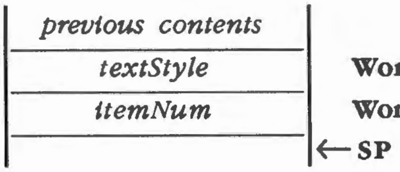
$350F SetMItemStyle

Sets the text style for a specified menu item.

If you need to change only one of the characteristics, call the GetMItemStyle routine and use the current states for the attributes that should stay the same.

Parameters

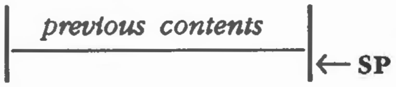
Stack before call



Word—Text style (see Figure 13-12)

Word—Number of item whose text style will be set

Stack after call



Errors

None

C

```
extern pascal void SetMItemStyle(textStyle,itemNum)
TextStyle     textStyle;
Word     itemNum;
```

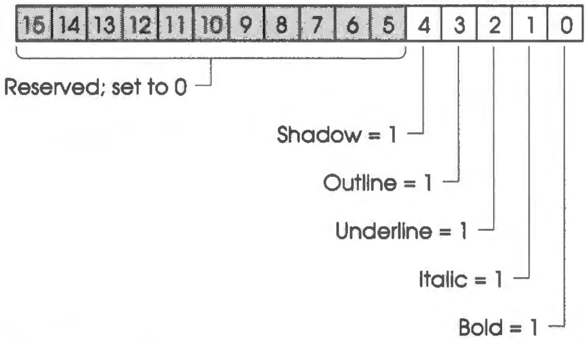


Figure 13-12
Menu text style word

Important

Shadow, outline, and italic styles are available only if QuickDraw II Auxiliary has been loaded and started up. Also, fonts that have a descent value of less than 2, such as Shaston 8 (the system font at the time of publication), will not be underlined.

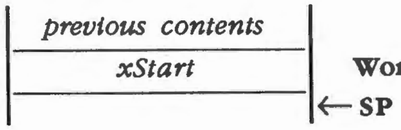
$190F SetMTitleStart

Sets the starting position for the leftmost title within the current menu bar.

For square-cornered menu bars, *xStart* should be at least 1 (0 overwrites the left line of the menu bar).

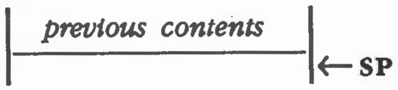
Parameters

Stack before call



Word—Starting position of first title from left in pixels (0–127)

Stack after call



Errors

None

C

```
extern pascal void SetMTitleStart(xStart)

Word     xStart;
```

$1D0F SetMTitleWidth

Sets the width of a title in pixels. The title width defines the area in which the user can select the menu and the area that is inverted when the title is highlighted.

Parameters

Stack before call

previous contents	
newWidth	**Word**—Width of title in pixels
menuNum	**Word**—Number of menu whose width will be set
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

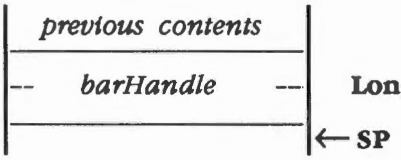
```
extern pascal void SetMTitleWidth(newWidth,menuNum)

Word     newWidth;

Word     menuNum;
```

$120F SetSysBar

Sets a new system bar. The new system menu bar becomes the current menu bar.

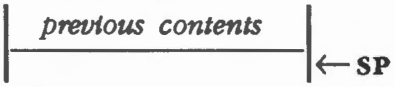
Parameters

Stack before call



Long—HANDLE to new system bar

Stack after call



Errors

None

C

```
extern pascal void SetSysBar(barHandle)

CtlRecHndl     barHandle;
```

Menu Manager summary

This section briefly summarizes the constants and data structures contained in the Menu Manager. There are no tool set error codes for the Menu Manager.

Important
These definitions are provided in the appropriate interface file.

Table 13-10
Menu Manager constants

Name	Value	Description
Masks for MenuFlag		
`mInvis`	$0004	Set if menu is not visible
`mCustom`	$0010	Set if menu is a custom menu
`mXor`	$0020	Set if menu title is highlighted using XOR
`mSelected`	$0040	Set if menu title is highlighted
`mDisabled`	$0080	Set if menu is disabled
Messages to menu definition procedures		
`mDrawMsg`	$0000	Draw menu
`mChooseMsg`	$0001	Hit test item
`mSizeMsg`	$0002	Compute menu size
`mDrawTitle`	$0003	Draw menu's title
`mDrawMItem`	$0004	Draw item
`mGetMItemID`	$0005	Return item ID number
Inputs to the SetMenuFlag routine		
`customMenu`	$0010	Menu will be considered a custom menu
`disableMenu`	$0080	Menu will be dimmed and not selectable
`enableMenu`	$FF7F	Menu will not be dimmed and will be selectable
`colorReplace`	$FFDF	Menu title and background will be redrawn highlighted
`standardMenu`	$FFEF	Menu will be considered a standard menu
Inputs to the SetMItemFlag routine		
`xorMItemHilite`	$0020	Menu title area will be XORed to highlighted
`underMItem`	$0040	Underline item
`noUnderMItem`	$FFBF	Don't underline item
`colorMItemHilite`	$FFDF	Menu title and background will be redrawn highlighted
`disableItem`	$0080	Disable menu item
`enableItem`	$FF7F	Enable menu item

Table 13-11
Menu Manager data structures

Name	Offset	Type	Definition
MENU (Menu record)			
menuID	$00	Word	Menu ID number
menuWidth	$02	Word	Width of menu
menuHeight	$04	Word	Height of menu
menuProc	$06	Pointer	Menu definition procedure
menuFlag	$0A	Byte	Bit flags
menuRes	$0B	Byte	Reserved for internal use
firstItem	$0C	Byte	Reserved for future use
numOfItems	$0D	Byte	Reserved for future use
titleWidth	$0E	Word	Width of menu title
titleName	$10	Pointer	Menu title

Note: The actual assembly-language equates have a lowercase *o* (the letter) in front of all of the names given in this table.

Chapter 14

Miscellaneous Tool Set

The **Miscellaneous Tool Set** is comprised of several small tool sets. Most of these tool sets deal with various low-level functions of the Apple IIGS; in fact, several other tool sets make calls to the Miscellaneous Tool Set.

Each of the small tool sets is discussed separately in the section "Using the Miscellaneous Tool Set" in this chapter.

A preview of the Miscellaneous Tool Set routines

Table 14-1 lists the functions of all of the Miscellaneous Tool Set routines in the order in which they appear later in this chapter.

❖ *Note:* Because the Miscellaneous Tool Set is actually a collection of small tool sets, it does not lend itself to the alphabetical routine order organization used in the rest of the chapters of this reference.

Table 14-1
Miscellaneous Tool Set routines and their functions

Routine	Description
Housekeeping routines	
MTBootInit	Initializes the Miscellaneous Tool Set; called only by the Tool Locator—must not be called by an application
MTStartUp	Starts up the Miscellaneous Tool Set for use by an application
MTShutDown	Shuts down the Miscellaneous Tool Set
MTVersion	Returns the Miscellaneous Tool Set version number
MTReset	Resets the Miscellaneous Tool Set; called only when the system is reset—must not be called by an application
MTStatus	Indicates whether the Miscellaneous Tool Set is active
Battery RAM routines	
WriteBRam	Writes 252 bytes of data from a specified memory location, plus 4 checksum bytes, to the Battery RAM
ReadBRam	Reads 252 bytes of data from the Battery RAM, plus 4 checksum bytes, and writes the data into a specified memory location
WriteBParam	Writes data to a specified parameter in Battery RAM
ReadBParam	Reads data from a specified parameter in Battery RAM
Clock routines	
ReadTimeHex	Returns current time in hexadecimal format
WriteTimeHex	Sets the current time in hexadecimal format
ReadAsciiTime	Reads elapsed time since 00:00:00, January 1, 1904, converts the elapsed time to ASCII time output, and places the output at the specified address
Firmware entry routine	
FWEntry	Allows some Apple II emulation-mode entry points to be supported from full native mode
Get address routine	
GetAddr	Returns an address of a byte, word, or long parameter referenced by the firmware
Tick counter routine	
GetTick	Returns the current value of the tick counter
Interrupt control routines	
GetIRQEnable	Returns with the hardware interrupt enable states for interrupt sources that can be controlled by the Miscellaneous Tool set
IntSource	Enables or disables certain interrupt sources

Table 14-1 (continued)
Miscellaneous Tool Set routines and their functions

Routine	Description
Mouse and absolute clamp routines	
ClampMouse	Sets clamp values to new values and then sets the mouse position to the minimum clamp values
ClearMouse	Sets the X and Y axis to $0000 if minimum clamps are negative or to the minimum clamp position if the clamps are positive
GetMouseClamp	Returns the current mouse clamp values
HomeMouse	Positions the mouse at the minimum clamp position
InitMouse	Initializes mouse clamp values to $000 minimum and $3FF maximum and clears the mouse mode and status
PosMouse	Positions the mouse at specified coordinates
ReadMouse	Returns mouse position, status, and mode
ServeMouse	Returns the mouse interrupt status
SetMouse	Sets the mouse mode
SetAbsClamp	Sets the clamp values for an absolute device to new values
GetAbsClamp	Returns the current values of the absolute device clamps
Packing and munging routines	
PackBytes	Packs bytes into a special format that uses less storage space
UnPackBytes	Unpacks data from the packed format used by PackBytes
Munger	Manipulates bytes in a string of bytes
Heartbeat routines	
SetHeartBeat	Installs a specifed task into the HeartBeat Interrupt Task queue
DelHeartBeat	Deletes a specified task from the Heartbeat Interrupt Task queue
ClrHeartBeat	Removes all tasks from the Heartbeat Interrupt Task queue by clearing the Heartbeat Task pointer
System bell routine	
SysBeep	Calls the Apple II monitor entry point BELL1
System Failure Manager routine	
SysFailMgr	Displays system failure message and halts program execution
User ID Manager routines	
GetNewID	Creates a new user ID
DeleteID	Deletes all references to a specified user ID
StatusID	Indicates whether a specified user ID is active
Vector initialization routines	
SetVector	Sets the vector address for a specified interrupt manager or handler
GetVector	Returns the vector address for a specified interrupt manager or handler

Using the Miscellaneous Tool Set

This section discusses how the Miscellaneous Tool Set routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

The Miscellaneous Tool Set depends upon the presence of the tool sets listed in Table 14-2 and requires that at least the indicated version of the tool set be present.

Table 14-2
Miscellaneous Tool Set—other tool sets required

Tool set number		Tool set name	Minimum version needed
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0

Your application should make an MTStartUp call before making any other Miscellaneous Tool Set calls.

❖ *Note:* At the time of publication, the MTStartUp call was not an absolute requirement, because the Tool Locator automatically started up the Miscellaneous Tool Set at boot time. However, you should make the call anyway to guarantee that your application remains compatible with all future versions of the system.

If your application starts up the Miscellaneous Tool Set, the application should make the MTShutDown call when the application quits.

In keeping with the flexible spirit of the Apple IIGS, many of the routines in the Miscellaneous Tool Set retrieve the address of a given parameter or return the value of an appropriate parameter so that you do not need the address.

Important
Use these calls only as directed; there is no guarantee that an address being used for something in this version will be used the same way in future versions.

You can use the WriteBRam, ReadBRam, WriteBParam, and ReadBParam routines to write and read data to and from the Battery RAM. Any data written to the Battery RAM will affect the default system configuration, which will be used the next time the system is booted.

The clock routines ReadTimeHex, ReadAsciiTime, and WriteTimeHex provide you with a way to read the current time either in hex or ASCII format or to set the current time using hex format.

Use the SetVector and GetVector routines to set or return the vector address for a specified interrupt manager or handler. For more information about interrupt handlers, see the *Apple IIGS Firmware Reference.* Similarly, the GetAddr routine returns the address of some important firmware parameters, and the FWEntry routine allows some Apple II emulation-mode entry points to be used from full native mode.

The IntSource routine allows your application to enable or disable certain interrupt sources. The GetIRQEnable routine returns the current status of those interrupts.

The HeartBeat routines allow you to install or delete tasks from the HeartBeat Interrupt Task queue. Such tasks include controlling cursor movement, posting a disk-insert event, or checking the stack.

The SysFailMgr routine allows you to customize the system failure message. Thus, if the user causes your application to fail, you can produce a message that gives the user an idea of what happened.

The mouse routines allow your application to directly control the mouse. However, the Event Manager calls these routines automatically, so most applications don't need to make the calls. If you're not using the Event Manager, you will need to use the mouse routines. Similarly, you could use the SetAbsClamp and GetAbsClamp routines to set the clamps for absolute devices, but normally won't need to, since the Event Manager automatically handles those clamps. For more information, see the section "Using Alternative Pointing Devices" in Chapter 7, "Event Manager," and the chapter about the mouse in the *Apple IIGS Firmware Reference.*

The PackBytes routine packs data to make a file smaller. This can be useful, for instance, in graphic images, which would ordinarily take up too much space on disk. UnPackBytes unpacks the data from the PackBytes format.

The Munger routine allows your application to manipulate strings easily, and the SysBeep routine causes the system speaker to beep.

$0103 MTBootInit

Initializes the Miscellaneous Tool Set; called only by the Tool Locator. The routine performs the following:

- Initializes the Heartbeat Task pointer to $00000000
- Clears the tick counter
- Sets the mouse flag to NOT FOUND
- Asks the Memory Manager for a block of memory with a length of one word for use by the User ID 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 Call must not be made by an application.

$0203 MTStartUp

Starts up the Miscellaneous Tool Set for use by an application. Your application should make an MTStartUp call before making any other Miscellaneous Tool Set calls.

❖ *Note:* At the time of publication, the MTStartUp call was not an absolute requirement, because the Tool Locator automatically started up the Miscellaneous Tool Set at boot time. However, you should make the call anyway to guarantee that your application remains compatible with all future versions of the system.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void MTStartUp()
```

$0303 MTShutDown

Shuts down the Miscellaneous Tool Set when an application quits.

Important

If your application has started up the Miscellaneous Tool Set, the application must make this call before it quits.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

None

C

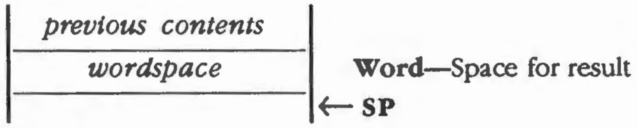
```
extern pascal void MTShutDown()
```

$0403 MTVersion

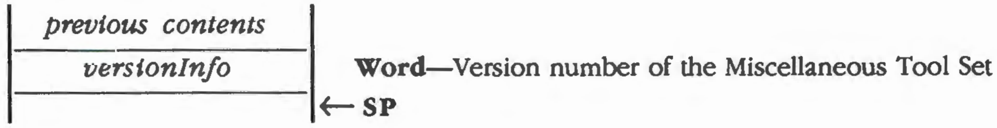
Returns the version number of the Miscellaneous Tool Set.

Parameters

Stack before call



Stack after call



Errors

None

C

```
extern pascal Word MTVersion()
```

$0503 MTReset

Resets the Miscellaneous Tool Set; called when the system is reset. Clears the Heartbeat Task pointer and sets the Mouse flag to NOT FOUND.

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

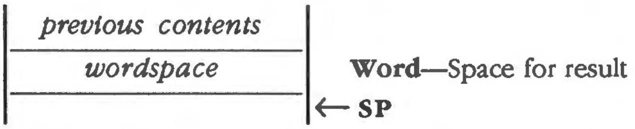
C Call must not be made by an application.

$0603 MTStatus

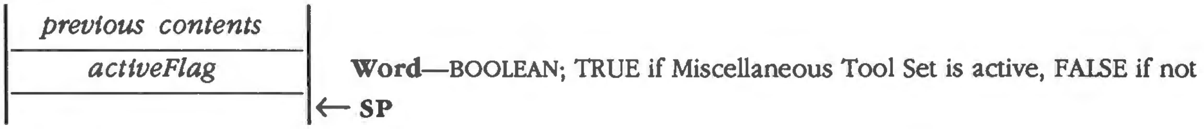
Indicates whether the Miscellaneous Tool Set is active.

Parameters

Stack before call



Stack after call



previous contents
activeFlag
Word—BOOLEAN; TRUE if Miscellaneous Tool Set is active, FALSE if not
← **SP**

Errors None

C

```
extern pascal Boolean MTStatus()
```

$0903 WriteBRam

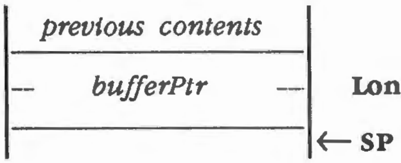
Writes 252 bytes of data from a specified memory location, plus 4 checksum bytes, to the Battery RAM.

Important

The WriteBRam routine affects the default system configuration, which will be used the next time the system is booted. The routine does not change the current system configuration.

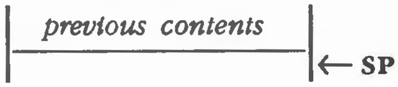
Parameters

Stack before call



Long—POINTER to the 252 bytes to be written to Battery RAM

Stack after call



Errors

None

C

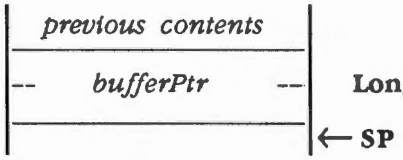
```
extern pascal void WriteBRam(bufferPtr)

Pointer     bufferPtr;
```

$0A03 ReadBRam

Reads 252 bytes of data from the Battery RAM, plus 4 checksum bytes, and writes the data into a specified memory location.

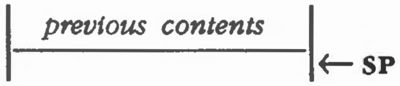
Parameters

Stack before call



Long—POINTER to 256 bytes to be written to application

Stack after call



Errors

None

C

```
extern pascal void ReadBRam(bufferPtr)

Pointer     bufferPtr;
```

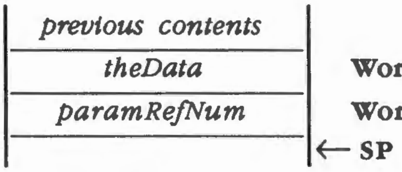
$0B03 WriteBParam

Writes data to a specified parameter in Battery RAM. The data is written to the location specified by the *paramRefNum*, as described in Table 14-3.

Important
The WriteBParam routine affects the default system configuration, which will be used the next time the system is booted. The routine does not change the current system configuration.

Parameters

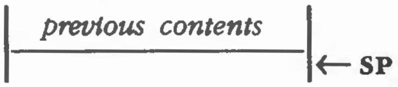
Stack before call



Word—Data to be written (low-order byte only)

Word—Parameter reference number (see Table 14-3)

Stack after call



Errors

None

C

```
extern pascal void WriteBParam(theData,paramRefNum)

Word     theData;

