Macintosh™

1

PACKING SLIP

X **INSIDE MACINTOSH DOCUMENTATION**

This set used to consist of 3 binders. The original documentation has been revised, and rewritten, and now fits into 2 binders.

The Pascal Workshop documentation and software is now available from your local Apple Dealer.
The Apple part number is A6D0201 .

Note - presently there is no information behind the Introduction, OS Utilities, and Other OS tabs. Also in the "Road Map" section, please disregard any references to Core Edit - this does not exist.

_____ **SOFTWARE SUPPLEMENT**

Note - this installs on the base Lisa Pascal Workshop.

This supplement includes 8 diskettes, with two documentation pieces labeled " The Macintosh Supplement to the Workshop " and " MacCom Instructions " .

THIS SECTION
INTENTIONALLY
LEFT BLANK.

**WHEN AVAILABLE,
IT WILL BE SUPPLIED
AS PART OF THE
MACINTOSH SUPPLEMENT.**

MACINTOSH PUBLICATIONS

Macintosh User Interface Guidelines — USER.INTERFACE/NEWUIDOC

See Also:

Modification History:		
	First Draft	Hoffman 3/17/82
	Rearranged and Revised	Espinosa 5/11/82
	Total Redesign	Espinosa 5/21/82
	Second Edition Prerelease	Espinosa 7/11/82
	Second Edition	Espinosa 10/11/82

ABSTRACT

One of the major factors in making a system pleasant and easy to use is the system's consistency. This specification's purpose is to set down our agreements about the way programs will interact with users, so that we have a common method for dealing with interface problems, and so that all software written for the Macintosh computer (in-house or by outside vendors) will be consistent with respect to the issues discussed here.

CONTENTS

INTRODUCTION

Macintosh is intended to be the first mass-market personal computer. It is designed to appeal to an audience of non-programmers, including people who have traditionally feared and distrusted computers. To achieve this goal, Macintosh must be friendly. The system must, once and for all, dispel any notion that computers are difficult to use. Two key ingredients combine in making a system easy to use: familiarity and consistency.

Familiarity means that the conceptual underpinnings of a system are based on premises or procedures our users already know and employ. Most Macintosh applications are oriented towards common tasks: writing, graphics and paste-up work, ledger sheet arithmetic, chart and graph preparation, and sorting and filing. The actual environment for performing these tasks already exists in people's offices and homes; we mimic that environment to an extent which makes users comfortable with the system. Extensive use of graphics plays an important part in the creation of a familiar and intuitive environment.

Consistency means a uniform way of approaching tasks across applications. For example, when users learn how to insert text into a document, or how to select a column of figures in one application, they should be able to take that knowledge with them into other applications and build upon it. Uniformity and consistency in the user interface reduces frustration and makes a user more amenable to trying new techniques and new software to solve problems.

Consistency and familiarity are by no means orthogonal concepts. Familiar models should be used in a consistent manner to avoid confusion, and consistency should not lead to unfamiliar behavior.

Software Developers' Responsibility

Preservation of a truly consistent working environment requires some deliberate and conscious effort on the part of applications programmers.

If Macintosh is to be successful as a truly mass-market personal computer, software developers must maintain consistency throughout applications by conforming to a common user interface.

(hand)

> It is the responsibility of everyone who writes software for Macintosh to preserve the integrity of the system.

Years of software development, testing, and research have gone into the definition of the Macintosh user interface. The mechanisms outlined in this document have been shown to be well-suited for a variety of applications and tasks. If your application requires approaches not specified in this document, we urge you to build your schemes on top of existing ones and avoid incompatibility at all costs.

Macintosh's Commitment

On many other computers, since little or no user interface aids are built in, each applications programmer invents a new and original interface for each program--which leads to hundreds of different, conflicting, and confusing interfaces.

We hope to avoid this situation on Macintosh by building tools for a versatile, well-tested user interface and placing them in ROM to be used by all applications programs. There's no strict requirement that an applications program must use all or any of the supplied interface tools; but programmers who create their own interface do so at the expense of their own development time, the user's data space, and the entire system's coherency.