Word     paramRefNum;
```

Assembly-language example

```
PEA   $0005          ; Data in low byte

PEA   $0028          ; Reference number specifying startup slot

_WriteBParam
```

(continued)

Table 14-3
Battery RAM parameter reference numbers

Number	Parameter
$00	Port 1 Printer/modem
$01	Port 1 Line length
$02	Port 1 Delete line feed after carriage return
$03	Port 1 Add line feed after carriage return
$04	Port 1 Echo
$05	Port 1 Buffer
$06	Port 1 Baud
$07	Port 1 Data/stop Bits
$08	Port 1 Parity
$09	Port 1 DCD handshake
$0A	Port 1 DSR handshake
$0B	Port 1 XON/XOFF handshake
$0C	Port 2 Printer/modem
$0D	Port 2 Line length
$0E	Port 2 Delete line feed after carriage return
$0F	Port 2 Add line feed after carriage return
$10	Port 2 Echo
$11	Port 2 Buffer
$12	Port 2 Baud
$13	Port 2 Data/stop Bits
$14	Port 2 Parity
$15	Port 2 DCD handshake
$16	Port 2 DSR handshake
$17	Port 2 XON/XOFF handshake
$18	Display color/monochrome
$19	Display 40/80 column
$1A	Display text color
$1B	Display background color
$1C	Display border color
$1D	50/60 Hz
$1E	User volume
$1F	Bell volume
$20	System speed
$21	Slot 1 Internal/external
$22	Slot 2 Internal/external
$23	Slot 3 Internal/external
$24	Slot 4 Internal/external
$25	Slot 5 Internal/external
$26	Slot 6 Internal/external
$27	Slot 7 Internal/external
$28	Startup slot
$29	Text display language
$2A	Keyboard language
$2B	Keyboard buffering
$2C	Keyboard repeat speed
$2D	Keyboard repeat delay
$2E	Double click time
$2F	Flash Rate
$30	Shift caps/lowercase
$31	Fast space/delete keys
$32	Dual speed
$33	High mouse resolution
$34	Month/day/year format
$35	24/am-pm format
$36	Minimum RAM for RAM disk
$37	Maximum Ram for RAM disk
$38–40	Number of languages
$41–51	Number of layouts
$52–7F	Reserved
$80	AppleTalk node number
$81–A1	Operating system variables
$A2–FB	Reserved
$FC–FF	Checksum

$0C03 ReadBParam

Reads data from a specified parameter in Battery RAM. The data is read from the location specified by the *paramRefNum,* as described in Table 14-3 in the section "WriteBParam" in this chapter.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
paramRefNum	**Word**—Parameter reference number (see Table 14-3)
	← **SP**

Stack after call

previous contents	
theData	**Word**—Data that was read (low-order byte only)
	← **SP**

Errors

None

C

```
extern pascal word ReadBParam(paramRefNum)

Word        paramRefNum;
```

Assembly-language example

```
PEA    $0000          ; Space for result

PEA    $0028          ; Reference number specifying startup slot

_ReadBParam
```

$0D03 ReadTimeHex

Returns current time in hexadecimal format.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
wordspace	**Word**—Space for result
wordspace	**Word**—Space for result
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents		
weekDay	*null*	**Byte**—Day of week (1–7, with 1 = Sunday...) | **Byte**—Filling word
month	*day*	**Byte**—Month (0–11, with 0 = January...) | **Byte**—Date (0–30)
curYear	*hour*	**Byte**—Current year minus 1900 | **Byte**—Hour (0–23)
minute	*second*	**Byte**—Minute (0–59) | **Byte**—Second (0–59)
		← **SP**

Errors

None

C

```
extern pascal TimeRec ReadTimeHex()
```

$0E03 WriteTimeHex

Sets the current time using hexadecimal format.

❖ *Note:* The value for *curYear* cannot be 0, 1, 2, or 3.

Parameters

Stack before call

previous contents		
month | *day*	**Byte**—Month (0–11 with 0 = January...) | **Byte**—Date (0–30)	
curYear | *hour*	**Byte**—Current year minus 1900 | **Byte**—Hour (0–23)	
minute | *second*	**Byte**—Minute (0–59) | **Byte**—Second (0–59)	
	← **SP**	

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void WriteTimeHex(month,day,curYear,hour,minute,second)

Byte     month;
Byte     day;
Byte     curYear;
Byte     hour;
Byte     minute;
Byte     second;
```

Assembly-language example

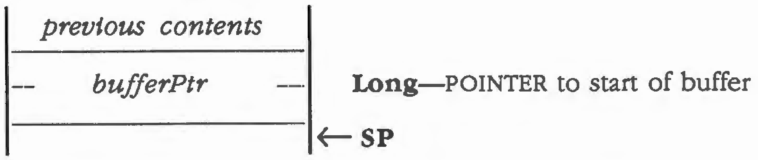
```
PEA   $0104          ; February 5th (numbers are from 0-30)
PEA   $560A          ; 1986, 10th hour
PEA   $1900          ; 25th minute, no seconds
_WriteTimeHex
```

$0F03 ReadAsciiTime

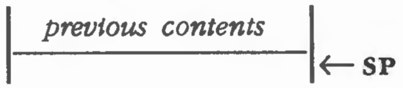
Reads elapsed time since 00:00:00, January 1, 1904, converts the elapsed time to ASCII time output, and places the output at the specified address.

Parameters

Stack before call



Stack after call



Errors

None

C

```
extern pascal void ReadAsciiTime(bufferPtr)

Pointer     bufferPtr;
```

ASCII output time

The output is always 20 characters, with the most significant byte of each character set to 1. The time format is defined by the format set up in the Battery RAM by the Control Panel, as shown in Table 14-4.

Table 14-4
ASCII time

Date format	Time format	ASCII time format		Values	
0	0	*mm/dd/yy*	*HH:MM:SS AM or PM*	*HH*	Hour
1	0	*dd/mm/yy*	*HH:MM:SS AM or PM*	*M M*	Minute
2	0	*yy/mm/dd*	*HH:MM:SS AM or PM*	*S S*	Second
0	1	*mm/dd/yy*	*HH:MM:SS*	*mm*	Month
1	1	*dd/mm/yy*	*HH:MM:SS*	*dd*	Day
2	1	*yy/mm/dd*	*HH:MM:SS*	*yy*	Year

$2403 FWEntry

Allows some Apple II emulation-mode entry points to be supported from full native mode. FWEntry preserves the state of the data bank and direct page registers before it dispatches to the firmware entry point. During the execution of the firmware task, the data bank and direct page registers are set to 0; the registers are restored on return from the firmware entry point.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
wordspace	**Word**—Space for result
wordspace	**Word**—Space for result
wordspace	**Word**—Space for result
aRegValue	**Word**—Accumulator at entry (low-order byte only)
xRegValue	**Word**—X register at entry (low-order byte only)
yRegValue	**Word**—Y register at entry (low-order byte only)
eModeEntryPt	**Word**—Emulation mode entry point
	← **SP**

Stack after call

previous contents	
status	**Word**—Processor status at exit (high-order byte only)
aRegExit	**Word**—Accumulator at exit (low-order byte only)
xRegExit	**Word**—X register at exit (low-order byte only)
yRegExit	**Word**—Y register at exit (low-order byte only)
	← **SP**

Errors

None

C

```
extern pascal FWRec FWEntry(aRegValue,xRegValue,yRegValue,eModeEntryPt)

Word      aRegValue;

Word      xRegValue;

Word      yRegValue;

Word      eModeEntryPt;
```

Assembly-language example

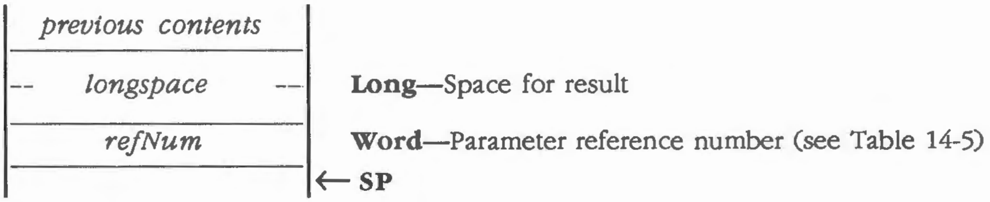
```
lda       #$0000
pha                         ; Space for result
pha                         ; Space for result
pha                         ; Space for result
pha                         ; Space for result
pea       A_Val             ; Low byte = A register on FW entry
pea       X_Val             ; Low byte = X register on FW entry
pea       Y_Val             ; Low byte = Y register on FW entry
pea       FW_Addr           ; Apple II monitor ROM entry point to be called
_FWENTRY
pla                         ; Save returned Y Reg
sta       Y_Result
pla                         ; Save returned X Reg
sta       X_Result
pla                         ; Save returned A Reg
sta       A_Result
pla                         ; Save returned P Reg
sta       P_Result
bcs       error             ; To error routine if tool returned an error
```

$1603 GetAddr

Returns an address of a byte, word, or long parameter referenced by the firmware.

Parameters

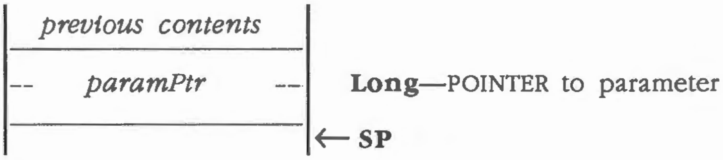
Stack before call



Long—Space for result

Word—Parameter reference number (see Table 14-5)

Stack after call



Long—POINTER to parameter

Errors

$0301	`badInputErr`	Bad input parameter

C

```
extern pascal Pointer GetAddr(refNum)

Word     refNum;
```

(continued)

Table 14-5
GetAddr parameter reference numbers

Number	Length	Parameter
$0000	Byte	Reserved
$0001	Byte	Reserved
$0002	Byte	Reserved
$0003	Byte	Reserved
$0004	Byte	Reserved
$0005	Long	Tick counter (TICKCNT)
$0006	Byte	IRQ volume (IRQ.VOLUME)
$0007	Byte	IRQ active (IRQ.ACTIVE)
$0008	Byte	IRQ sound data (IRQ.SOUNDDATA)
$0009	20 bytes	Variables after a BRK (BRK.VAR). The bytes are defined as follows: Word A register Word X register Word Y register Word Stack pointer Word Direct register Byte Processor Status Byte Data bank register Byte Emulation flag Byte Program bank register Word Program counter Byte State Byte Shadow Byte Speed register Byte MSlot
$000A	6 bytes	Event manager data (EVMGRDATA). The bytes are defined as follows: Word Journaling flag (JournalFlag) Long Pointer to journal driver (JournalPtr)
$000B	Byte	Mouse location/flag (MouseSlot). This is a flag used by the mouse calls. If MouseSlot contains a positive value, then it indicates the slot that the mouse resides in. If MouseSlot contains a negative value, the mouse has not been initialized by an InitMouse call.

Table 14-5 (continued)
GetAddr parameter reference numbers

Number	Length	Parameter
$000C	4 bytes	Mouse clamps (MouseClamps). The bytes are defined as follows: Byte Low X-axis mouse clamp Byte Low Y-axis mouse clamp Byte High X-axis mouse clamp Byte High Y-axis mouse clamp **Important** Do not set the mouse values directly. Use the mouse routines to correctly set the clamps.
$000D	4 bytes	Absolute clamps (AbsClamps). The bytes are defined as follows: Byte Low X-axis absolute device clamp Byte Low Y-axis absolute device clamp Byte High X-axis absolute device clamp Byte High Y-axis absolute device clamp Absolute device drivers are responsible for clamping the device position within the absolute device bounds.
$000E	Byte	Serial Communications Controller (SCC) interrupt flag. You can use this reference number to set a system interrupt flag byte called SerFlag and handle interrupts yourself. Do so by taking the following steps: 1. Set your bank $00 handling address to $3FE and $3FF. 2. Check the version of the firmware by making an emulation call to $00FE1F. If the version is $0, the address of the SerFlag byte is $E10104. If the version is greater than $0, retrieve the address of SerFlag by making a GetAddr call with this parameter reference number ($000E). 3. Preserve the SerFlag byte's current value. 4. Turn on the bits in the SerFlag byte to reflect the port you're using to handle interrupts, as follows: Port 1 ORA #%00111000 Port 2 ORA #%00000111 5. When you no longer wish to handle interrupts from that port (such as at application shutdown), restore the byte to its original value.

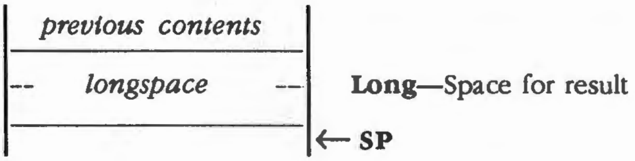
$2503 GetTick

Returns the current value of the tick counter. The value will be incremented only if the Heartbeat Interrupt Handler is installed (always true if the Event Manager is active) and VBL interrupts are enabled.

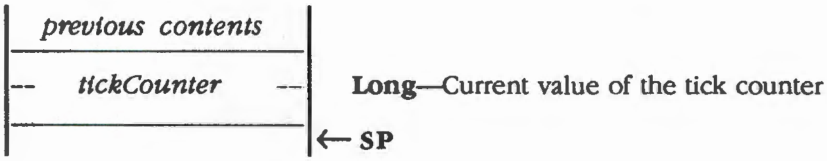
If you want your application to use the GetTick routine without activating the Event Manager, install at least one Heartbeat Task, which will then automatically install the Heartbeat Interrupt Handler.

Parameters

Stack before call



Stack after call



Errors

None

C

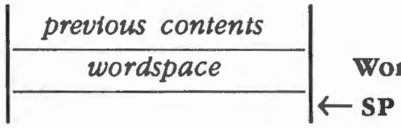
```
extern pascal LongWord GetTick()
```

$2903 GetIRQEnable

Returns with the hardware interrupt enable states for the interrupt sources that can be controlled by the Miscellaneous Tool Set. The interrupt sources are given in Figure 14-1.

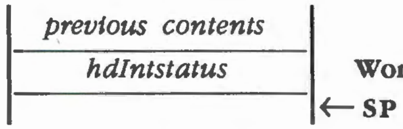
Parameters

Stack before call



Word—Space for result

Stack after call



Word—Status of hardware interrupts (see Figure 14-1)

Errors

None

C

```
extern pascal Word GetIRQEnable()
```

(continued)

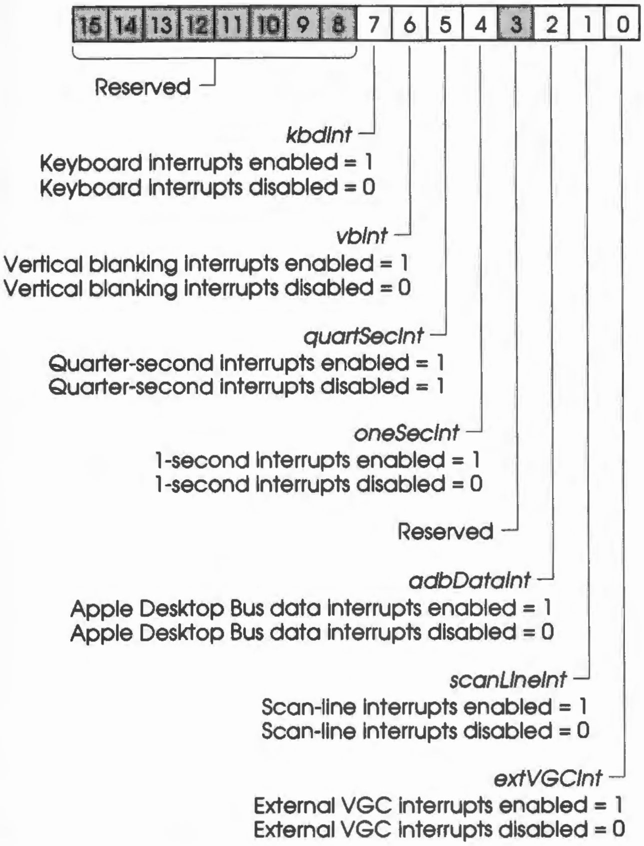

15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Reserved
kbdInt
Keyboard interrupts enabled = 1
Keyboard interrupts disabled = 0
vbInt
Vertical blanking interrupts enabled = 1
Vertical blanking interrupts disabled = 0
quartSecInt
Quarter-second interrupts enabled = 1
Quarter-second interrupts disabled = 1
oneSecInt
1-second interrupts enabled = 1
1-second interrupts disabled = 0
Reserved
adbDataInt
Apple Desktop Bus data interrupts enabled = 1
Apple Desktop Bus data interrupts disabled = 0
scanLineInt
Scan-line interrupts enabled = 1
Scan-line interrupts disabled = 0
extVGCInt
External VGC interrupts enabled = 1
External VGC interrupts disabled = 0

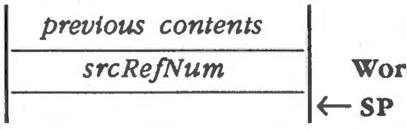
Figure 14-1
Hardware interrupt status

$2303 IntSource

Enables or disables certain interrupt sources, as specified by the *srcRefNum* parameter.

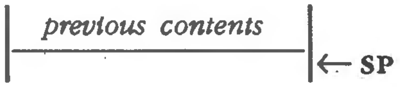
Parameters

Stack before call



Word—Source reference number of interrupt (see Table 14-6)

Stack after call



Errors

None

C

```
extern pascal void IntSource(srcRefNum)

Word     srcRefNum;
```

(continued)

Table 14-6
Interrupt source reference numbers

Number	Name	Action
$0000	kybdEnable	Enable keyboard interrupts
$0001	kybdDisable	Disable keyboard interrupts
$0002	vblEnable	Enable vertical blanking interrupts
$0003	vblDisable	Disable vertical blanking interrupts
$0004	qSecEnable	Enable quarter-second interrupts
$0005	qSecDisable	Disable quarter-second interrupts
$0006	oSecEnable	Enable 1-second interrupts
$0007	oSecDisable	Disable 1-second interrupts
$0008	Reserved	
$0009	Reserved	
$000A	adbEnable	Enable ADB data interrupts
$000B	adbDisable	Disable ADB data interrupts
$000C	scLnEnable	Enable scan-line interrupts
$000D	scLnDisable	Disable scan-line interrupts
$000E	exVCGEnable	Enable external VGC interrupts
$000F	exVCGDisable	Disable external VGC interrupts

Assembly-language example

The following example installs and enables a 1-second interrupt handler. For more information about the 1-second handler, see the section "Writing a 1-Second Interrupt Handler" in this chapter.

```
PEA          $0015         ; Set 1-second vector

PUSHLONG     #ONEHANDLER   ; Pointer to handler

_SETVECTOR

BCS          ERROR         ; If tool set error occurred

PEA          $0006         ; Enable 1-second IRQ

_INTSOURCE

BCS          ERROR         ; If tool set error occurred
```

About keyboard interrupts

When keyboard interrupts are enabled, the firmware installs a task into the Heartbeat Interrupt Task queue and enables VBL interrupts (there is no hardware enable). This causes the Heartbeat Interrupt Handler to be installed into the VBL interrupt vector. This task checks the status of the keyboard register during each VBL interrupt.

If a key is pending, the task dispatches to the Keyboard Interrupt Handler via the keyboard interrupt vector (as installed by the SetVector routine; see the section "SetVector" in this chapter). Since the Heartbeat Interrupt Handler will be installed into the VBL interrupt vector, the application cannot install its own VBL interrupt handler if keyboard interrupts are to be used.

If keyboard interrupts are disabled, the keyboard task is removed from the Heartbeat Interrupt Task queue, but the VBL interrupt is not disabled. If the application does not want the overhead of the background VBL interrupts, the application must also disable the VBL interrupts.

❖ *Note:* If no other tasks have been installed into the Heartbeat Interrupt Task queue, the additional interrupt overhead is minimal (just that for the Interrupt dispatcher and Heartbeat Interrupt Handler, which only increments the tick count before returning).

Writing a 1-second interrupt handler

You can use the vector initialization routines to install a 1-second interrupt handler into the 1-second interrupt vector. You then use the IntSource routine to enable the 1-second interrupt.

The built-in interrupt handler calls the 1-second interrupt handler in 8-bit native mode (m and x registers set to 1). The 1-second interrupt handler must clear the hardware source of the interrupt before executing an RTL to the interrupt manager.

❖ *Note:* Your interrupt handler must return to the built-in interrupt handler with the carry flag cleared if the interrupt source was serviced.

(continued)

An example of a 1-second interrupt handler that increments a memory location is shown here.

```
ONEHANDLER   START
             LONGA    OFF
             LONGI    OFF
             PHB                          ; Save environment
             PHA
             PHK                          ; Set data bank to program
             PLB
             INC      LOCATION
             LDA      #%01000000          ; Clear 1-second IRQ source
             TSB      $C032
             PLA                          ; Restore environment
             PLB
             CLC                          ; Indicate IRQ was serviced
             RTL
             END
```

Writing a quarter-second interrupt handler

You can use the vector initialization routines to install a quarter-second interrupt handler into the quarter-second interrupt vector. You then use the IntSource routine to enable the quarter-second interrupt.

Important
Quarter-second interrupts are reserved for AppleTalk.

The Interrupt Manager calls the quarter-second interrupt handler in 8 bit native mode (m and x registers set to 1). The quarter-second interrupt handler must clear the hardware source of the interrupt before executing an RTL to the interrupt manager.