Consistency in the user interface is most important in three areas:

- Data selection and editing;
- Command invocation;
- Performance of common system-wide functions.

These are common to all applications. But each application also has its unique requirements, all of which we cannot forsee. To accommodate each application's specific needs, most of the features of the user interface are extensible: a programmer can "customize" the appearance or function of a common interface feature to suit the application.

Macintosh system software is designed to make the implementation of the user interface as simple as possible for the programmer. Most of the recommended user interface features outlined below are implemented with simple calls to the User Interface ToolBox or the Operating System. The substantial documentation available for those packages should serve as an introduction to implementing the user interface described in this document.

About Modes

> "A good man will prefer that mode, by which he can produce the greatest effect."
>
> -- Paley, 1794

We adhere to the principles of modeless behavior. Larry Tesler defines a mode as follows:

> A mode of an interactive computer system is a state of the user interface that lasts for a period of time, is not associated with any particular object, and has no role other than to place an interpretation on operator input.

Modes are most confusing when you're in the wrong one. Unfortunately, this is the most common case. Being in the wrong mode is confusing because it makes future actions contingent upon past ones; it changes the behavior of familiar objects and commands; and it makes habitual actions cause unexpected results.

We advocate avoidance of modes whenever possible. Of course, exceptions must be made, however; there are certain tradeoffs among modality, usefulness, and implementability that must be considered. There are three cases in which modal behavior is generally tolerated:

- Long-term modes with a procedural basis: doing word processing vs. graphics editing, etc. Each application program in Macintosh is a mode.

- Short-term "spring-loaded" modes, in which the user is constantly doing something to perpetuate the mode. Holding down a button or key is the most common example of this kind of mode.

- Alert modes, where the user must rectify an unusual situation before proceeding. Such situations, however, should have been avoided in the first place.

Other modes are acceptable if they meet the following requirements:

- They emulate a familiar real-life model which is itself modal, like picking up different-sized paintbrushes in a graphics editor; or

- They change only the attributes of something, and not its behavior, like the boldface and underline modes of text entry; or

- They block most other normal operations of the system to emphasize the modality, as in error conditions incurable through software ("There's no diskette in the disk drive", for example).

Whatever the modality entails, it must be visible. There must be a clear visual indication of the current mode, and the indication should be near the object being most affected by the mode.

THE GRAPHIC SCREEN

Macintosh distinguishes itself from all other personal computers by its high-resolution graphic screen. While other computers posess similar or greater graphics resolution or ability, no other applies its graphic powers as widely and generally as Macintosh.

Macintosh has a purely graphic display: there is no "text mode" in the machine at all. Text, to Macintosh, is merely a special kind of graphics. Problems of mixing text with graphics go away because they're really the same thing.

Other computers don't do this because of inherent limitations in their processor speed and data path width, and because of a lack of software support of graphics. Not only does Macintosh have a Motorola MC68ØØØ microprocessor (running at a nominal 7 MHz with a 16-bit data path, giving it several times the bandwidth of the Apple II's 65Ø2), but it also has Bill Atkinson's QUICKDRAW graphics package, revolutionary in its speed and ability.

But far more important than raw graphic power is what the software does with it. What Macintosh does can be explained quite simply:

(hand)

All commands, features, and parameters of the application, and all the user's data, appear as graphic objects on the screen.

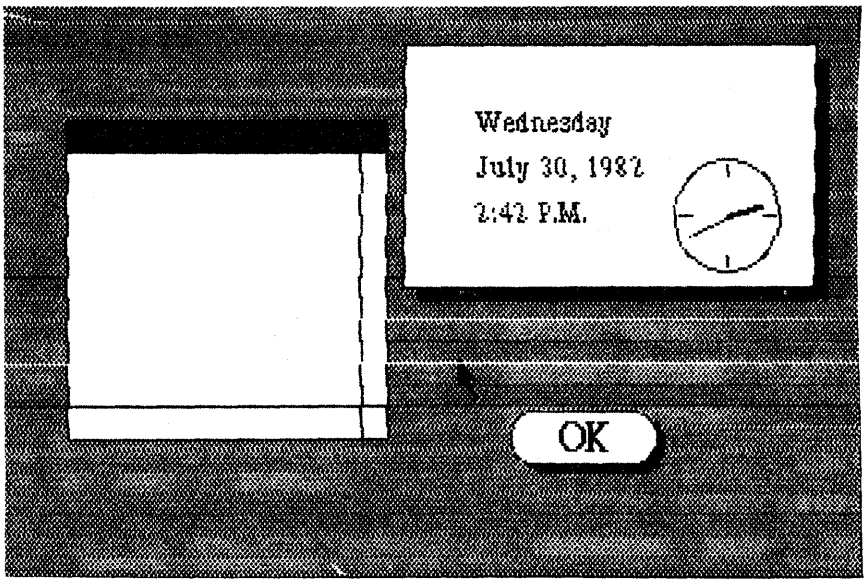


Figure 1. Objects on the Screen

Objects, whenever applicable, resemble the familiar material objects they emulate. Objects that act like pushbuttons "light up" when pressed; objects that act like tab stops look like their counterparts

on a typewriter. Dozens of objects, some emulating everyday objects and some unique to Macintosh, are defined in the User Interface Tool Box.

Objects are designed to look beautiful on the screen. Using the graphic patterns in QuickDraw can give objects a shape and texture beyond simple line graphics. Placing a drop-shadow slightly below and to the right of an object can give it a three-dimensional appearance. The highest aesthetic sensibilities should be used in the design, placement, and animation of objects.

Graphics can distinguish different states of the same object. Many objects on the screen have two states: a "normal" state and a "special" state. Most objects in their normal state are predominantly white, with detail (lettering, symbols, etc.) in black. Inverting the polarity of the object, to make it black with white detail, will highlight the object to represent its special state.

Icons

A fundamental object in Macintosh software is the icon, a small, 32-by-32 square graphic that can be drawn, edited, and moved easily. The Icon Manager has facilities for drawing icons on the screen and setting or resetting bits within them.

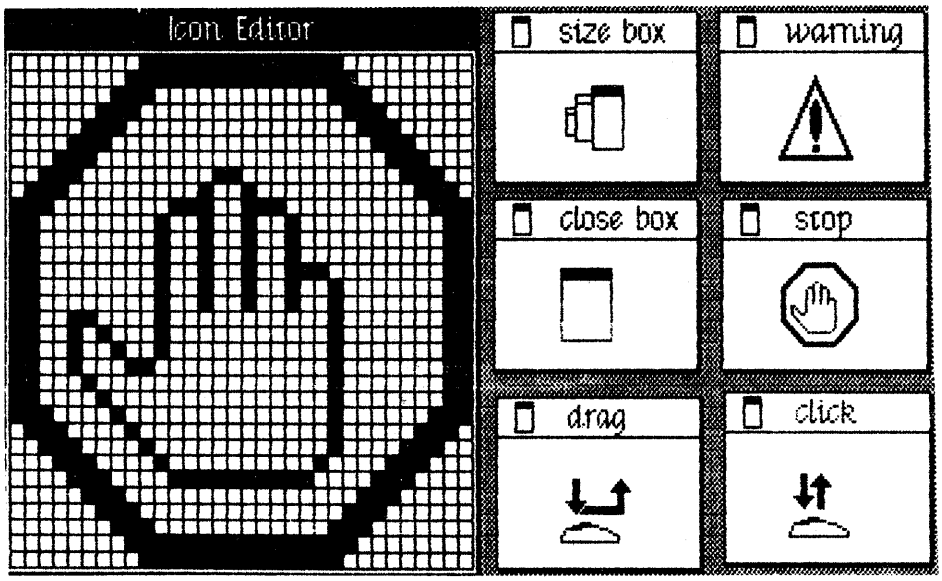


Figure 2. Icons

Icons should be sprinkled liberally over the screen. Wherever an explanation or label is needed, first consider using an icon before using text as the label or explanation. Icons not only contribute to the understandability and attractiveness of the system, they don't need to be translated into foreign languages.