❖ *Note:* Your interrupt handler must return to the interrupt manager with the carry flag cleared if the interrupt source was serviced.

An example of a quarter-second interrupt handler that increments a memory location is shown here.

```
QTRHANDLER   START
             LONGA    OFF
             LONGI    OFF
             PHB                        ; Save environment
             PHA
             PHK                        ; Set data bank to program
             PLB
             INC      LOCATION
             STA      $C047             ; Clear 1/4-second IRQ source
             PLA                        ; Restore environment
             PLB
             CLC                        ; Indicate IRQ was serviced
             RTL
             END
```

$1C03 ClampMouse

Sets clamp values to new values and then sets the mouse position to the minimum clamp values. The clamp values limit the maximum and minimum X and Y coordinates for the mouse position. For more information about clamping, see the mouse firmware chapter in the *Apple IIGS Firmware Reference.*

Parameters

Stack before call

previous contents	
xMinClamp	**Word**—Minimum clamp value for the X axis
xMaxClamp	**Word**—Maximum clamp value for the X axis
yMinClamp	**Word**—Minimum clamp value for the Y axis
yMaxClamp	**Word**—Maximum clamp value for the Y axis
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void ClampMouse(xMinClamp,xMaxClamp,yMinClamp,yMaxClamp)

Word     xMinClamp;

Word     xMaxClamp;

Word     yMinClamp;

Word     yMaxClamp;
```

You can also use the following alternate form of the call:

```
extern pascal void ClampMouse(clamp)

ClampRec     clamp;
```

$1B03 ClearMouse

Sets the X and Y axis to $0000 if minimum clamps are negative, or to the minimum clamp position if the clamps are positive.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

None

C

```
extern pascal void ClearMouse()
```

$1D03 GetMouseClamp

Returns the current mouse clamp values. The values can be set by a ClampMouse call (see the section "ClampMouse" in this chapter).

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
wordspace	**Word**—Space for result
wordspace	**Word**—Space for result
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
xMinClamp	**Word**—Minimum clamp value for the X axis
xMaxClamp	**Word**—Maximum clamp value for the X axis
yMinClamp	**Word**—Minimum clamp value for the Y axis
yMaxClamp	**Word**—Maximum clamp value for the Y axis
	← **SP**

Errors

None

C

```
extern pascal ClampRec GetMouseClamp()
```

$1A03 HomeMouse

Positions the mouse at the minimum clamp position.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

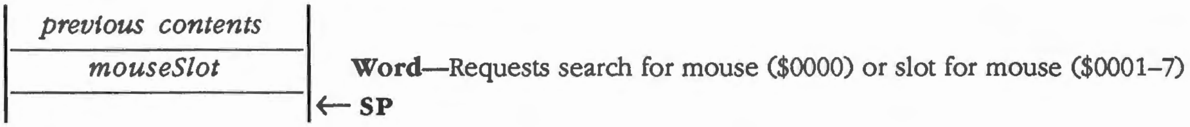
```
extern pascal void HomeMouse()
```

$1803 InitMouse

Initializes mouse clamp values to $000 minimum and $3FF maximum and clears the mouse mode and status.

Parameters

Stack before call

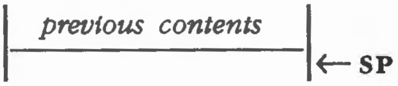


previous contents

mouseSlot **Word**—Requests search for mouse ($0000) or slot for mouse ($0001–7)

← **SP**

Stack after call



previous contents

← **SP**

Errors $0302 `noDevParamErr` No device for input parameter

C

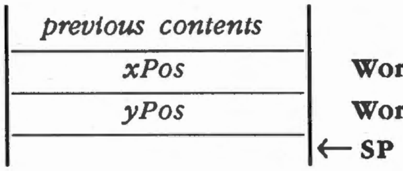
```
extern pascal void InitMouse(mouseSlot)
Word      mouseSlot;
```

$1E03 PosMouse

Positions mouse at specified coordinates.

Parameters

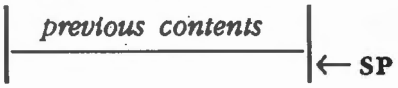
Stack before call



Word—X axis (horizontal) position of the mouse

Word—Y axis (vertical) position of the mouse

Stack after call



Errors

None

C

```
extern pascal void PosMouse(xPox,yPos)

Integer     xPox;

Integer     yPos;


extern pascal void PosMouse(pos)

Point     pos;
```

$1703 ReadMouse

Returns mouse position, status, and mode.

❖ *Note:* The value of *mouseStatus* is the same as the low-order byte of *intStatus* as defined in Figure 14-2 (see the section "ServeMouse" in this chapter); the value of *mouseMode* is the same as that shown in Table 14-7 (see the section "SetMouse" in this chapter).

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
wordspace	**Word**—Space for result
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
xPosition	**Word**—X position of the mouse
yPosition	**Word**—Y position of the mouse
status | *mode*	**Byte**—Mouse status | **Byte**—Mouse mode
	← **SP**

Errors

$0309	`unCnctdDevErr`	Dispatch attempted to unconnected device

C

```
extern pascal MouseRec ReadMouse()
```

$1F03 ServeMouse

Returns the mouse interrupt status.

Parameters

Stack before call

previous contents
wordspace

Word—Space for result

← **SP**

Stack after call

previous contents
intStatus

Word—Mouse interrupt status (see Figure 14-2)

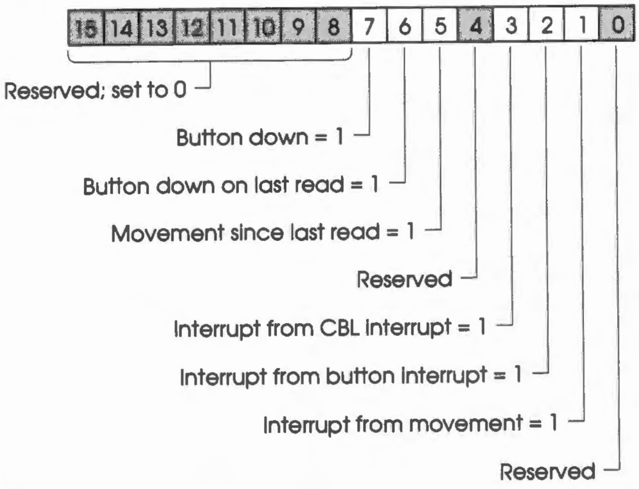
← **SP**

Errors

None

C

```
extern pascal Word ServeMouse()
```



15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Reserved; set to 0
Button down = 1
Button down on last read = 1
Movement since last read = 1
Reserved
Interrupt from CBL interrupt = 1
Interrupt from button interrupt = 1
Interrupt from movement = 1
Reserved

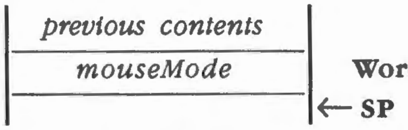
Figure 14-2
Mouse interrupt status word

$1903 SetMouse

Sets the mouse mode.

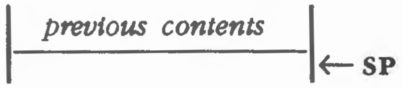
Parameters

Stack before call



Word—Mouse mode, low-order byte only (see Table 14-7)

Stack after call



Errors

None

C

```
extern pascal void SetMouse(mouseMode)

Word     mouseMode;
```

Table 14-7
Mouse mode values

Name	Value	Description
`mouseOff`	$0000	Turn mouse off
`transparent`	$0001	Set transparent mode
`moveIntrpt`	$0003	Set movement interrupt mode
`bttnIntrpt`	$0005	Set button interrupt mode
`bttnOrMove`	$0007	Set button or movement interrupt mode
`mouseOffVI`	$0008	Turn mouse off, VBL IRQ active (VBL IRQ supports emulation-mode interrupts via $3F2)
`transparentVI`	$0009	Set transparent mode, VBL IRQ active
`moveIntrptVI`	$000B	Set movement interrupt mode, VBL IRQ active
`bttnIntrptVI`	$000D	Set button interrupt mode, VBL IRQ active
`bttnOrMoveVI`	$000F	Set button or movement interrupt mode, VBL IRQ active

$2A03 SetAbsClamp

Sets clamp values for an absolute device to new values. The clamp values limit the X and Y position of the absolute device to the specified minimum and maximum values.

Parameters

Stack before call

previous contents	
xMinClamp	**Word**—Minimum clamp value for the X axis
xMaxClamp	**Word**—Maximum clamp value for the X axis
yMinClamp	**Word**—Minimum clamp value for the Y axis
yMaxClamp	**Word**—Maximum clamp value for the Y axis
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetAbsClamp(xMinClamp,xMaxClamp,yMinClamp,yMaxClamp)

Word      xMinClamp;

Word      xMaxClamp;

Word      yMinClamp;

Word      yMaxClamp;
```

You can also use the following alternate form of the call:

```
extern pascal void SetAbsClamp(clamp)

ClampRec     clamp;
```

$2B03 GetAbsClamp

Returns the current values for the absolute device clamps.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
wordspace	**Word**—Space for result
wordspace	**Word**—Space for result
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
xMinClamp	**Word**—Minimum clamp value for the X axis
xMaxClamp	**Word**—Maximum clamp value for the X axis
yMinClamp	**Word**—Minimum clamp value for the Y axis
yMaxClamp	**Word**—Maximum clamp value for the Y axis
	← **SP**

Errors

None

C

```
extern pascal ClampRec GetAbsClamp()
```

$2603 PackBytes

Packs bytes into a special format that uses less storage space.

When the call is finished, the pointer to the area to be packed is moved forward to the next packable byte, and the size of area pointed to by the second input parameter is reduced by the number of bytes traversed.

The packed data is in the form of 1 byte containing a flag in the first 2 bits and a count in the remaining 6 bits, followed by 1 or more data bytes, depending on the flags, as follows:

00xxxxxx : (xxxxxx : 0 → 63) = 1 to 64 bytes follow—all different
01xxxxxx : (xxxxxx : 2, 4, 5, or 6) = 3, 5, 6, or 7 repeats of next byte
10xxxxxx : (xxxxxx : 0 → 63) = 1 to 64 repeats of next 4 bytes
11xxxxxx : (xxxxxx : 0 → 63) = 1 to 64 repeats of next byte taken as 4 bytes
(as in 10xxxxxx case)

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
— *startHandle* —	**Long**—POINTER to POINTER to start of area to be packed
— *sizePtr* —	**Long**—POINTER to Word containing the size of the area
— *bufferPtr* —	**Long**—POINTER to start of the output buffer area
bufferSize	**Word**—Size of the output buffer area
	← **SP**

Stack after call

previous contents	
numPackbytes	**Word**—Number of packed bytes generated
	← **SP**

Errors

None

C

```
extern pascal Word PackBytes(startHandle,sizePtr,bufferPtr,bufferSize)
Handle     startHandle;
Word  *sizePtr;
Pointer     bufferPtr;
Word     bufferSize;
```

Assembly-language example

This example packs a screen image and writes it to a file named f.

```
PB          START
            lda         #$7D00                  ; Size of area to pack
            sta         PicSize
            lda         #$E12000                ; Address of screen image
            sta         PicPtr
            lda         #$^E12000
            sta         PicPtr+2
            lda         #buffer                 ; Pointer to local buffer
            sta         BufPtr
            lda         #^Buffer
            sta         BufPtr+2

loop        PUSHWORD    #0                      ; Space for result
            PUSHLONG    #PicPtr                 ; Pointer to Pointer to data to pack
            PUSHLONG    #PicSize                ; Pointer to word with size of area
            PUSHLONG    BufPtr                  ; Pointer to start of output area
            PUSHWORD    BufSize                 ; Size of output buffer area
            _PACKBYTES
            pla                                 ; Get how much packed this pass
            sta         HowMuch
```

```
          CALL      WRITE(f,BufPtr,HowMuch)  ; Perform I/O to write "HowMuch" bytes from
                                             ; "BufPtr" to file "f"


          lda       PicSize                  ; See if any more to pack

          bne       loop                     ; If there is, go back for more

          rts


PicPtr    ds        4                        ; Set to $e12000 on entry (screen area)

PicSize   ds        2                        ; Size of a picture; set to $7d00 on entry

BufPtr    ds        4                        ; Set to point to "Buffer" on entry

BufSize   dc        i2'$400'                 ; Local buffer for storing packed data

HowMuch   ds        2                        ; Local storage for value from PackBytes

Buffer    ds        $400                     ; Actual buffer

          END
```

$2703 UnPackBytes

Unpacks data from the packed format used by PackBytes (see the section "PackBytes" in this chapter).

When the call is finished, the pointer to the unpacked data is positioned 1 byte past the last unpacked byte, and the size of the area is reduced by the amount unpacked.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *bufferPtr* --	**Long**—POINTER to the buffer containing the packed data
bufferSize	**Word**—Size of the packed data buffer
-- *startHandle* --	**Long**—POINTER to POINTER to area where the data will be unpacked
-- *sizePtr* --	**Long**—POINTER to Word containing size of area for unpacked data
	← **SP**

Stack after call

previous contents	
numUnpackbytes	**Word**—Number of source bytes unpacked
	← **SP**

Errors

None

C

```
extern pascal Word UnPackBytes(bufferPtr,bufferSize,startHandle,sizePtr)

Pointer     bufferPtr;

Word     bufferSize;

Handle     startHandle;

Word  *sizePtr;
```

Assembly-language example

This example unpacks data from a file named f.

```
PB          START
            stz         Mark                    ; Mark is the file mark we position to
            lda         #$7D00                  ; Size of area to unpack into
            sta         PicSize
            lda         #$E12000                ; Address of screen image
            sta         PicPtr
            lda         #$^E12000
            sta         PicPtr+2

loop        CALL        SETFILEMARK(f,Mark)     ; Position file "f" to position "Mark"
            CALL        READ(f,BufPtr,BufSize); Read "BufSize" bytes into "BufPtr"
            PUSHWORD    #0                      ; Space for result
            PUSHLONG    BufPtr                  ; Pointer to start of output area
            PUSHWORD    bufsize                 ; Size of output buffer area
            PUSHLONG    #PicPtr                 ; Pointer to Pointer to data to pack
            PUSHLONG    #PicSize                ; Pointer to word with size of area
            _UNPACKBYTES
            pla                                 ; Get how much unpacked this pass
            clc                                 ; Add to previous mark position
            adc         Mark
            sta         Mark
            lda         picsize                 ; See if more to unpack
            beq         done                    ; If there isn't, we're done

            CALL        EOF(f)                  ; Did we reach end of file? (safety check)
            bne         loop                    ; If not, go back for more
```

(continued)

```
Done        rts

BufPtr      dc          i4'Buffer'              ; Pointer to buffer area

bufsize     dc          i2'$400'                ; Local buffer for storing packed data

PicPtr      ds          4                       ; Set to $e12000 on entry (screen area)

PicSize     ds          2                       ; Size of a picture; set to $7d00 on entry

Mark        ds          2                       ; File mark position

Buffer      ds          $400                    ; Actual buffer

            END
```

$2803 Munger

Manipulates bytes in a string of bytes.

The basic operation is to search a destination string for a target string and, if one is found, replace it with a replacement string. If the destination string is shortened, the end of the string is padded with a pad character. If the string is elongated, the extra characters are truncated. Other special cases are discussed in this section.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *destPtr* --	**Long**—POINTER to pointer to the text to be manipulated
-- *destLenPtr* --	**Long**—POINTER to number of bytes to manipulate
-- *targPtr* --	**Long**—POINTER to be searched for from *destPtr*
targLen	**Word**—Number of bytes for *targPtr*
-- *replPtr* --	**Long**—POINTER to string to replace when *targPtr* found
replLen	**Word**—Number of bytes for *replPtr*
-- *padPtr* --	**Long**—POINTER to character value to be added to short input
	← **SP**

Stack after call

previous contents	
padBytesFound	**Word**—Number of bytes padded or truncated; 0 if target found, negative if not found
	← **SP**

Errors

None

C

```
extern pascal Word Munger(destPtr,destLenPtr,targPtr,targLen,
replPtr,replLen,padPtr)
Handle     destPtr;
Word  *destLenPtr;
Pointer     targPtr;
Word     targLen;
Pointer     replPtr;
Word     replLen;
Pointer     padPtr;
```

Special cases

If *targPtr* is NIL, the substring of length *targLen* is replaced by the *replPtr* string.

If *targLen* is 0, the string pointed to by *replPtr* is inserted at *destPtr.*

If *replPtr* is NIL, *destPtr* is updated past the end of the match of the *targPtr* string.

If *replLen* is 0 (and *replPtr* is not NIL), the *targPtr* string is deleted rather than replaced (since the replacement string is empty).

There is one case in which Munger performs a replacement, even if it doesn't find all of the target string. If the entire *destPtr* string is at the beginning of the *targPtr* string, then the *destPtr* string is totally replaced by the *replPtr* string.

Assembly-language example

This example replaces a word in lowercase format with its uppercase equivalent.

```
*  Changes "robert irwin eagle toranaga marcia houdini berns"
*     into "robert irwin EAGLE toranaga marcia houdini berns"
MG          START
            lda         #Name                 ; Set pointer to name
            sta         DestPtr
            lda         #^Name
            sta         DestPtr+2
            lda         #48                   ; Get length
            sta         DestLen

            PUSHWORD    #0                    ; Space for result
            PUSHLONG    DestPtr               ; Pointer to text string to manipulate
            PUSHLONG    #DestLen              ; Pointer to word with number bytes to change
            PUSHLONG    #eagleLC              ; Points to "eagle" (lowercase)
            PUSHWORD    #5                    ; "eagle" has 5 letters
            PUSHLONG    #eagleUC              ; Pointer to "EAGLE" (uppercase)
            PUSHWORD    #5                    ; "EAGLE" has 5 letters
            PUSHLONG    #PAD                  ; Pad character (don't care for this example)
            _MUNGER
            pla                               ; (This will be 0, as will pad)
            rts
DestPtr     ds          4                     ; On entry will point to name
DestLen     ds          2                     ; On entry will be set to "NLen"
PAD         ds          2                     ; Pad value
eagleLC     dc          c'eagle'
eagleUC     dc          c'EAGLE'
name        dc          c'robert irwin eagle toranaga marcia houdini berns'
```

$1203 SetHeartBeat

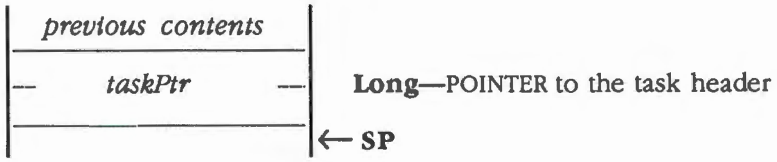
Installs a specifed task into the Heartbeat Interrupt Task queue.

You must set *taskPtr* to point to the task header that precedes the task. The task header consists of a long link pointer, a count word, and a signature word of $5AA5.

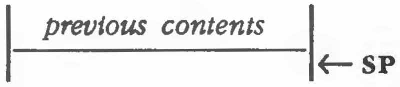
SetHeartBeat maintains *taskPtr,* which is set to NIL ($00000000) if the task is the last task in the queue. When a task is installed, the *taskPtr* of the previous task is set to point at the task header for the task being installed.

Parameters

Stack before call



Stack after call



Errors

$0303	`taskInstlErr`	Specified task already in Heartbeat queue
$0304	`noSigTaskErr`	No signature detected in task header
$0305	`queueDmgdErr`	Damaged Heartbeat queue detected

C

```
extern pascal void SetHeartBeat(taskPtr)

Pointer     taskPtr;
```

Installing a task

The following example increments a location in memory every tenth VBL.

```
Task1Hdr   Data
           dc         h'00000000'          ; Space for task pointer
Task1Cnt   dc         i'10'                ; Count word preset to 10
           dc         h'5AA5'              ; Signature word $A55A
           End
Task1      Start
           Using Task1Hdr
           rep        #$20                 ; 16-bit 'm'
           longa      on
           phk                             ; Data bank = program bank
           plb
           lda        #10                  ; Reset the task count
           sta        Task1Cnt
           sep        #$20                 ; 8-bit 'm'
           longa      off
           lda        >TestLoc             ; Increment an address
           inc        a
           sta        >TestLoc
           rtl
           End
```

The count word indicates the number of VBL interrupts that must occur before the associated task is executed (10 in the preceding example). The application sets the count word before the task is installed, and the count word must then be maintained by either the task or the application. For recurring tasks, the task should reset the count word. For tasks that are run one time only, the application should reset the count word.

❖ *Note:* If you want to use just the tick counter feature of the Heartbeat Interrupt Handler without the task execution feature, install a task with a count of 0.

SetHeartBeat decrements a nonzero count word for each VBL interrupt. If the decrement results in a count word of 0, the task is executed. A count word with a value of 0 is not decremented during a VBL interrupt. This effectively makes the task inactive until a nonzero value is stored for the count word.

The signature word must be set to $5AA5 prior to a task installation and is used by the SetHeartBeat routine and the Heartbeat Interrupt Handler to check the integrity of the Heartbeat Interrupt Task queue.

Tasks are executed in native mode with 8-bit m and x registers. You should terminate the task with an RTL instruction.

The following code installs the task shown above.

```
Install    anop

           pushlong #label

           _SetHeartBeat
```

When you install a task into the Heartbeat Interrupt Task queue, the Heartbeat Interrupt Handler is automatically installed into the VBL interrupt handler vector. This displaces any handler already installed. However, installing the task into the Heartbeat queue doesn't automatically enable VBL interrupts; your application must do so. Also, since tasks are linked with simple pointers, the tasks should reside in locked memory. Tasks that make use of system resources should conform to the protocol detailed in Chapter 19, "Scheduler," in Volume 2.

Installing a ROM-based task

To install a ROM-based task, your application must allocate 12 bytes of RAM for the task header. The task must execute a jump absolute long to the ROM-based task. An example of this is as follows:

```
Task1Hdr   DATA

           DC          H'00000000'          ; Space for link pointer

Task1Cnt   DC          i'10'                ; Count word preset to 10

Task1Sig   DC          H'5AA5'              ; Signature word $A55A

           DC          H'00000000'

           END

Task1Jmp   START

           USING TASK1HDR

           jmp         >RomTask1            ; Jump to ROM-based task

           END
```

An example that shows how a program can construct the task header area in RAM for a ROM-based task is shown next. Note that this program is run in full native mode (16-bit m and x registers).

```
InstallT1 Start

          Using Task1Hdr

          lda       #$0001              ; Initialize task count

          sta       >Task1Cnt

          lda       #$A55A              ; Initialize task signature

          sta       >Task1Sig

          lda       #RomTask1           ; Now install JMP to task

          pha

          xba

          and       #$FF00

          ora       #$005C

          sta       >Task1Jmp

          pla

          and       #$FF00

          ora       #^RomTask1

          xba

          sta       >Task1Jmp+2

          pushlong  Task1Hdr

          _SetHeartBeat
```

$1303 DelHeartBeat

Deletes a specified task from the Heartbeat Interrupt Task queue.

Parameters

Stack before call

```
|  previous contents  |
|---------------------|
|--     taskPtr     --|   Long—POINTER to the task header
|---------------------|
|                     |<— SP
```

Stack after call

```
|  previous contents  |
|---------------------|
|                     |<— SP
```

Errors

$0305	`queueDmgdErr`	Damaged Heartbeat queue detected
$0306	`taskNtFdErr`	Specified task not in queue

C

```
extern pascal void DelHeartBeat(taskPtr)

Pointer     taskPtr;
```

$1403 ClrHeartBeat

Removes all tasks from the Heartbeat Interrupt Task queue by clearing the Heartbeat Task pointer.

Important
A desk accessory may have installed tasks in the Heartbeat Interrupt Task queue. If you make a ClrHeartBeat call, you will remove those tasks. Therefore, under normal circumstances you should not make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void ClrHeartBeat()
```

$2C03 SysBeep

Calls the Apple II monitor entry point BELL1. The bell routine can be patched out using the SetVector routine to patch out the bell vector. Any bell routine installed will be called in native mode (8-bit m and x registers). The routine must return with the carry flag cleared via an RTL instruction.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

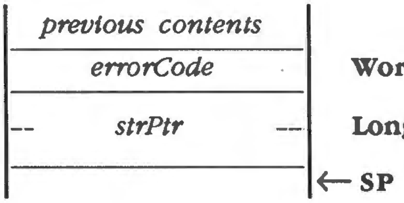
```
extern pascal void SysBeep()
```

$1503 SysFailMgr

Displays system failure message and halts application execution. At system power-up time, a default System Failure Manager is installed into the System Failure Manager vector. The manager displays either a default system failure message followed by an error code, or a user-defined system failure message followed by an error code.

Parameters

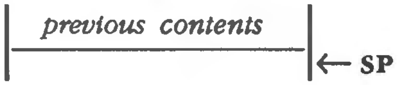
Stack before call



Word—Error code (see Table 14-8)

Long—POINTER to ASCII string to be displayed as system failure message

Stack after call



Errors

None

C

```
extern pascal void SysFailMgr(errorCode,strPtr)

Word      errorCode;

Pointer     strPtr;
```

Default message

The default system failure message displays a sliding Apple below a message as follows:

```
FATAL SYSTEM ERROR -> XXXX
```

If a system failure call is made with a user-defined message, the message is displayed starting at the upper-left corner of the screen. (The text may be moved down by embedding carriage return characters in the text.) The user-defined message may contain 1 to 254 characters.

```
(USER-DEFINED MESSAGE) (XXXX)
```

If *strPtr* is set to 0, the default system failure message and the error code passed as the tool input are displayed. If *strPtr* is set to point to an ASCII string, the ASCII string will be displayed with the error code. The first byte of the ASCII string should contain a count equal to the number of characters to be displayed. The ASCII string should have the most significant byte turned off.

Table 14-8
System failure error codes

Code	Name	Description
$0001	`pdosUnClmdIntErr`	Unclaimed interrupt (ProDOS 16)
$0004	`divByZeroErr`	Division by 0
$000A	`pdosVCBErr`	Volume control block unusable (ProDOS 16)
$000B	`pdosFCBErr`	File control block unusable (ProDOS 16)
$000C	`pdosBlk0Err`	Block zero allocated illegally (ProDOS 16)
$000D	`pdosIntShdwErr`	Interrupt with I/O shadowing off (ProDOS 16)
$0015	`segLoader1Err`	Segment Loader error
$0017	`sPackage0Err`	Can't load a package
$0018	`package1Err`	Can't load a package
$0019	`package2Err`	Can't load a package
$001A	`package3Err`	Can't load a package
$001B	`package4Err`	Can't load a package
$001C	`package5Err`	Can't load a package
$001D	`package6Err`	Can't load a package
$001E	`package7Err`	Can't load a package
$0020	`package8Err`	Can't load a package

(continued)

Table 14-8 (continued)
System failure error codes

Code	Name	Description
$0021	`package9Err`	Can't load a package
$0022	`package10Err`	Can't load a package
$0023	`package11Err`	Can't load a package
$0024	`package12Err`	Can't load a package
$0025	`outOfMemErr`	Out of memory
$0026	`segLoader2Err`	Segment Loader error
$0027	`fMapTrshdErr`	File map destroyed
$0028	`stkOvrFlwErr`	Stack overflow
$0030	`psInstDiskErr`	Please insert disk (File Manager alert)
$0032–53		Memory Manager errors
$0100	`stupVolMntErr`	Can't mount system startup volume

System failure error codes above $0100 are specific to the tool set reporting the error, with the high-order byte containing the tool set number and the low-order byte returning the error code as defined by that tool set. No tool set will report an error with the low-order byte set to $00.

$2003 GetNewID

Creates a new user ID. The user ID marks memory segments as belonging to a specific application or desk accessory. The routine passes the *type* and *auxID* to the User ID Manager, which concatenates the next available *mainID* to the *type* and *auxID* fields. The resulting user ID is returned to the caller. The fields are illustrated in Figure 14-3.

The *type* field must be nonzero. Only 255 ID tags can be assigned for any *type* ID.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
idTag	**Word**—High-order byte = *type* and *auxID* fields; low-order byte = $0000
	← **SP**

Stack after call

previous contents	
userID	**Word**—Complete user ID
	← **SP**

Errors

$0301	`badInputErr`	Bad input parameter
$030B	`idTagNtAvlErr`	No ID tags available

C

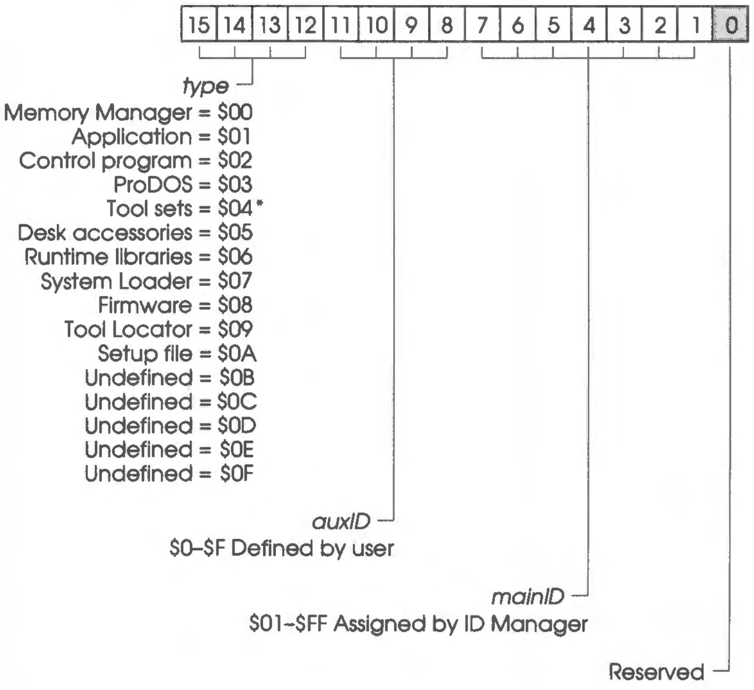
```
extern pascal Word GetNewID(idTag)

Word     idTag;
```

(continued)

User ID fields

User IDs are made up of three fields—the *type, auxID,* and *mainID* fields—encoded in a word parameter. The *type* field is encoded in bits 15–12, the *auxID* in bits 11–8, and the *mainID* field in bits 7–0.

The *auxID* field is defined by the caller. The *mainID* field is generated by the User ID Manager. The user ID is always assigned a nonzero value in the *mainID* field. The *type* field has fixed assignments, as shown in the Figure 14-3.



* Apple controls assignment of IDs in this class. At the time of publication, the only assignments were $41xx = Miscellaneous Tool Set and $42xx = Scrap Manager.

Figure 14-3
User ID fields

$2103 DeleteID

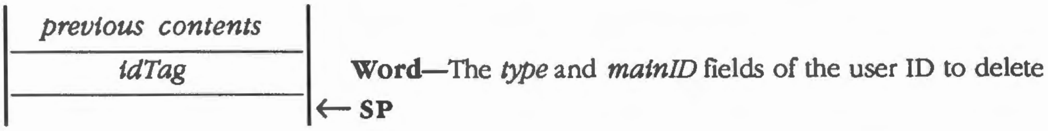
Deletes all references to a specified user ID. Any user IDs with the same *mainID* and *type* are deleted from the current user ID list. The fields are illustrated in Figure 14-3 (see the section "GetNewID" in this chapter). This routine does not report an error if the tag is not found.

Important

The *auxID* field is ignored during this call.

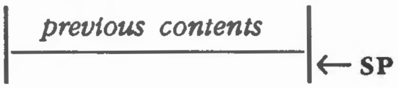
Parameters

Stack before call



Word—The *type* and *mainID* fields of the user ID to delete

Stack after call



Errors

None

C

```
extern pascal void DeleteID(idTag)

Word     idTag;
```

$2203 StatusID

Indicates whether a specified user ID is active.

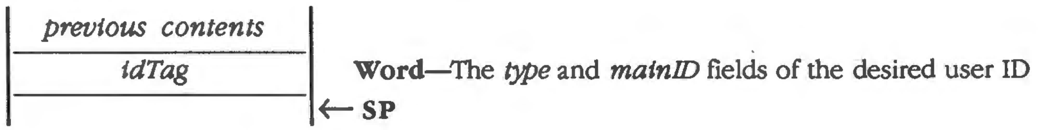
The call must specify the *type* and *mainID* of the desired user ID. The fields are illustrated in Figure 14-3 (see the section "GetNewID" in this chapter). If the user ID is active, no error will be returned.

Important

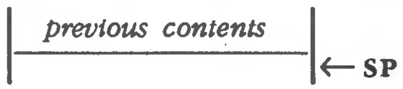
The *auxID* is ignored during this call.

Parameters

Stack before call


previous contents
idTag
Word—The *type* and *mainID* fields of the desired user ID
← **SP**

Stack after call



Errors

$030B	`idTagNtAvlErr`	No ID tag available

C

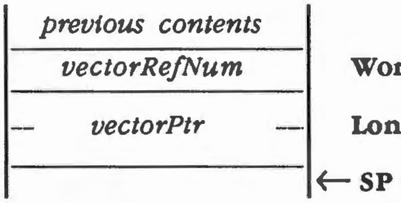
```
extern pascal void StatusID(idTag)

Word     idTag;
```

$1003 SetVector

Sets the vector address for the interrupt manager or handler specified by the *vectorRefNum.* The vector reference numbers are given in Table 14-9. You can retrieve the current vector address for a specified interrupt manager or handler with a GetVector call (see the section "GetVector" in this chapter for more information).

Parameters

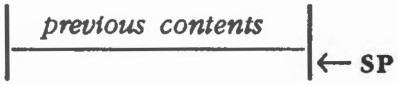
Stack before call



Word—Vector to be set (see Table 14-9)

Long—POINTER to the manager or handler

Stack after call



Errors

None

C

```
extern pascal void SetVector(vectorRefNum,vectorAddrPtr)

Word      vectorRefNum;

Pointer      vectorAddrPtr;
```

(continued)

Table 14-9
Vector reference numbers

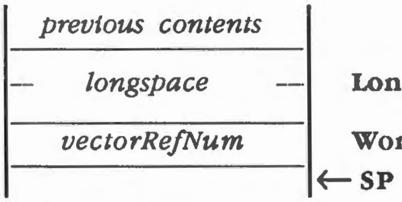
Number	Vector	Number	Vector
$0000	Tool Locator #1	$001B	Bell vector (for Sound tools)
$0001	Tool Locator #2	$001C	Break vector (for debuggers)
$0002	User's tool locator #1	$001D	Trace vector
$0003	User's tool locator #2	$001E	Step vector
$0004	Interrupt Manager	$001F	Reserved vector
$0005	COP Manager	$0020	Reserved vector
$0006	Abort Manager	$0021	Reserved vector
$0007	System Death Manager	$0022	Reserved vector
$0008	AppleTalk interrupt handler	$0023	Reserved vector
$0009	Serial Communications Controller interrupt handler	$0024	Reserved vector
$000A	Scan-line interrupt handler	$0025	Reserved vector
$000B	Sound interrupt handler	$0026	Reserved vector
$000C	Vertical blanking interrupt handler	$0027	Reserved vector
$000D	Mouse interrupt handler	$0028	Control Y vector
$000E	Quarter-second interrupt handler	$0029	Reserved vector
$000F	Keyboard interrupt handler	$002A	ProDOS 16 MLI vector
$0010	ADB response byte interrupt handler	$002B	Operating system vector
$0011	ADB SRQ interrupt handler	$002C	Message pointer vector
$0012	Desk Accessory Manager		
$0013	Flush buffer handler		
$0014	Keyboard micro interrupt handler		
$0015	1-second interrupt handler		
$0016	External VGC interrupt handler		
$0017	Other unspecified interrupt handler		
$0018	Cursor update handler		
$0019	Increment busy flag (for Scheduler)		
$001A	Decrement busy flag (for Scheduler)		

$1103 GetVector

Returns the vector address for the interrupt manager or handler for a specified vector reference number. Vector reference numbers are given in Table 14-9 in the section "SetVector" in this chapter; you can use the SetVector routine to set the vector address for an interrupt manager or handler.

Parameters

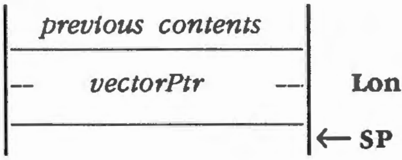
Stack before call



Long—Space for result

Word—Vector reference number of manager or handler

Stack after call



Long—POINTER to the specified manager or handler

Errors

None

C

```
extern pascal Pointer GetVector(vectorRefNum)