Icons are by no means unique to the software; thy appear on the Macintosh main unit itself, on the shipping materials, unpacking instructions, and in the user manuals. The standard icons used to denote various parts of the Macintosh hardware are shown in the Appendix on icons.

ACCEPTING USER INPUT

All meaningful interaction between a Macintosh and its user takes place via a piece of hardware built in or connected to the main unit. The principal devices for original input to the Macintosh are the mouse and the keyboard; the Macintosh responds to these devices by displaying images on the screen or making sounds with its speaker. No other action of the Macintosh (such as spinning its disk drive, etc.) constitutes a meaningful message to the user.

The Mouse

The mouse is a small device the size of a deck of playing cards, connected to the computer by a long, flexible cable. There is a square button on the top of the mouse. The user holds the mouse and rolls it on a flat, smooth surface.

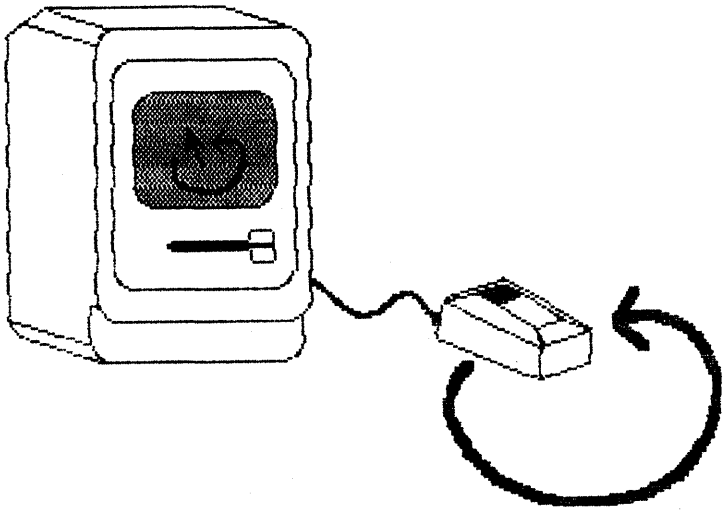


Figure 3. The Mouse and Pointer

A pointer on the screen follows the motion of the mouse. Simply moving the mouse results only in a corresponding movement of the pointer and no other action. Most actions take place when the user positions the focus of the pointer (which should be intuitive, like the point of an arrow or the center of a crosshairs) over an object on the screen and presses the mouse button.

The purpose of the mouse is to allow high-resolution specification of elements on a graphic screen. Many researchers, at Apple and elsewhere, have conducted extensive experimentation with various pointing devices: cursor keys, light pens, graphic tablets, trac balls, etc. We chose the mouse for its ease of use, accuracy, size, and cost. It is compact and lightweight; it resolves to 2ØØ points per inch; it retains its position when not being used; and it requires little muscular strain to position it.

Mouse Actions

The three basic mouse actions are:

- Clicking: Positioning the pointer with the mouse, and briefly pressing and releasing the mouse button without moving the mouse;
- Pressing: Positioning the pointer with the mouse, and pressing and holding the mouse button without moving the mouse; and
- Dragging: Positioning the pointer with the mouse, pressing and holding the mouse button down, moving the mouse to a new position, and releasing the button.

Clicking something with the mouse performs an instantaneous action: selecting a location within the user's document or activating an object.

Pressing an object usually has the same effect as clicking it repeatedly. For example, clicking a scroll arrow causes a document to scroll one line; pressing a scroll arrow causes the document to scroll repeatedly until the mouse button is released.

Dragging can have different effects, depending upon what is under the pointer when the button is pressed. Beginning a drag inside the document frequently results in selection of data. Beginning a drag over an object usually moves that object on the screen. Only certain objects are draggable; large draggable objects have a special area with which the user drags the entire object. Our tests show that users understand dragging an object by a well-marked area rather than by a large, general area.