Word     vectorRefNum;
```

Miscellaneous Tool Set summary

This section briefly summarizes the constants, data structures, and tool set errors contained in the Miscellaneous Tool Set.

Important

These definitions are provided in the appropriate interface file.

Table 14-10
Miscellaneous Tool Set constants

Name	Value	Description
Battery RAM parameter reference numbers		
`p1PrntModem`	$0000	Port 1 Printer/modem
`p1LineLnth`	$0001	Port 1 Line length
`p1DelLine`	$0002	Port 1 Delete line feed after carriage return
`p1AddLine`	$0003	Port 1 Add line feed after carriage return
`p1Echo`	$0004	Port 1 Echo
`p1Buffer`	$0005	Port 1 Buffer
`p1Baud`	$0006	Port 1 Baud
`p1DtStpBits`	$0007	Port 1 Data/stop bits
`p1Parity`	$0008	Port 1 Parity
`p1DCDHndShk`	$0009	Port 1 DCD handshake
`p1DSRHndShk`	$000A	Port 1 DSR handshake
`p1XnfHndShk`	$000B	Port 1 XON/XOFF handshake
`p2PrntModem`	$000C	Port 2 Printer/modem
`p2LineLnth`	$000D	Port 2 Line length
`p2DelLine`	$000E	Port 2 Delete line feed after carriage return
`p2AddLine`	$000F	Port 2 Add line feed after carriage return
`p2Echo`	$0010	Port 2 Echo
`p2Buffer`	$0011	Port 2 Buffer
`p2Baud`	$0012	Port 2 Baud
`p2DtStpBits`	$0013	Port 2 Data/stop Bits
`p2Parity`	$0014	Port 2 Parity
`p2DCDHndShk`	$0015	Port 2 DCD handshake
`p2DSRHndShk`	$0016	Port 2 DSR handshake
`p2XnfHndShk`	$0017	Port 2 XON/XOFF handshake
`dspColMono`	$0018	Display color/monochrome
`dsp40or80`	$0019	Display 40/80 column
`dspTxtColor`	$001A	Display text color
`dspBckColor`	$001B	Display background color
`dspBrdColor`	$001C	Display border color
`hrtz50or60`	$001D	50/60 Hz

Table 14-10 (continued)
Miscellaneous Tool Set constants

Name	Value	Description
Battery RAM parameter reference numbers		
userVolume	$001E	User volume
bellVolume	$001F	Bell volume
sysSpeed	$0020	System speed
slt1intExt	$0021	Slot 1 Internal/external
slt2intExt	$0022	Slot 2 Internal/external
slt3intExt	$0023	Slot 3 Internal/external
slt4intExt	$0024	Slot 4 Internal/external
slt5intExt	$0025	Slot 5 Internal/external
slt6intExt	$0026	Slot 6 Internal/external
slt7intExt	$0027	Slot 7 Internal/external
startupSlt	$0028	Startup slot
txtDspLang	$0029	Text display language
kybdLang	$002A	Keyboard language
kyBdBuffer	$002B	Keyboard buffering
kyBdRepSpd	$002C	Keyboard repeat speed
kyBdRepDel	$002D	Keyboard repeat delay
dblClkTime	$002E	Double-click time
flashRate	$002F	Flash rate
shftCpsLCas	$0030	Shift caps/lowercase
fstSpDelKey	$0031	Fast space/delete keys
dualSpeed	$0032	Dual speed
hiMouseRes	$0033	High mouse resolution
dateFormat	$0034	Month/day/year format
clockFormat	$0035	24 hour/AM-PM format
rdMinRam	$0036	Minimum RAM for RAM disk
rdMaxRam	$0037	Maximum RAM for RAM disk
langCount	$0038	Number of languages
lang1	$0039	First language
lang2	$003A	Second language
lang3	$003B	Third language
lang4	$003C	Fourth language
lang5	$003D	Fifth language
lang6	$003E	Sixth language
lang7	$003F	Seventh language
lang8	$0040	Eighth language
layoutCount	$0041	Number of keyboard layouts
layout1	$0042	First keyboard layout
layout2	$0043	Second keyboard layout
layout3	$0044	Third keyboard layout

(continued)

Table 14-10 (continued)
Miscellaneous Tool Set constants

Name	Value	Description
Battery RAM parameter reference numbers		
layout4	$0045	Fourth keyboard layout
layout5	$0046	Fifth keyboard layout
layout6	$0047	Sixth keyboard layout
layout7	$0048	Seventh keyboard layout
layout8	$0049	Eighth keyboard layout
layout9	$004A	Ninth keyboard layout
layout10	$004B	Tenth keyboard layout
layout11	$004C	Eleventh keyboard layout
layout12	$004D	Twelfth keyboard layout
layout13	$004E	Thirteenth keyboard layout
layout14	$004F	Fourteenth keyboard layout
layout15	$0050	Fifteenth keyboard layout
layout16	$0051	Sixteenth keyboard layout
aTalkNodeNo	$0080	AppleTalk node number
GetAddr parameter reference numbers		
irqIntFlag	$0000	Reserved for internal use
irqDataReg	$0001	Reserved for internal use
irqSerial1	$0002	Reserved for internal use
irqSerial2	$0003	Reserved for internal use
irqAplTlkHi	$0004	Reserved for internal use
tickCnt	$0005	Tick counter
irqVolume	$0006	IRQ volume
irqActive	$0007	IRQ bctive
irqSndData	$0008	IRQ sound data
brkVar	$0009	Variables after a BRK
evMgrData	$000A	Event Manager data
mouseSlot	$000B	Mouse location/flag
mouseClamps	$000C	Mouse clamps
absClamps	$000D	Absolute clamps
sccIntFlag	$000E	Serial Communications Controller (SCC) interrupt flag
Hardware interrupt status (returned by GetIRQEnable)		
extVGCInt	$01	External VGC interrupts enabled
scanLineInt	$02	Scan line interrupts enabled
adbDataInt	$04	Apple Desktop Bus interrupts enabled
oneSecInt	$10	1-second interrupts enabled
quartSecInt	$20	Quarter-second interrupts enabled
vbInt	$40	Vertical blanking interrupts enabled
kbdInt	$80	Keyboard interrupts enabled

Table 14-10 (continued)
Miscellaneous Tool Set constants

Name	Value	Description
Interrupt reference numbers		
`kybdEnable`	$0000	Enable keyboard Interrupts
`kybdDisable`	$0001	Disable keyboard Interrupts
`vblEnable`	$0002	Enable vertical blanking interrupts
`vblDisable`	$0003	Disable vertical blanking interrupts
`qSecEnable`	$0004	Enable quarter-second interrupts
`qSecDisable`	$0005	Disable quarter-second interrupts
`oSecEnable`	$0006	Enable 1-second interrupts
`oSecDisable`	$0007	Disable 1-second interrupts
`adbEnable`	$000A	Enable FDB data interrupts
`adbDisable`	$000B	Disable FDB data interrupts
`scLnEnable`	$000C	Enable scan-line interrupts
`scLnDisable`	$000D	Disable scan-line interrupts
`exVCGEnable`	$000E	Enable external VGC interrupts
`exVCGDisable`	$000F	Disable external VGC interrupts
Mouse mode reference numbers		
`mouseOff`	$0000	Turn mouse off
`transparent`	$0001	Set transparent mode
`moveIntrpt`	$0003	Set movement interrupt mode
`bttnIntrpt`	$0005	Set button interrupt mode
`bttnOrMove`	$0007	Set button or movement interrupt mode
`mouseOffVI`	$0008	Turn mouse off, VBL IRQ active (VBL IRQ supports emulation-mode interrupts via $3F2)
`transparentVI`	$0009	Set transparent mode, VBL IRQ active
`moveIntrptVI`	$000B	Set movement interrupt mode, VBL IRQ active
`bttnIntrptVI`	$000D	Set button interrupt mode, VBL IRQ active
`bttnOrMoveVI`	$000F	Set button or movement interrupt mode, VBL IRQ active
Vector reference numbers		
`toolLoc1`	$0000	Tool Locator #1
`toolLoc2`	$0001	Tool Locator #2
`usrTLoc1`	$0002	User's tool locator #1
`usrTLoc2`	$0003	User's tool locator #2
`intrptMgr`	$0004	Interrupt Manager
`copMgr`	$0005	COP Manager
`abortMgr`	$0006	Abort Manager
`sysFailMgr`	$0007	System Failure Manager
`aTalkIntHnd`	$0008	AppleTalk interrupt handler
`sccIntHnd`	$0009	Serial Communications Controller interrupt handler

(continued)

Table 14-10 (continued)
Miscellaneous Tool Set constants

Name	Value	Description
Vector reference numbers		
`scLnIntHnd`	$000A	Scan-line interrupt handler
`sndIntHnd`	$000B	Sound interrupt handler
`vblIntHnd`	$000C	Vertical blanking interrupt handler
`mouseIntHnd`	$000D	Mouse interrupt handler
`qSecIntHnd`	$000E	Quarter-second interrupt handler
`kybdIntHnd`	$000F	Keyboard interrupt handler
`adbRBIHnd`	$0010	ADB response byte interrupt handler
`adbSRQHnd`	$0011	ADB SRQ interrupt handler
`deskAccHnd`	$0012	Desk Accessory Manager
`flshBufHnd`	$0013	Flush buffer handler
`kybdMicHnd`	$0014	Keyboard micro interrupt handler
`oneSecHnd`	$0015	1-second interrupt handler
`extVCGHnd`	$0016	External VGC interrupt handler
`otherIntHnd`	$0017	Other unspecified interrupt handler
`crsrUpdtHnd`	$0018	Cursor update handler
`incBsyFlag`	$0019	Increment busy flag (for Scheduler)
`decBsyFlag`	$001A	Decrement busy flag (for Scheduler)
`bellVector`	$001B	Bell vector (for Sound Tool Set)
`breakVector`	$001C	Break vector (for debuggers)
`traceVector`	$001D	Trace vector
`stepVector`	$001E	Step vector
`ctlYVector`	$0028	Control Y vector
`proDOSVctr`	$002A	ProDOS 16 MLI vector
`osVector`	$002B	Operating system vector
`msgPtrVctr`	$002C	Message pointer vector

Table 14-11
Miscellaneous Tool Set data structures

Name	Offset	Type	Definition
TimeRec			
second	$00	Byte	Second (0–59)
minute	$01	Byte	Minute (0–59)
hour	$02	Byte	Hour (0–23)
year	$03	Byte	Current year minus 1900
day	$04	Byte	Date (0–30)
month	$05	Byte	Month (0–11, with 0 = January and so on)
extra	$06	Byte	Fills up word
weekDay	$07	Byte	Day of week (1–7, with 1 = Sunday and so on)
MouseRec			
mouseMode	$00	Byte	Mouse mode
mouseStatus	$01	Byte	Mouse status
yPos	$02	Word	Y position of the mouse
xPos	$04	Word	X position of the mouse
ClampRec			
yMaxClamp	$00	Word	Maximum clamp value for the Y axis
yMinClamp	$02	Word	Minimum clamp value for the Y axis
xMaxClamp	$04	Word	Maximum clamp value for the X axis
xMinClamp	$06	Word	Minimum clamp value for the X axis
FWRec			
yRegExit	$00	Word	Y register at exit
xRegExit	$02	Word	X register at exit
aRegExit	$04	Word	A register at exit
status	$06	Word	Processor status at exit

Note: The actual assembly-language equates have a lowercase *o* (the letter) in front of all of the names given in this table.

Table 14-12
Miscellaneous Tool Set error codes

Code	Name	Description
System failure codes		
$0001	pdosUnClmdIntErr	Unclaimed interrupt (ProDOS 16)
$0004	divByZeroErr	Division by 0
$000A	pdosVCBErr	Volume control block unusable (ProDOS 16)
$000B	pdosFCBErr	File control block unusable (ProDOS 16)
$000C	pdosBlk0Err	Block zero allocated illegally (ProDOS 16)
$000D	pdosIntShdwErr	Interrupt with I/O shadowing off (ProDOS 16)
$0015	segLoader1Err	Segment Loader error
$0017	sPackage0Err	Can't load a package
$0018	package1Err	Can't load a package
$0019	package2Err	Can't load a package
$001A	package3Err	Can't load a package
$001B	package4Err	Can't load a package
$001C	package5Err	Can't load a package
$001D	package6Err	Can't load a package
$001E	package7Err	Can't load a package
$0020	package8Err	Can't load a package
$0021	package9Err	Can't load a package
$0022	package10Err	Can't load a package
$0023	package11Err	Can't load a package
$0024	package12Err	Can't load a package
$0025	outOfMemErr	Out of memory
$0026	segLoader2Err	Segment Loader error
$0027	fMapTrshdErr	File map destroyed
$0028	stkOvrFlwErr	Stack overflow
$0030	psInstDiskErr	Please insert disk (File Manager alert)
$0032–53		Memory Manager errors
$0100	stupVolMntErr	Can't mount system startup volume
Miscellaneous Tool Set error codes		
$0301	badInputErr	Bad input parameter
$0302	noDevParamErr	No device for input parameter
$0303	taskInstlErr	Specified task already in Heartbeat queue
$0304	noSigTaskErr	No signature detected in task header
$0305	queueDmgdErr	Damaged Heartbeat queue detected
$0306	taskNtFdErr	Specified task not in queue
$0307	firmTaskErr	Unsuccessful firmware task
$0308	hbQueueBadErr	Damaged Heartbeat queue detected
$0309	unCnctdDevErr	Dispatch attempted to unconnected device
$030B	idTagNtAvlErr	No ID tag available

Chapter 15

Print Manager

The **Print Manager** allows you to use standard QuickDraw II routines to print text or graphics on a printer. The Print Manager calls a printer driver to perform the specific printing tasks, so that your application doesn't need to know what kind of printer is connected to the computer.

You should already be familiar with QuickDraw II.

An application that supports printing must have three items in its File menu: Choose Printer, Page Setup, and Print. The following actions occur when the user selects one of these items:

- **Choose Printer:** When the user selects the Choose Printer item, a dialog is displayed that allows the user to select a destination device from the printer drivers on the system disk. The Choose Printer dialog also lets the user pick the port or slot to which the device is connected from the port drivers on the system disk. If AppleTalk is installed, the network is scanned for the names of all printers of the specified printer type.

❖ *Macintosh programmers:* On the Apple IIGS, the Choose Printer function is part of the Print Manager, rather than being implemented as part of the Chooser desk accessory.

- **Page Setup:** When the user selects the Page Setup item, the style dialog is displayed. The style dialog allows the user to specify formatting information, such as the page size and printing orientation. This information is not changed frequently and is usually saved with the document.
- **Print:** When the user selects the Print item, the job dialog is displayed. The job dialog lets the user select print quality, page range, number of copies, and printer-specific features, such as color printing.

Your application defines the image to be printed by using either the GrafPort that the Print Manager automatically gives you when you open a document for printing, or by supplying its own GrafPort.

❖ *Note:* The Print Manager does not automatically save the current GrafPort when it initializes the new GrafPort. If the application will need that GrafPort's information for later use, it must save the information itself.

Your application then prints text and graphics by drawing into the GrafPort with QuickDraw II, just as if it was drawing on the screen. The Print Manager installs its own versions of QuickDraw II's low-level drawing routines in this GrafPort, causing your higher-level QuickDraw II calls to drive the printer instead of drawing on the screen.

Important

Don't customize the QuickDraw II routines in the GrafPort being used for printing unless you're sure of what you're doing.

A preview of the Print Manager routines

To introduce you to the capabilities of the Integer Math Tool Set, all Print Manager routines are grouped by function and briefly described in Table 15-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the Print Manager routines (discussed in alphabetical order).

Table 15-1
Print Manager routines and their functions

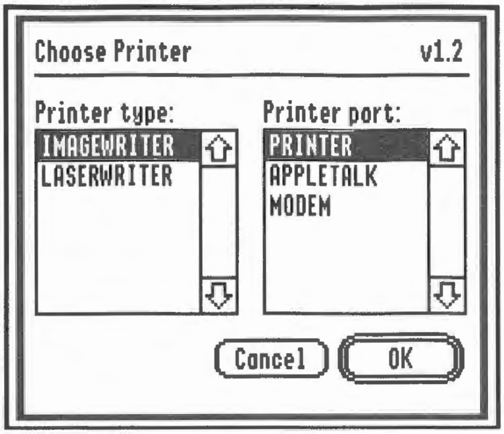
Routine	Description
Housekeeping routines	
PMBootInit	Initializes the Print Manager; called only by the Tool Locator—must not be called by an application
PMStartUp	Starts up the Print Manager for use by an application
PMShutDown	Shuts down the Print Manager
PMVersion	Returns the version number of the Print Manager
PMReset	Resets the Print Manager; called only when the system is reset—must not be called by an application
PMStatus	Indicates whether the Print Manager is active
Print record and dialog routines	
PrDefault	Fills the fields of a specified print record with default values for the appropriate printer
PrValidate	Checks the contents of the specified print record for compatibility with the current version number of the Print Manager and the currently installed printer
PrStlDialog	Conducts a style dialog with the user to determine the page dimensions and other information needed for page setup
PrJobDialog	Conducts a job dialog with the user to determine the print quality, range of pages to print, and so on
PrChoosePrinter	Conducts a Choose Printer dialog with the user to determine the printer and port driver to use
Printing routines	
PrOpenDoc	Initializes a GrafPort for use in printing a document, makes it the current port, and returns a pointer to the port
PrCloseDoc	Closes the GrafPort being used for printing
PrOpenPage	Begins a new page
PrClosePage	Ends the printing of the current page
PrPicFile	Prints a spooled document
PrPixelMap	Prints all or part of a specified pixel map
Error handling routines	
PrError	Returns the last printer error code left during the printing loop by Print Manager routines
PrSetError	Stores a specified value into the global variable where the Print Manager keeps its printer error code
Printer driver and port driver routines	
PrDriverVer	Returns the version number of the currently installed printer driver
PrPortVer	Returns the version number of the currently installed port driver

Print dialog boxes

The dialog boxes seen by the user when he or she chooses an item from the File menu are the Choose Printer dialog, the style dialog, and the job dialog. These dialogs allow the user to specify information needed by the Print Manager to process the print job. This information is stored in the appropriate print record fields.

Choose Printer dialog box

The **Choose Printer dialog box** allows the user to choose the printer to be used for printing and the port that connects that printer to the system, as illustrated in Figure 15-1.



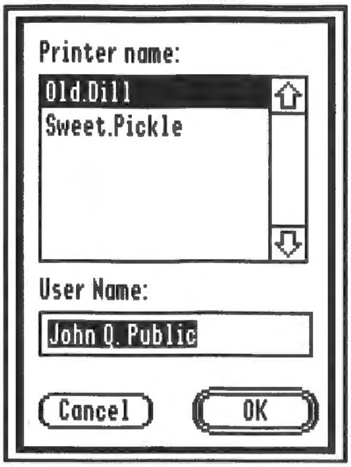
Choose Printer v1.2

Printer type: IMAGEWRITER, LASERWRITER

Printer port: PRINTER, APPLETALK, MODEM

Figure 15-1
Choose Printer dialog box

If the printer chosen is connected to the system via AppleTalk, then an additional dialog box appears, showing all available printers of the specified type on the network, as illustrated in Figure 15-2.



Printer name:

Old.Dill

Sweet.Pickle

User Name: John Q. Public

Figure 15-2
Printer names dialog box

Style dialog box

The **style dialog box** presents the user with a choice of paper size and orientation.

Both the ImageWriter® and the LaserWriter® printers interpret the paper options as shown in Table 15-2, although each printer does not offer all of the options.

Table 15-2
Printer paper sizes

Option	Dimensions
US Letter	8 1/2 by 11 inches
US Legal	8 1/2 by 14 inches
A4 Letter	210 by 297 millimeters
B5 Letter	176 by 250 millimeters
International fanfold	210 millimeters by 12 inches

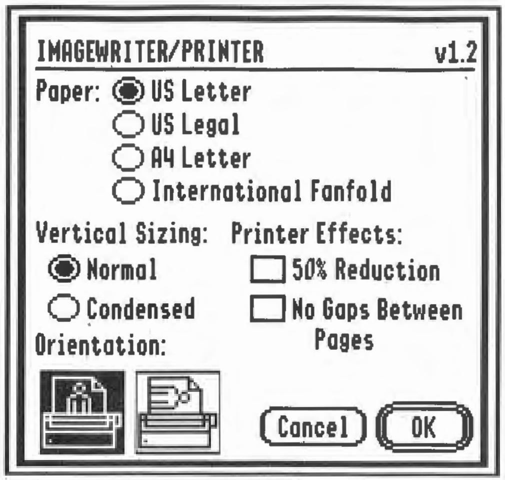
Both printers also use the same definitions of vertical sizes, although the ImageWriter cannot print intermediate text. The vertical-size definitions are

- Normal, which prints for 640 mode at 80 ppi (pixels per inch), and for 320 mode at 40 ppi horizontally and 36 ppi vertically
- Intermediate, which prints at 54 ppi vertically
- Condensed, which prints at 72 ppi vertically

❖ *Macintosh programmers:* The condensed vertical size resembles the screen size of Macintosh text.

Both printers also allow the user to choose between printing in **portrait mode** (in which text prints from left to right) and **landscape mode** (in which text prints from top to bottom).

Other options differ on the two printers, so the style dialog boxes also differ. The dialog box that appears if the user chooses an ImageWriter is shown in Figure 15-3.



IMAGEWRITER/PRINTER v1.2

Paper: ● US Letter
○ US Legal
○ A4 Letter
○ International Fanfold

Vertical Sizing: ● Normal ○ Condensed

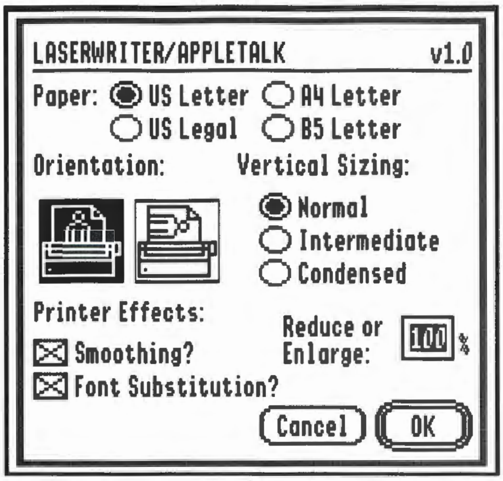
Printer Effects: □ 50% Reduction □ No Gaps Between Pages

Orientation:

Figure 15-3
Style dialog box for ImageWriter

The printer effects choices for the ImageWriter allow the user to print at half size as well as with no gaps between pages (that is, the printer uses the full vertical 11 inches).

The dialog box that appears if the user chooses a LaserWriter is shown in Figure 15-4.



LASERWRITER/APPLETALK v1.0

Paper: ● US Letter ○ A4 Letter
○ US Legal ○ B5 Letter

Orientation:

Vertical Sizing:
● Normal
○ Intermediate
○ Condensed

Printer Effects:
☒ Smoothing?
☒ Font Substitution?

Reduce or Enlarge: 100 %

Figure 15-4
Style dialog box for LaserWriter

There are two printing options available on the LaserWriter that don't exist on the ImageWriter.

Smoothing asks the system to smooth out any bit-mapped fonts with jagged edges. **Font substitution** tells the system to make the following substitutions if the specified font is not in the LaserWriter: Helvetica for Geneva, Courier for Monaco, Times for New York. Any other font is downloaded as a bit map to the LaserWriter.

❖ *Note:* At the time of publication, the Print Manager did not support downloading bit-mapped fonts to the LaserWriter; that is, Courier will be substituted for all fonts other than those listed above. However, on the Laserwriter Plus, the Zapf Chancery font will be substituted for the Venice font.

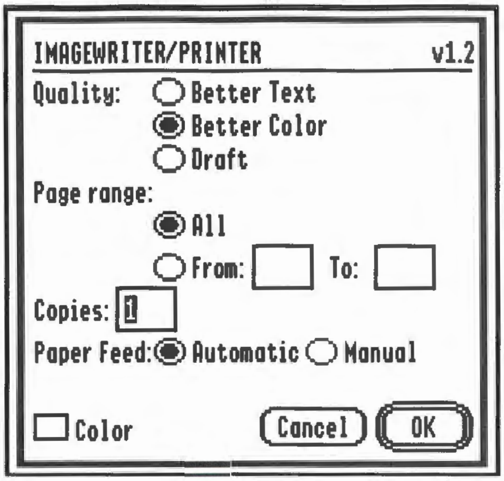
The printed representation of the document can range from 25 to 400 percent of the original size, with 100 percent representing normal size.

Job dialog box

The **job dialog box** allows the user to communicate the page range, the number of copies, and the paper source to the Print Manager. In addition, the ImageWriter job dialog offers print-quality choices and the option to print in color.

In most cases, your application doesn't need to know what choices the user makes for this dialog; that is, the application will simply use QuickDraw II calls to draw into the GrafPort being used for printing, and the Print Manager will handle the user's selections.

The ImageWriter job dialog box is shown in Figure 15-5.



IMAGEWRITER/PRINTER v1.2

Quality: ○ Better Text

● Better Color

○ Draft

Page range:

● All

○ From: □ To: □

Copies: 1

Paper Feed: ● Automatic ○ Manual

□ Color

Figure 15-5
Job dialog box for ImageWriter

The Better Text option doubles the resolution, but halves the color choices available; therefore, if your application isn't printing color graphics, this choice by the user will produce the highest quality.

The Better Color option prints the document at the same resolution as the screen, with the same number of screen colors available. The option is most appropriate when printing color graphics.

The Draft option is useful only when the user wants to quickly print text without any formatting information.

If the user clicks the Color box, the ImageWriter driver will print using a color ribbon if the ribbon is available on the specified ImageWriter. If the ribbon is not available, or if the user does not click the Color box, the ImageWriter will print in black and white.

The LaserWriter job dialog box is shown in Figure 15-6.

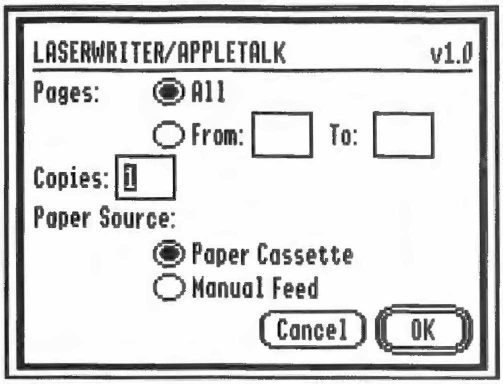


Figure 15-6
Job dialog box for LaserWriter

Print records

To format and print a document, the Print Manager checks the information contained in a data structure called a **print record.** The Print Manager fills in the entire print record for you by using information specified by the user during the job and style dialogs.

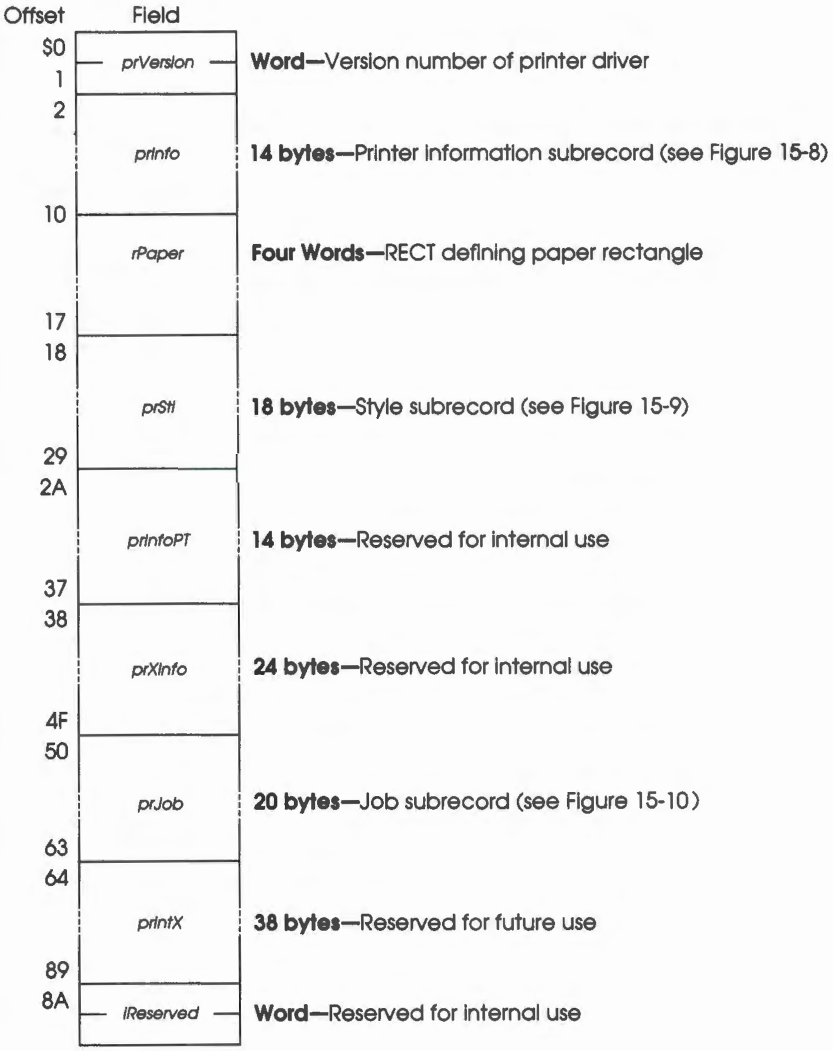
❖ *Note:* Whenever your application saves a document, it should write an appropriate print record into the document. This sets up the printing parameters so that the document retains and uses those parameters the next time the document is printed.

Information contained in the print record includes the following:

- Dimensions of the printable area of the page
- Whether the application must calculate the margins, the size of the physical sheet of paper, and the printer's vertical and horizontal resolution
- Whether draft or spool printing is being used

Important
Your application doesn't need to change much of the data in the print record; usually, the only fields you'll need to set directly are those containing optional information in the job subrecord. You should use use standard dialog routines for controlling the print record information. If you want to directly change values in the print record, be sure you know what you are doing.

Print records are referred to by handles. The structure of a print record is shown in Figure 15-7.



Offset	Field	
$0 1	*prVersion*	**Word**—Version number of printer driver
2	*prInfo*	**14 bytes**—Printer information subrecord (see Figure 15-8)
10 17	*rPaper*	**Four Words**—RECT defining paper rectangle
18 29	*prStl*	**18 bytes**—Style subrecord (see Figure 15-9)
2A 37	*prInfoPT*	**14 bytes**—Reserved for internal use
38 4F	*prXInfo*	**24 bytes**—Reserved for internal use
50 63	*prJob*	**20 bytes**—Job subrecord (see Figure 15-10)
64 89	*printX*	**38 bytes**—Reserved for future use
8A	*iReserved*	**Word**—Reserved for internal use

Figure 15-7
Print record

The *prVersion* field identifies the version of the printer driver that initialized this print record. If you try to use a print record that is invalid for the current version of the Print Manager or for the currently installed printer, the Print Manager will correct the record by filling it with default values.

The other fields of the print record are discussed in the following sections.

Printer information subrecord

The **printer information subrecord** (the print record field *prInfo*) gives you the information needed for page composition. The structure of the printer information subrecord is shown in Figure 15-8.

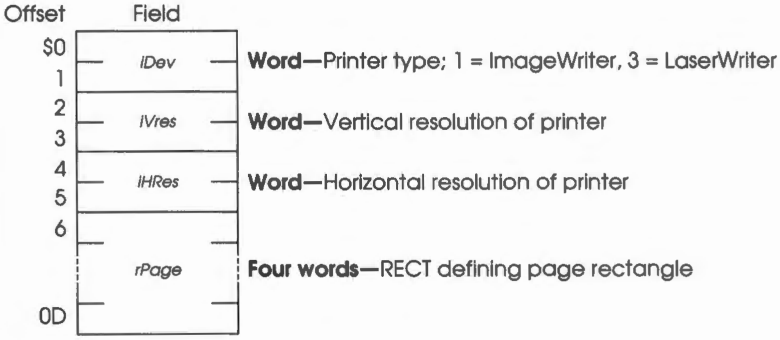

Offset
Field
$0
1
2
3
4
5
6
0D
iDev
iVres
iHRes
rPage
Word—Printer type; 1 = ImageWriter, 3 = LaserWriter
Word—Vertical resolution of printer
Word—Horizontal resolution of printer
Four words—RECT defining page rectangle

Figure 15-8
Printer information subrecord

The *iDev* field identifies which type of printer the user selected in the Choose Printer dialog.

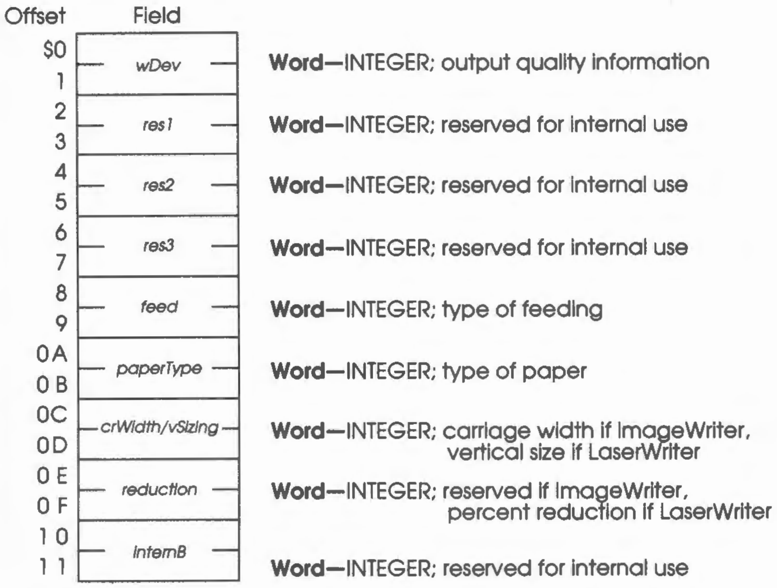
The *rPage* field is the page rectangle, representing the boundaries of the printable page. The GrafPort's boundsRect, portRect, and clipRgn are set to this rectangle. Its top-left corner always has coordinates (0,0); the coordinates of the bottom-right corner give the maximum page height and width attainable on the given printer, in pixels. Typically these are slightly less than the physical dimensions of the paper, because of the printer's mechanical limitations. The value of *rPage* is set as a result of the style dialog.

The *rPage* rectangle is inside the paper rectangle, specified by the print record *rPaper* field. The *rPaper* field gives the physical paper size, defined in the same coordinate system as *rPage*. Thus, the top-left coordinates of the paper rectangle are typically negative, and the bottom-right coordinates are greater than those of the page rectangle.

The *iVRes* and *iHRes* fields contain the printer's vertical and horizontal resolution in pixels per inch. Thus, if you divide the width of *rPage* by *iHRes*, you get the width of the page rectangle in inches.

Style subrecord

The **style subrecord** (the print record field *prStl*) contains information obtained from the user via the style dialog as well as the job dialog. Some of the fields in this subrecord have different meanings for different printers. The structure of the style subrecord is shown in Figure 15-9.



Offset	Field	
$0 1	*wDev*	**Word**—INTEGER; output quality information
2 3	*res1*	**Word**—INTEGER; reserved for internal use
4 5	*res2*	**Word**—INTEGER; reserved for internal use
6 7	*res3*	**Word**—INTEGER; reserved for internal use
8 9	*feed*	**Word**—INTEGER; type of feeding
0A 0B	*paperType*	**Word**—INTEGER; type of paper
0C 0D	*crWidth/vSizing*	**Word**—INTEGER; carriage width if ImageWriter, vertical size if LaserWriter
0E 0F	*reduction*	**Word**—INTEGER; reserved if ImageWriter, percent reduction if LaserWriter
10 11	*internB*	**Word**—INTEGER; reserved for internal use

Figure 15-9
Printer style subrecord

The possible values for these fields are shown in the following sections.

ImageWriter style subrecord values

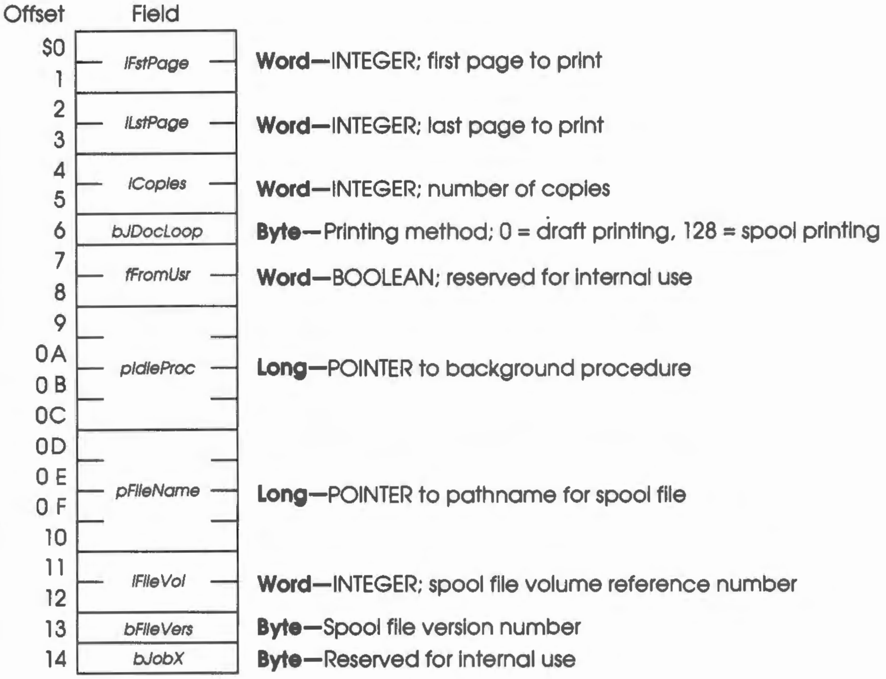
wDev	Bit 6	0 = no gap, 1 = gap
	Bit 5	0 = black and white, 1 = color
	Bit 4	reserved
	Bit 3	0 = 50% reduction, 1 = full size
	Bit 2	0 = condensed, 1 = normal
	Bit 1	0 = landscape, 1 = portrait
	Bit 0	0 = normal quality, 1 = best quality
feed	feedManual = 0, feedAuto = 1	
paperType	UsLetter = 0	
	UsLegal = 1	
	A4Letter = 2	
	IntlFanFold = 3	
crWidth	960 for all paper types	

LaserWriter style subrecord values

wDev	Bit 3	0 = substitute fonts, 1 = don't substitute fonts
	Bit 2	0 = smoothing; 1 = no smoothing
	Bit 1	0 = portrait, 1 = landscape
	Bit 0	Reserved
feed	feedManual = 0, feedAuto = 1	
paperType	UsLetter = 0	
	UsLegal = 1	
	A4Letter = 2	
	B5Letter = 3	
vSizing	Normal = 0	
	Intermediate = 1	
	Condensed = 2	

Job subrecord

The **job subrecord** (the print record field *prJob*) contains information about a particular printing job. Its contents are set as a result of the job dialog. The job subrecord is defined as shown in Figure 15-10.



Offset	Field	
$0 1	*iFstPage*	**Word**—INTEGER; first page to print
2 3	*iLstPage*	**Word**—INTEGER; last page to print
4 5	*iCopies*	**Word**—INTEGER; number of copies
6	*bJDocLoop*	**Byte**—Printing method; 0 = draft printing, 128 = spool printing
7 8	*fFromUsr*	**Word**—BOOLEAN; reserved for internal use
9 0A 0B 0C	*pIdleProc*	**Long**—POINTER to background procedure
0D 0E 0F 10	*pFileName*	**Long**—POINTER to pathname for spool file
11 12	*iFileVol*	**Word**—INTEGER; spool file volume reference number
13	*bFileVers*	**Byte**—Spool file version number
14	*bJobX*	**Byte**—Reserved for internal use

Figure 15-10
Job subrecord

The *iFstPage* and *iLstPage* fields designate the first and last pages to be printed. These page numbers are relative to the first page counted by the Print Manager. The Print Manager cannot use any page numbering placed within a document by an application.

The *iCopies* field is the number of copies to print. The Print Manager automatically handles multiple copies for spool printing or for printing on the LaserWriter. Your application needs this number only for draft printing on the ImageWriter.

The *bJDocLoop* field designates the printing method, either draft or spool, that the Print Manager will use. Draft printing means that the document will be printed immediately. Spool printing means that printing may be deferred: The Print Manager writes a representation of the document's printed image to a disk file (or possibly to memory); this information is then converted into a bit image and printed. Your application can check this field to determine which type of printing the user selected.

The *pIdleProc* field is a pointer to the background procedure (explained later) for this printing operation. In a newly initialized print record this field is set to NIL, designating the default background procedure, which just polls the keyboard and cancels further printing if the user types Apple-period. You can install your own background procedure by storing a pointer to that procedure directly into the *pIdleProc* field.

For spool printing, your application may optionally provide a spool file pathname, volume reference number, and version number, as follows:

- *pFileName* is the full pathname of the spool file as a ProDOS string. This field is initialized to NIL and is generally not changed by the application. NIL denotes the default filename.
- *iFileVol* is the unit number of spool file volume, initialized to 0.
- *bFileVers* is the version number of the spool file, initialized to 0.

Printing modes and resolutions

The Print Manager supports the various display modes available on the Apple IIGS. The IIGS is capable of both color and gray scale in either 320 or 640 modes. When the user requests color printing via the job dialog, the Print Manager prints the document in color on a printer that is capable of color printing, such as the ImageWriter II; otherwise it will print it in black and white.

This next section presents an overview of the algorithm for printing the Super Hi-Res color screen to the ImageWriter printer. It describes both the transformation from the color screen to a color printer and the method used to print the color screen in black and white (gray scale).

The IIGS screen has two resolution modes:

- 320 horizontal by 200 vertical pixels, 16 colors per line
- 640 horizontal by 200 vertical pixels, 4 colors per line

For more detail regarding the exact implementation of the color, see Chapter 16, "QuickDraw II," in Volume 2.

The ImageWriter II can print horizontally at 160 dots per inch and vertically at 144. With a color ribbon, which consists of bands of cyan, magenta, yellow, and black, the ImageWriter II supports eight possible colors per dot. These color dot combinations, except for black, are composed of all the different mixtures of the colors cyan, magenta, and yellow. Black is represented by the black ribbon, and no other color dots include black and another color.

The color screen is initially converted into color pixels before the screen-to-printer transformation occurs. Each pixel consists of a total of 12 bits: 4 bits each for red, blue, and green (RGB). Thus, each color has 16 possible levels, and each pixel has 4096 possible colors (or levels). The first part of the algorithm must convert these RGB pixels into color pixels that the printer understands. These pixels consist of three colors: cyan, magenta, and yellow (CMY).

This conversion routine performs a straightforward matrix multiplication to achieve the translation from RGB to CMY space. All level information is retained, which means that the CMY result is also 4 bits per color. The matrix values used for this translation are described at the end of this section.

After the algorithm has calculated the CMY levels, it enters the second translation phase: converting from 16 levels of each color to a number of color levels supported by the printer. The number of color levels and the resolution (color pixels per inch) can be traded off by the higher-level software routines. For example, at 160 color pixels per inch (ppi), each pixel can represent one of 8 possible colors, but if the resolution is cut in half (80 ppi), then each pixel has 27 possible colors. Table 15-3 illustrates the possible tradeoffs allowed by the low-level driver.

Table 15-3
Resolution, colors, and gray scales

Average resolution (ppi)	Colors per pixel	Levels per ribbon color	Gray-scale levels
160	8	2	2
80	27	3	3
40	125	5	5
20	729	9	9

The gray-scale column in Table 15-3 helps clarify how the colors-per-pixel value is determined. For example, when there are only 40 pixels per inch, there can be 4 possible dots for each pixel (as determined by the 160-dots-per-inch capability of the printer). These 4 possible dots represent five different levels of brightness: 0 dots, 1 dot, 2 dots, 3 dots, and 4 dots can be printed at a time. Regardless of which 3 dots are printed, the eye will perceive the same amount of brightness and color from that pixel. Since there are 3 possible colors (CMY) that can be printed at each dot, this means that there are 5^3, or 125, possible colors.

Table 15-3 specifies average resolution because the dots for each pixel don't always fall on the same printing line. At 40 pixels per inch, each pixel is actually comprised of two horizontal by two vertical dots. At 20 pixels per inch, the pixel consists of four horizontal by two vertical dots. The concepts are illustrated in Figure 15-11.

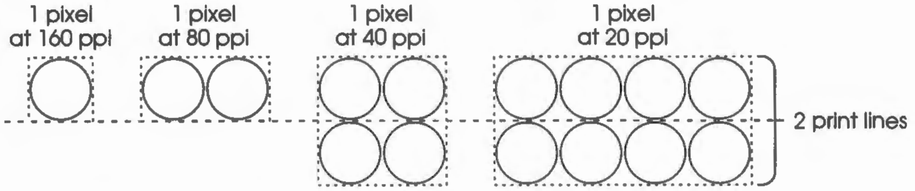


Figure 15-11
Pixels and print lines

The second phase of the color transformation consists of a simple linear translation of the old color-level value (16 possible levels per color) to a new color-level value (either 2, 3, 5, or 9 possible levels, depending on the mode).

At this point, the color data information is defined for printing on the color printer. However, for a black-and-white printer, there must be one more transformation that changes the pixels from color space to black-and-white space. This transformation is performed by another matrix multiplication that converts the three color values (CMY) into a single black-and-white value.

The resultant black-and-white value is actually a gray-scale value that is sent to the printer in the same manner as the color values. The only difference is that each pixel is a specified number of black dots instead of color dots. The level information is retained in the translation from color to black and white.

The horizontal and vertical resolutions, the pixel size, and the page rectangle size of the various modes, and the print quality for the ImageWriter II are summarized in Figure 15-12.

❖ *Note:* The terms ***portrait*** and ***landscape*** refers to the page orientation, representing vertical and horizontal orientations, respectively. The printing area for a page with a gap is 8 x 10 1/2 inches, and that for a page without a gap is 8 x 11 inches.

Dots per pixel	Horizontal resolution	Vertical resolution	Page rectangle
Portrait, 320 mode			
□□□□ □□□□ 4 x 2 = 8 dots Normal quality	40 ppi/160 dpi	36 ppi/72 dpi	with gap: 0,0,378,320 w/o gap: 0,0,396,320
□□ 2 x 1 = 2 dots Best quality	80 ppi/160 dpi	72 ppi/72 dpi	Same as above
Portrait, 640 mode			
□□ □□ 2 x 2 = 4 dots Normal quality	80 ppi/160 dpi	36 ppi/72 dpi	with gap: 0,0,378,640 w/o gap: 0,0,396,640
□ 1 dot Best quality	160 ppi/160 dpi	72 ppi/72 dpi	Same as above
Landscape, 320 mode			
□□□□ □□□□ 4 x 2 = 8 dots Normal quality	36 ppi/72 dpi	40 ppi/160 dpi	with gap: 0,0,320,378 w/o gap: 0,0,320,396
□□ 2 x 1 = 2 dots Best quality	72 ppi/72 dpi	80 ppi/160 dpi	Same as above
Landscape, 640 mode			
□□□□ 4 x 1 = 4 dots Normal quality	72 ppi/72 dpi	40 ppi/160 dpi	with gap: 0,0,320,756 w/o gap: 0,0,320,792
□□ 2 x 1 = 2 dots Best quality	144 ppi/144 dpi	80 ppi/160 dpi	Same as above

Figure 15-12
Resolution, pixel size, page size, and print quality

Using the Print Manager

This section discusses how the Print Manager routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

The Print Manager depends upon the presence of the tool sets shown in Table 15-4 and requires that at least the indicated version of the tool set be present.

Table 15-4
Print Manager—other tool sets required

Tool set number		Tool set name	Minimum version needed
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	2.0
$03	#03	Miscellaneous Tool Set	2.0
$04	#04	QuickDraw II	2.0
$05	#05	Desk Manager	1.0
$0E	#14	Window Manager	1.3
$0F	#15	Menu Manager	1.3
$10	#16	Control Manager	1.3
$12	#18	QuickDraw II Auxiliary Routines	1.0
$14	#20	LineEdit Tool Set	1.0
$15	#21	Dialog Manager	1.1
$1B	#27	Font Manager	1.0
$1C	#28	List Manager	1.0

Your application must make a PMStartUp call once before making any other Print Manager calls. Conversely, if the application makes a PMStartUP call, the application must make a PMShutDown call before it quits or before it unloads the Print Manager.

Before you can print a document, you need a valid print record. You can use an existing print record (for instance, one saved with a document), or you can initialize one by calling PrDefault. If you use an existing print record, be sure to call PrValidate to make sure it's valid for the current version number of the Print Manager and for the currently installed printer.

To create a new print record, you must first create a handle to it with the Memory Manager NewHandle routine. A print record is 140 bytes long.

Printer and print record information is obtained via the style and job dialogs:

- Call PrChoosePrinter when the user chooses the Select Printer command from the File menu. No print record is required for this call.
- Call PrStlDialog to get the page dimensions when the user chooses the Page Setup command. From the *rPage* field of the printer information subrecord, you can then determine where page breaks will be in the document. You can show rulers and margins correctly by using the information in the print record *iVRes, iHRes,* and *rPaper* fields.
- Call PrJobDialog to get the specific information about that printing job (such as the page range and number of copies) when the user chooses the Print command.

When the user chooses the Print command, your application normally should immediately start its printing loop in order to conform to the *Human Interface Guidelines: The Apple Desktop Interface*.

The first step of the printing loop is using the PrOpenDoc routine to obtain a pointer to the GrafPort to be used for printing. This must be done only once for each print job.

The next step calls for beginning the inner loop of printing the pages one by one, as detailed in the next section.

Printing loop

To print a document, you call the following routines:

1. PrOpenDoc, which returns a pointer to the GrafPort to be used for printing
2. PrOpenPage, which starts each new page (reinitializing the GrafPort)
3. QuickDraw routines, for drawing the page into the GrafPort whose pointer was returned by PrOpenDoc
4. PrClosePage, which terminates the page
5. PrCloseDoc, at the end of the entire document, to close the GrafPort being used for printing

Each page is either printed immediately (draft printing) or written to the disk or to memory (spool printing). You should test to see whether spooling was done and, if so, print the spooled document. First, your application should use the Memory Manager routine MaxBlock to ensure that a 10K block of memory is available. If it isn't, you should swap out enough memory to allow for that 10K block and then call PrPicFile.

You should check for errors after each Print Manager call. If an error occurs, you should abort printing by setting the error to `prAbort` using the PrSetError routine. Be sure that your application exits the printing loop normally so that all allocated memory is deallocated accordingly; that is, be sure that PrOpenDoc is matched by PrCloseDoc and that every PrOpenPage is matched by a PrClosePage.

❖ *Note:* The maximum number of pages in a spool file is 16,382. If you need to print more than 16,382 pages at one time, just repeat the printing loop (without calling PrValidate, PrStlDialog, or PrJobDialog).

Printing a specified range of pages

Your application can try to print an entire document even when the user wants to print only a selected subrange of pages. The Print Manager processes each page but actually prints only the pages from *iFstPage* to *iLstPage*.

However, if the application knows the page boundaries in the document, it is much faster to loop through only the specified pages. The application can do this by saving the values of *iFstPage* and *iLstPage* after the PrJobDialog call, recalculating the page range using 1 as the starting page and storing these values into the appropriate fields in the print record. For example, to print pages 20 to 25 of a document, you would set *iFstPage* to 1 and *iLstPage* to 6 and then begin the printing loop at the document's page 20.

Remember that *iFstPage* and *iLstPage* are relative to the first page counted by the Print Manager. The Print Manager counts one page each time PrOpenPage is called; the count begins at 1.

Using QuickDraw II for printing

When drawing into the QuickDraw II GrafPort being used for printing, you should note the following:

- With each new page, you get a completely reinitialized GrafPort, so you'll need to reset font information and other GrafPort characteristics as you want them.
- Don't use clipping to select text to be printed. There are a number of subtle differences between how text appears on the screen and how it appears on the printer; you can't count on knowing the exact dimensions of the rectangle occupied by the text.
- Don't use fixed-width fonts to align columns. Since spacing is adjusted by the printer, you should explicitly move the pen to where you want it.
- Don't make calls that don't do anything on the printer. For example, erase operations are time consuming and normally aren't needed on the printer.

For printing to the LaserWriter, you'll need to observe the following limitations:

- Regions aren't supported; try to simulate them with polygons.
- Clipping regions should be limited to rectangles.
- Invert routines, such as the QuickDraw II routines InvertRect and InvertRgn, aren't supported.
- Copy is the only transfer mode supported for all objects except text and bit images. For text, Bic is also supported. For bit images, the only transfer mode not supported is XOR.
- Don't change the GrafPort's local coordinate system (with SetOrigin) within the printing loop (between PrOpenPage and PrClosePage).

Sequence of events

The following pseudocode will help you understand the entire sequence of calls necessary to print a document. Your application should take the following steps:

```
 1. Call PrChoosePrinter, if user selects Choose Printer menu item.
 2. Call PrStlDialog, if user selects Page Setup menu item.
 3. Call PrJobDialog, when user selects Print menu item.
 4. Call either
        PrDefault, if no existing print record
                or
        PrValidate, if there is an existing print record.
 5. Call PrOpenDoc.
        If tool call error, then PrSetError to prAbort.
 6. Enter printing loop:
        Call PrOpenPage.
                If tool call error, then PrSetError to prAbort.
        Call appropriate QuickDraw II routines, including those that reset font
        information.
        Call PrClosePage.
                If tool call error, then PrSetError to prAbort.
 7. Repeat loop for each page printed.
 8. Call PrCloseDoc.
 9. Call PrError; if error not zero, then skip PrPicFile call.
10. Call Memory Manager routine MaxBlock to check for a 10K block, if none, swap
    out parts of application to make room.
11. Call PrPicFile.
```

Methods of printing

There are two basic methods of printing documents: draft and spool.

- **Draft printing:** Your QuickDraw II calls are converted directly into command codes the printer understands, which are then immediately used to drive the printer. The LaserWriter always uses draft printing, since the QuickDraw II calls are translated immediately into PostScript commands. The ImageWriter and other unintelligent dot matrix printers are written to in draft mode for text only. High-quality pixel-map images are produced only during spool printing.
- **Spool printing:** The Print Manager processes your printing requests in two steps. First it writes (spools) a representation of your document's printed image to a disk file or to memory. Second, this information is converted into a bit image and printed. This method is used to print graphics on the ImageWriter.

Printer and port drivers

Both the ImageWriter and LaserWriter printers are fully supported. Other printers should work so long as drivers are written for them; these drivers may be developed by Apple or third-party developers.

Printer drivers

Apple provides the printer drivers for the ImageWriter and LaserWriter.

The user can install new printer drivers into the system by saving a printer driver file into the DRIVERS subdirectory within the SYSTEM subdirectory. The printer driver file must be of file type $BB and have an aux type of $0001.

At the time of publication, no more information about printer driver formats was available.

Printer peripheral cards and printer ports

Port drivers are used to support the various methods of connecting a printer to the Apple IIGS. A port driver can be written to work with a built-in port or with a peripheral card in a slot. Currently, the following three types of drivers are defined, all of which support the internal ports:

- Printer.Port
- Modem.Port
- Appletalk.Port

The user can install new port drivers into the system by saving a port driver file into the DRIVERS subdirectory within the SYSTEM subdirectory. The port driver file must be of file type $BB and have an aux type of $0002.

At the time of publication, no more information about port driver formats was available.

Background processing

As already mentioned, the job subrecord includes a pointer, *pIdleProc,* to an optional **background procedure** run whenever the Print Manager has directed output to the printer and is waiting for the printer to finish. The background procedure has no parameters and returns no result; the Print Manager simply runs it at every opportunity.

If you don't designate a background procedure, the Print Manager uses a default procedure for canceling printing. The default procedure polls the keyboard and sets a Print Manager error code if the user types Apple-period. If you use this option, you should display a dialog box during printing to inform the user that the Apple-period option is available.

If you do designate a background procedure, you must set *pIdleProc* after presenting the dialogs, validating the print record, and initializing the GrafPort. The routines that perform these operations reset *pIdleProc* to NIL.

Important

If you write your own background procedure, you must be careful to avoid a number of subtle concurrency problems that can arise. For instance, if the background procedure uses QuickDraw II, it must be sure to restore the GrafPort being used for printing as the current port before returning. It's particularly important not to attempt any printing from within the background procedure: The Print Manager is not reentrant. If you use a background procedure that runs your application concurrently with printing, it should disable all menu items having to do with printing, such as Page Setup and Print.

$0113 PMBootInit

Initializes the Print Manager; called only by the Tool Locator.

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 Call must not be made by an application.

$0213 PMStartUp

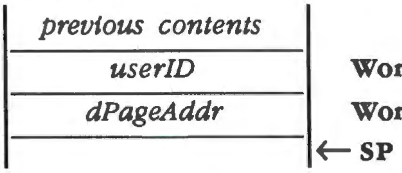
Starts up the Print Manager for use by an application.

Important

Your application must make this call before it makes any other Print Manager calls.

Parameters

Stack before call



Word—ID number of the application

Word—Bank $0 starting address for 2 pages of direct-page space

Stack after call

previous contents
← SP

Errors

$1301	`missingDriver`	Specified driver not in DRIVERS subdirectory of SYSTEM subdirectory
System Loader errors		Returned unchanged
Tool Locator errors		Returned unchanged
Memory Manager errors		Returned unchanged

C