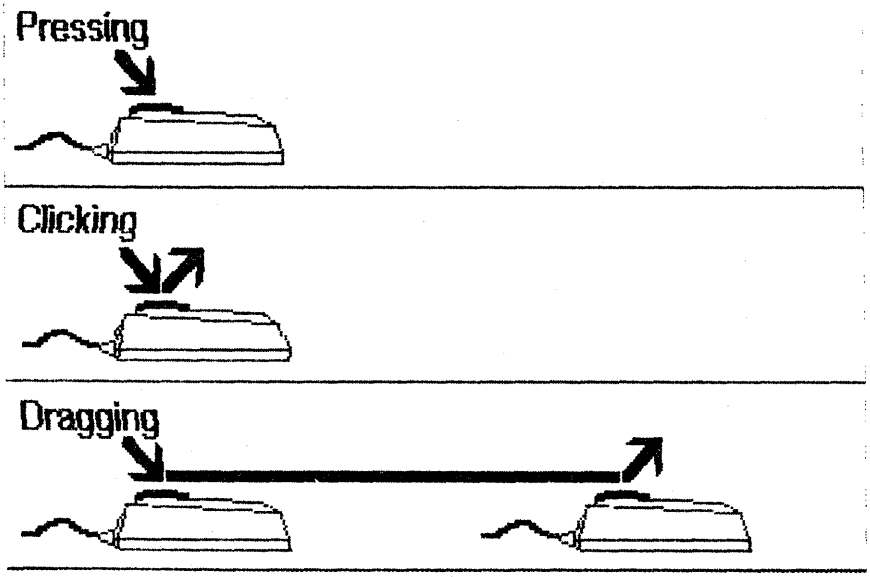


Figure 4. Clicking, Pressing, and Dragging

Dragging is also used to choose an item from a menu, as described below.

(hand)

In general, pushing the mouse button indicates intention, while releasing the button confirms the action.

Dragging an object attaches a flickering outline of the object to the pointer. The outline follows the pointer around the screen while the mouse button is being held down. When the user releases the mouse button, the object moves to the position of the flickering outline, and the outline vanishes.

Every object is restricted to certain boundaries. If the user tries to drag an object out of its natural boundaries, the flickering outline disappears when the pointer travels out of those boundaries. If the user moves the pointer back inside the boundaries with the button still held down, the outline reappears under the pointer and dragging resumes. If, however, the user releases the button while the outline is invisible, the object being dragged does not move; in this way the user can cancel a drag in progress.

Double-Clicking

A variant of clicking involves performing a second click shortly after the end of an initial click. If the downstroke of the second click follows the upstroke of the first by 700 milliseconds or less, the second click should be considered not an independent event, but rather an extension of the first: this action is called "double-clicking". Its most common use is as an optimized means of performing an action that can be performed in another, slower, manner.

(hand)

To allow the software to distinguish efficiently between single clicks and double-clicks on objects that respond to both, a function invoked by double-clicking an object must be an enhancement, superset, or extension of the feature invoked by single-clicking that object.

Changing Pointer Shapes

The pointer may change shape to give feedback on the range of activities that make sense in a particular area of the screen, in a current mode, or both.

1. The results of any mouse action depend on the item under the pointer when the mouse button is pressed. To emphasize the differences among mouse actions, the pointer may assume different appearances in different areas to indicate the mouse's behavior in each area.

2. Although modal behavior is generally discouraged in the Macintosh user interface, sometimes introducing modes makes it simpler to differentiate among the multiplicity of functions of the mouse. For example, in the Graphics Editor, the mouse functions both to draw graphics and to manipulate graphics already drawn. Thus, in this particular application, the mouse is employed in two

different modes. To accent the difference in behavior in these two modes, the pointer may change shape.

The facility to change the pointer appearance to convey modal information is not a unilateral endorsement of modal behavior; see the discussion "About Modes" on page 6 of this document.

The Keyboard

Connected to the Macintosh main unit by a six-foot coil cord is a compact alphanumeric keyboard. The keyboard is used mainly for text and numeric entry.

The keys on the keyboard are arranged in familiar typewriter fashion; there is a utility program with which the user can change the positions of the keys or the characters they generate.

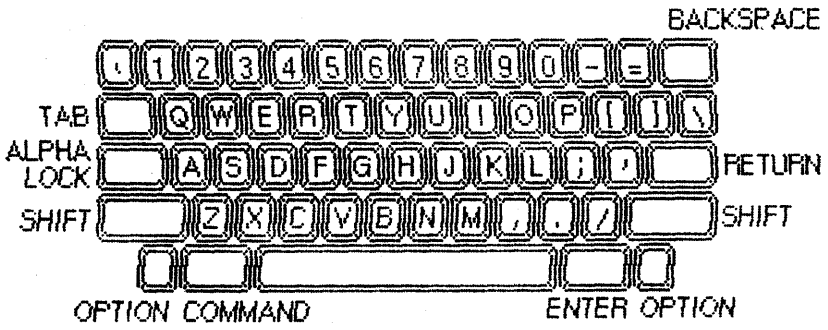


Figure 5. The Macintosh Keyboard

In terms of functionality, the keys are divided into three sets: character keys, modifier keys, and special keys. Character keys enter characters into the computer; modifier keys, in conjunction with character keys, choose among different characters on a key; and special keys give special instructions to the computer.

Character Keys

The alphabetic, numeric, and symbolic keys, and the space bar, enter characters into the computer. Any character key may be associated (and/or labeled) with more than one character; the modifier keys choose among the different characters on each key.

The Basic Editing Paradigms (see that section) define the ways in which characters are typed into a document. All text, whether it be a file name, part of a document, or a search pattern, is typed in and can be edited in exactly the same way.

The keyboard hardware scans the character keys such that it can recognize any two character keys being pressed simultaneously. This feature is called "two-key rollover".

Modifier Keys: SHIFT, CAPS LOCK, OPTION, and COMMAND

Six keys on the keyboard--two labeled SHIFT, two labeled OPTION, one labeled CAPS LOCK, and one labeled COMMAND--change the interpretation of keystrokes or other inputs to the computer. When one of these keys is held down, the behavior of the other keys (and occasionally that of the mouse button) may change. A program can enquire the status of the modifier keys at any time.

The SHIFT and OPTION keys choose among the characters on each character key. SHIFT gives the upper character on two-character keys, or the uppercase letter on alphabetic keys. OPTION gives an alternate character set interpretation, for foreign characters, special symbols, etc. SHIFT and OPTION can be used in combination.

CAPS LOCK latches in the down position when pressed, and releases when pressed again. When down it gives the uppercase letter on alphabetic keys. The operation of CAPS LOCK on alphabetic keys is parallel to that of the SHIFT key, and the CAPS LOCK key has no effect whatsoever on any of the other keys. CAPS LOCK and OPTION can be used in combination on alphabetic keys.

The keyboard hardware can sense any or all of the modifier keys being pressed simultaneously.

The COMMAND key

Pressing a key while holding down the COMMAND key signals that the keypress is not data input, but rather a command invocation (see the section on Commands).

(hand)

> As the OPTION and COMMAND keys are unfamiliar features to users familiar with typewriters, their use should be restricted to expert functions not normally encountered by novice users.

Special keys: ENTER, TAB, RETURN, and BACKSPACE

When the user enters or edits information, the ENTER key confirms that entry. When ENTER is pressed, the computer checks and validates the current entry and allows the user to proceed to a different one. Commonly used to confirm the entry of text, ENTER tells the computer to accept changes made to a field or form (such as a spreadsheet formula or a new file name).

The TAB key is a signal to proceed: it signals movement to the next item in a sequence. TAB often carries the implicit meaning of ENTER before the motion is performed.

The RETURN key is another signal to proceed, but it defines a different type of motion than TAB. A press of the RETURN key signals movement to the leftmost field one step down (just like a carriage return on a typewriter). RETURN also can carry the implicit meaning of ENTER before it performs the movement.