```
extern pascal void PMStartUp(userID,dPageAddr)

Word      userID;

Word      dPageAddr;
```

$0313 PMShutDown

Shuts down the Print Manager.

Important

If your application has started up the Print Manager, the application must make this call before it quits.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

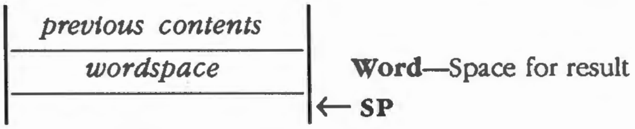
```
extern pascal void PMShutDown()
```

$0413 PMVersion

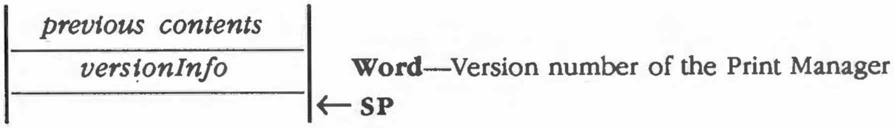
Returns the version number of the Print Manager.

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal Word PMVersion()
```

$0513 PMReset

Resets the Print Manager; called 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 Call must not be made by an application.

$0613 PMStatus

Indicates whether the Print Manager is active.

Parameters

Stack before call

previous contents
wordspace — **Word**—Space for result
← **SP**

Stack after call

previous contents
activeFlag — **Word**—BOOLEAN; TRUE if Print Manager active, FALSE if inactive
← **SP**

Errors None

C

```
extern pascal Boolean PMStatus()
```

$1613 PrChoosePrinter

Conducts a chooser dialog with the user to determine what printer and port driver to use. The *driverChangedFlag*, if TRUE, signals that the user has selected a different driver than the one currently being used.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
driverChangedFlag	**Word**—BOOLEAN; TRUE if driver changed, FALSE if no change
	← **SP**

Errors

None

C

```
extern pascal Boolean PrChoosePrinter()
```

$0F13 PrCloseDoc

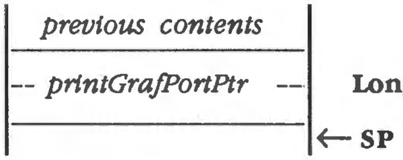
Closes the GrafPort being used for printing. For draft printing, PrCloseDoc ends the printing job.

For spool printing, PrCloseDoc ends the spooling process; the spooled document must now be printed. Before printing it, your application should call the PrError routine to find out whether spooling succeeded. If spooling did succeed, the application must make sure that a 10K block of memory is available to allow for the printing buffer and then call the PrPicFile routine.

❖ *Note:* Your application can use the Memory Manager routine MaxBlock to ensure that a 10K block is available. See the section “MaxBlock” in Chapter 12, “Memory Manager.”

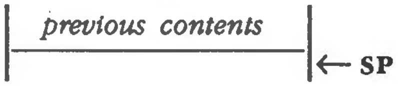
Parameters

Stack before call



Long—POINTER to GrafPort to be used for printing

Stack after call



Errors

$1302	portNotOn	Specified port not selected in Control Panel

C

```
extern pascal void PrCloseDoc(printGrafPortPtr)

GrafPortPtr     printGrafPortPtr;
```

$1113 PrClosePage

Ends the printing of the current page. The routine signals the Print Manager that your application is finished with this page, so that the Print Manager can perform whatever close operations are required for the current printer and printing method.

Parameters

Stack before call

previous contents
-- *printGrafPortPtr* -- **Long**—POINTER to GrafPort to be used for printing
← **SP**

Stack after call

previous contents
← **SP**

Errors

$1302 `portNotOn` Specified port not selected in Control Panel

C

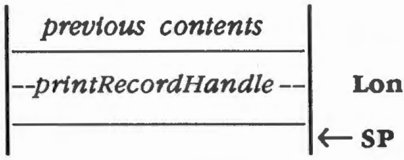
```
extern pascal void PrClosePage(printGrafPortPtr)

GrafPortPtr     printGrafPortPtr;
```

$0913 PrDefault

Fills the fields of the specified print record with default values for the current printer. The record with the *printRecordHandle* may be a new or an existing print record.

Parameters

Stack before call



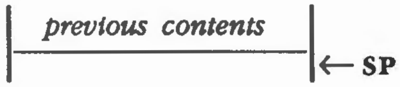
previous contents

--printRecordHandle--

← **SP**

Long—HANDLE to print record

Stack after call



previous contents

← **SP**

Errors

$1303	`noPrintRecord`	No print record was specified
Memory Manager errors		Returned unchanged

C

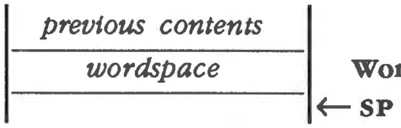
```
extern pascal void PrDefault(printRecordHandle)

PrRecHndl     printRecordHandle;
```

$2313 PrDriverVer

Returns the version number of the currently installed printer driver.

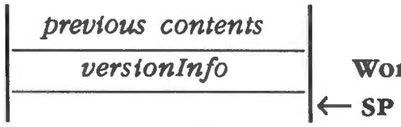
Parameters

Stack before call



Word—Space for result

Stack after call



Word—Version number of the printer driver

Errors

None

C

```
extern pascal Word PrDriverVer()
```

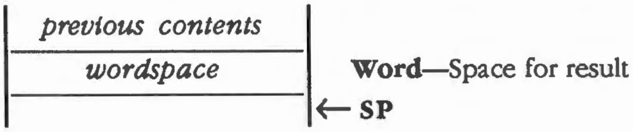
$1413 PrError

Returns the last printer error code left during the printing loop by Print Manager routines. In addition to the tool set error codes (see Table 15-7 in the section "Print Manager Summary" at the end of this chapter), other possible error codes are as follows:

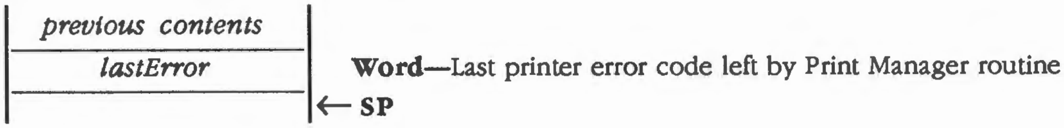
No error	$00
`prAbort`	$80

Parameters

Stack before call



Stack after call



previous contents
lastError
Word—Last printer error code left by Print Manager routine
← **SP**

Errors

None

C

```
extern pascal Word PrError()
```

$0C13 PrJobDialog

Conducts a job dialog with the user to determine the print quality, range of pages to print, and so on. The initial settings displayed in the dialog box are taken from the printer driver, where they were retained from the previous job (with the exception of the page range, which is set to All, and the number of copies, which is set to 1).

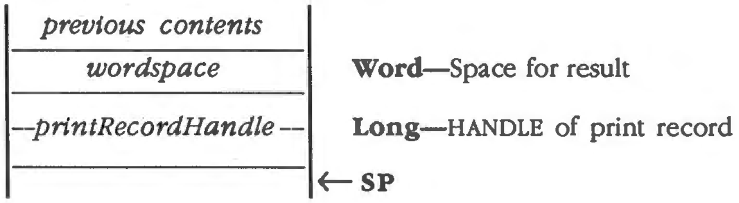
If the user confirms the dialog, the print record is updated, the PrValidate routine is automatically called, and the routine returns TRUE. Otherwise, the print record is left unchanged, and the routine returns FALSE.

Important
Never call PrJobDialog between the PrOpenPage and PrClosePage calls.

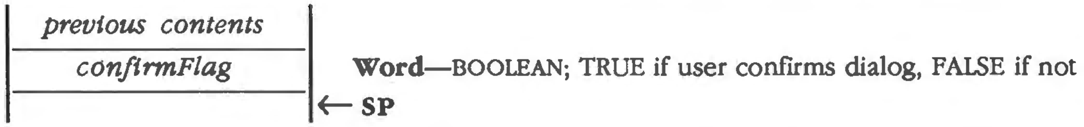
❖ *Note:* Since the job dialog is associated with the Print command, you should proceed with the requested printing operation if *confirmFlag* is TRUE.

Parameters

Stack before call



Stack after call



Errors

Memory Manager errors Returned unchanged

C

```
extern pascal Boolean PrJobDialog(printRecordHandle)
PrRecHndl     printRecordHandle;
```

$0E13 PrOpenDoc

Initializes a GrafPort for use in printing a document, makes it the current port, and returns a pointer to the port.

Important
You must balance every call to PrOpenDoc with a call to PrCloseDoc.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
--*printRecordHandle*--	**Long**—HANDLE to print record
-- *printGrafPortPtr* --	**Long**—POINTER to GrafPort, if desired; NIL to allocate new GrafPort
	← **SP**

Stack after call

previous contents	
-- *printGrafPortPtr* --	**Long**—POINTER to GrafPort
	← **SP**

Errors

$1302	`portNotOn`	Specified port not selected in Control Panel
$1304	`badLaserPrep`	The version of the LaserPrep file in the LaserWriter is not compatible with this version of the Print Manager
$1305	`badLPFile`	The version of the LaserPrep file in the DRIVERS subdirectory of the SYSTEM subdirectory is not compatible with this version of the Print Manager
$1306	`papConnNotOpen`	Connection can't be established with the LaserWriter
$1307	`papReadWriteErr`	Read-write error on the LaserWriter
Memory Manager errors		Returned unchanged
ProDOS errors		Returned unchanged

C

```
extern pascal GrafPortPtr PrOpenDoc(printRecordHandle,printGrafPortPtr)
PrRecHndl     printRecordHandle;
GrafPortPtr     printGrafPortPtr;
```

More about PrOpenDoc parameters

The print record with the *printRecordHandle* will be used for this printing operation; you should already have validated this print record.

Depending on the setting of the *bJDocLoop* field in the job subrecord, the GrafPort to be used for printing will be set up for draft or spool printing. For spool printing, the spool file name, volume reference number, and version number are taken from the job subrecord.

The *printGrafPortPtr* parameter is normally NIL. If the application passes a pointer to its own Grafport, the Print Manager will use it; otherwise, the routine allocates a new GrafPort and returns a pointer to that GrafPort.

$1013 PrOpenPage

Begins a new page. The page is printed only if it falls within the page range given in the job subrecord.

Important
You must balance every call to PrOpenPage with a call to PrClosePage.

For spool printing, *pageFramePtr* points to a rectangle to be used as the QuickDraw II picture frame for this page. This rectangle can be used for scaling. When you print the spooled document, this rectangle will be scaled (with the QuickDraw II Auxiliary DrawPicture routine) to coincide with the *rPage* rectangle in the printer information subrecord. Unless you want the printout to be scaled, you should set *pageFramePtr* to NIL. This uses the *rPage* rectangle as the picture frame, so that the page will be printed with no scaling.

Important
Don't call the QuickDraw II Auxiliary routine OpenPicture while a page is open (after a PrOpenPage call and before the following PrClosePage call). You can, however, call the QuickDraw II Auxiliary routine DrawPicture at any time.

Also, the GrafPort is completely reinitialized by PrOpenPage. Therefore, after a PrOpenPage call, you must set GrafPort features such as the font and font size for every page that you draw.

Parameters

Stack before call

previous contents	
-- *printGrafPortPtr* --	**Long**—POINTER to GrafPort being used for printing
-- *pageFramePtr* --	**Long**—POINTER to scaling rectangle; NIL for none
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$1302	`portNotOn`	Specified port not selected in Control Panel
Memory Manager errors		Returned unchanged

C

```
extern pascal void PrOpenPage(printGrafPortPtr,pageFramePtr)

GrafPortPtr     printGrafPortPtr;

Rect *pageFramePtr;
```

$1213 PrPicFile

Prints a spooled document. If spool printing is being used, your application should normally call PrPicFile after it calls the PrCloseDoc routine.

The print record should be the printer record already used for this job. The spool file name and version number are taken from the job subrecord of this print record. After printing is successfully completed, the Print Manager deletes the spool file from the disk.

You'll normally pass NIL for the *printGrafPortPtr*. The Print Manager will then allocate the memory blocks it needs from the Memory Manager.

Important

Be sure not to pass a pointer to the same GrafPort you received from PrOpenDoc. If that port was allocated by the PrOpenDoc routine itself (that is, if the printGrafPortPtr to PrOpenDoc was NIL), then the PrCloseDoc routine will have disposed of the port, making your pointer to it invalid. If you provided your own memory for PrOpenDoc, you can use the same memory again for PrPicFile.

Parameters

Stack before call

previous contents	
--*printRecordHandle*--	**Long**—HANDLE to print record
-- *printGrafPortPtr* --	**Long**—POINTER to printing GrafPort; NIL for default allocation
-- *statusRecPtr* --	**Long**—POINTER to printer status record (see Figure 15-13)
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$1302	`portNotOn`	Specified port not selected in Control Panel
Memory Manager errors		Returned unchanged

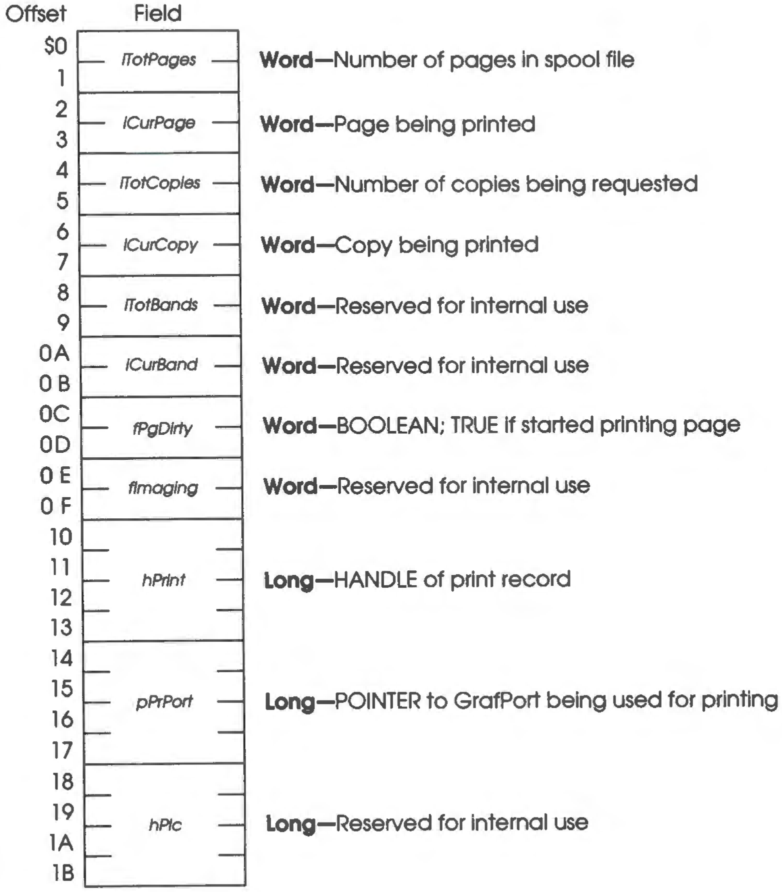
C

```
extern pascal void PrPicfile(printRecordHandle,printGrafPortPtr,
statusRecPtr)
PrRecHndl     printRecordHandle;
GrafPortPtr     printGrafPortPtr;
PrStatusRecPtr     statusRecPtr;
```

Printer status record

The PrPicFile routine uses the printer status record pointed to by *statusRecPtr* to report on its progress. The record is 28 bytes long and is shown in Figure 15-13. Your background procedures (if any) can use this record to monitor the state of the printing operation.

The *fPgDirty* field is TRUE if anything has already been printed on the current page; otherwise, the field is FALSE.



Offset	Field	
$0 1	*lTotPages*	**Word**—Number of pages in spool file
2 3	*lCurPage*	**Word**—Page being printed
4 5	*lTotCopies*	**Word**—Number of copies being requested
6 7	*lCurCopy*	**Word**—Copy being printed
8 9	*lTotBands*	**Word**—Reserved for internal use
0A 0B	*lCurBand*	**Word**—Reserved for internal use
0C 0D	*fPgDirty*	**Word**—BOOLEAN; TRUE if started printing page
0E 0F	*fImaging*	**Word**—Reserved for internal use
10 11 12 13	*hPrint*	**Long**—HANDLE of print record
14 15 16 17	*pPrPort*	**Long**—POINTER to GrafPort being used for printing
18 19 1A 1B	*hPic*	**Long**—Reserved for internal use

Figure 15-13
Printer status record

$0D13 PrPixelMap

Prints all or part of a specified pixel map.

Parameters

Stack before call

previous contents	
— *srcLocPtr* —	**Long**—POINTER to source LocInfo containing POINTER to pixel map
— *srcRectPtr* —	**Long**—POINTER to rectangle enclosing portion of pixel map to print
colorFlag	**Word**—BOOLEAN; TRUE if printing in color, FALSE if in black and white
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$1302	`portNotOn`	Specified port not selected in Control Panel
$1304	`badLaserPrep`	The version of the LaserPrep file in the LaserWriter is not compatible with this version of the Print Manager
$1305	`badLPFile`	The version of the LaserPrep file in the DRIVERS subdirectory of the SYSTEM subdirectory is not compatible with this version of the Print Manager
$1306	`papConnNotOpen`	Connection can't be established with the LaserWriter
$1307	`papReadWriteErr`	Read-write error on the LaserWriter
Memory Manager errors		Returned unchanged
QuickDraw II errors		Returned unchanged

C

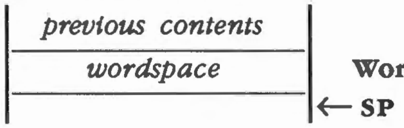
```
extern pascal void PrPixelMap(srcLocPtr,srcRectPtr,colorFlag)

LocInfoPtr     srcLocPtr;

RectPtr      srcRectPtr;

Boolean              colorFlag;
```

$2413 PrPortVer

Returns the version number of the currently installed port driver.

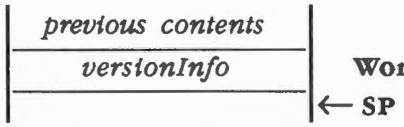
Parameters

Stack before call



Word—Space for result

Stack after call



Word—Version number of the port driver

Errors

None

C

```
extern pascal Word PrPortVer()
```

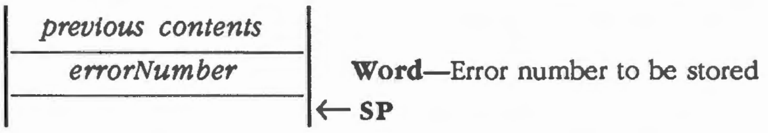
$1513 PrSetError

Stores a specified value into the global variable where the Print Manager keeps its printer error code.

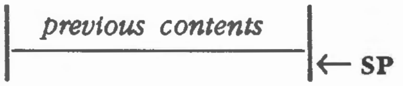
For example, you can use this procedure to abort a printing operation in progress by setting the error code to `prAbort`.

Parameters

Stack before call



Stack after call



Errors

None

C

```
extern pascal void PrSetError(errorNumber)

Word     errorNumber;
```

$0B13 PrStlDialog

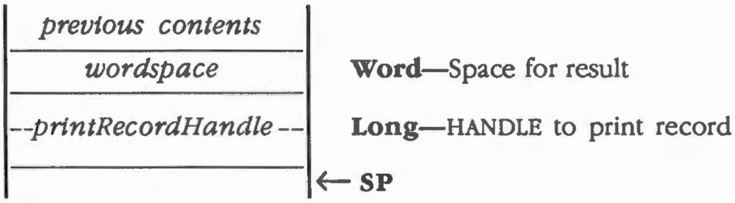
Conducts a style dialog with the user to determine the page dimensions and other information needed for page setup. The initial settings displayed in the dialog box are taken from the print record. If the user confirms the dialog, the results of the dialog are saved in the specified print record, the PrValidate routine is automatically called, and the routine returns TRUE. Otherwise, the print record is left unchanged, and the routine returns FALSE.

Important
Never call PrStlDialog between PrOpenPage and PrClosePage calls.

❖ *Note:* If the print record was taken from a document, you should update its contents in the document if confirmFlag is TRUE. This causes the results of the style dialog to remain with the document.

Parameters

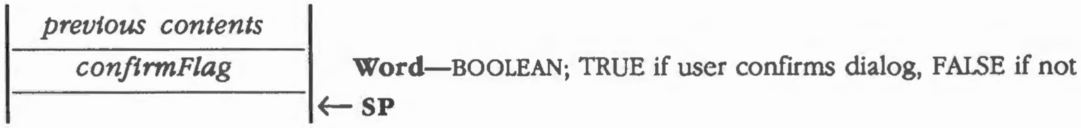
Stack before call



Word—Space for result

Long—HANDLE to print record

Stack after call



Word—BOOLEAN; TRUE if user confirms dialog, FALSE if not

Errors

Memory Manager errors	Returned unchanged

C

```
extern pascal Boolean PrStlDialog(printRecordHandle)

PrRecHndl     printRecordHandle;
```

$0A13 PrValidate

Checks the contents of the specified print record for compatibility with the current version number of the Print Manager and the currently installed printer. If the record is valid, the routine returns FALSE (no change). If the record is invalid, the record is adjusted to the default values for the current printer, and the routine returns TRUE.

Important
Never call PrValidate (or PrStlDialog or PrJobDialog, which call it) between PrOpenPage and PrClosePage calls.

If the current print record is set for an ImageWriter while a LaserWriter is selected or vice versa, PrValidate calls PrDefault to set the print record for the appropriate printer.

PrValidate also makes sure all the information in the print record is internally self-consistent and updates the print record as necessary. These changes do not affect the routine's Boolean result.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
--*printRecordHandle*--	**Long**—HANDLE to print record
	← **SP**

Stack after call

previous contents	
recAdjustFlag	**Word**—BOOLEAN; TRUE if record adjusted, FALSE if no change
	← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal Boolean PrValidate(printRecordHandle)
PrRecHndl     printRecordHandle;
```

Print Manager summary

This section briefly summarizes the constants, data structures, and tool set error codes contained in the Print Manager.

Important
These definitions are provided in the appropriate interface file.

Table 15-5
Print Manager constants

Name	Value	Description
Printer error codes		
`prAbort`	$80	Abort printing

Table 15-6
Print Manager data structures

Name	Offset	Type	Definition
Print record (PrRec)			
prVersion	$0	Word	Version number of the printer driver
prInfo	$2	PrInfoRec	Printer information subrecord
rPaper	$10	Rect	Paper rectangle
prStl	$18	PrStyleRec	Style subrecord
prInfoPT	$2A	14 bytes	Reserved for internal use
prXInfo	$38	24 bytes	Reserved for internal use
prJob	$50	PrJobRec	Job subrecord
printX	$64	38 bytes	Reserved for internal use
iReserved	$8A	Word	Reserved for internal use
Printer information subrecord (PrInfoRec)			
iDev	$0	Word	Printer type
iVRes	$2	Word	Vertical resolution of printer
iHRes	$4	Word	Horizontal resolution of printer
rPage	$6	Rect	Page rectangle

(continued)

Table 15-6 (continued)
Print Manager data structures

Name	Offset	Type	Definition
Printer style subrecord (PrStyleRec)			
wDev	$0	Word	Output-quality information
internA	$2	3 Words	Reserved for internal use
feed	$8	Word	Paper feed type
paperType	$A	Word	Paper type
crWidth	$C	Word	Carriage width for ImageWriter
vSizing	$C	Word	Vertical sizing for LaserWriter
reduction	$E	Word	Percent reduction, LaserWriter only
internB	$10	Word	Reserved for internal use
Job information subrecord (PrJobRec)			
iFstPage	$0	Word	First page to print
iLstPage	$2	Word	Last page to print
iCopies	$4	Word	Number of copies
bJDocLoop	$6	Byte	Printing method
fFromUser	$7	Byte	Reserved for internal use
pIdleProc	$9	Pointer	Pointer to background procedure
pFileName	$D	Pointer	Spool file pathname
iFileVol	$11	Word	Spool file volume reference number
bFileVers	$13	Byte	Spool file version number
bJobX	$14	Byte	Reserved for internal use
Printer status subrecord (PrStatusRec)			
iTotPages	$0	Word	Number of pages in spool file
iCurPage	$2	Word	Page being printed
iTotCopies	$4	Word	Number of copies requested
iCurCopy	$6	Word	Copy being printed
iTotBands	$8	Word	Reserved for internal use
iCurBand	$A	Word	Reserved for internal use
fPgDirty	$C	Boolean	TRUE if started printing page
fImaging	$E	Word	Reserved for internal use
hPrint	$10	PrRecHndl	Handle of print record
pPrPort	$14	GrafPortPtr	Pointer to GrafPort being use for printing
hPic	$18	Long	Reserved for internal use

Note: The actual assembly-language equates have a lowercase *o* (the letter) in front of all of the names given in this table.

Table 15-7
Print Manager error codes

Code	Name	Description
$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
$1304	`badLaserPrep`	The version of the LaserPrep file in the LaserWriter is not compatible with this version of the Print Manager
$1305	`badLPFile`	The version of the LaserPrep file in the DRIVERS subdirectory of the SYSTEM subdirectory is not compatible with this version of the Print Manager
$1306	`papConnNotOpen`	Connection can't be established with the LaserWriter
$1307	`papReadWriteErr`	Read-write error on the LaserWriter
$1321	`startUpAlreadyMade`	LLDStartUp call already made
$1322	`invalidCtlVal`	Invalid control value specified

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using the Apple Macintosh™ Plus and Microsoft® Word. Proof and final pages were created on the Apple LaserWriter® Plus. POSTSCRIPT®, the LaserWriter page-description language, was developed by Adobe Systems Incorporated. Line drawings were created with Adobe Illustrator™.

Text type is ITC Garamond® (a downloadable font distributed by Adobe Systems). Display type is ITC Avant Garde Gothic®. Bullets are ITC Zapf Dingbats®. Program listings are set in Apple Courier, a monospaced font.