(hand)

> In applications such as the word processor, the TAB and RETURN keys not only perform immediate actions, but store those actions in the text; in such applications the RETURN and TAB keys may be considered character keys.

BACKSPACE is used to delete characters from text, usually in the course of typing that text. The exact use of BACKSPACE is described in the section on the Basic Editing Paradigms.

Typeahead, Auto-Repeat, and Audio Feedback

If the user types at a time when Macintosh is unable to process the keypresses immediately, or the user types more quickly than Macintosh can process, the precocious keystrokes are queued for timely processing. As keystrokes are handled as events through the Operating System's event mechanism, the only limit on the number of characters that can be typed ahead of time is the length of the system's event queue.

Normally, Macintosh "clicks" slightly at every keystroke. This audio feedback in typing is a global preference that the user can change at any time (see the Preferences description, in the section on Desk Accessories).

When the user holds down a key for a certain amount of time, it starts repeating automatically. The delays and the rates of repetition are global preferences that can be changed by the user at any time.

All printable characters, the space bar, the BACKSPACE key, and the RETURN key, inherently have the auto-repeat ability. The auto-repeat ability of each key is a characteristic of the keyboard that the user can change with the same utility program that alters the keyboard layout.

Auto-repeat does not function during typeahead; it only operates when the application is ready to accept keyboard input.

Versions of the Keyboard

There are two physical versions of the keyboard: American and European. The European version has one more key than the American. The key layout on the European version is designed to conform to the ISO standard; the American key layout mimics that of common American office typewriters.

The American keyboard contains 49 character keys (including the space bar and RETURN) that produce all the printable ASCII characters. In

addition, there are the following modifier and special keys: SHIFT, CAPS LOCK, COMMAND, OPTION, ENTER, TAB, and BACKSPACE.

The European keyboard contains 50 character keys; the special and modifier keys are equivalent to those on the American keyboard, but their labels denote their functions symbolically.

(hand)

As the keyboard interface is a general-purpose clocked bidirectional serial port, other devices (such as a music keyboard, etc.) may eventually be attached to this port.

The Numeric Keypad

An optional numeric keypad is offered that connects between the main unit and the standard keyboard. The keypad contains 18 keys that, while labeled similarly to keys on the main keyboard, return different keycodes to the main unit. An application can thus determine the origin of a keystroke. If desired, the keypad keys can be assigned ASCII codes equivalent to their counterparts on the main keyboard.

The character keys on the keypad are labeled with the digits 0 through 9, a decimal point, the four standard arithmetic operators for plus, minus, times, and divide, and a comma. The keypad also contains the special keys ENTER and CLEAR; it has no modifier keys.

The keys on the numeric keypad follow the same rules for typeahead, auto-repeat, and audio feedback as the main keyboard.

Four keys on the numeric keypad are labeled with "field-motion" symbols: small rectangles with arrows exiting them in various directions. Some applications may use these keys to move an object or indicator orthogonally around the screen, and require the user to use the SHIFT key to obtain the four characters (+ * / ,) normally available on those keys.

(hand)

As the numeric keypad is optional equipment, no application shall require it or any keys available on it in order to perform standard functions. Specifically, as the CLEAR key is not available on the main keybaord, a CLEAR function may be implemented with this key only as an optimization of another CLEAR command (such as in a menu).

CONCEPTUAL MODELS

Macintosh, as an appliance computer, has one purpose only: to manipulate information. With it, a user can access, display, interpret, modify, transfer, replicate, and destroy information. Consequently, the central concepts on which the Macintosh system is built deal with things relating to information:

- The container of information, which we call a file;
- The manipulator of information, which we call a tool;
- The presenter and interpreter of information, which we call a window;
- The working environment, which we call the desk top; and
- The information itself, which we call a document.

On the continuum between pure concept and pure object, each of these has its own place. We hope to present our users with only the physical objects that represent these concepts, so that they can grasp the concepts by inference; we will not require them to know the concepts before they encounter any of the objects.

Of the above, files are the most conceptual; we will use the term internally here to mean a generic container of information. As described below, files have many distinct incarnations that the user will encounter.

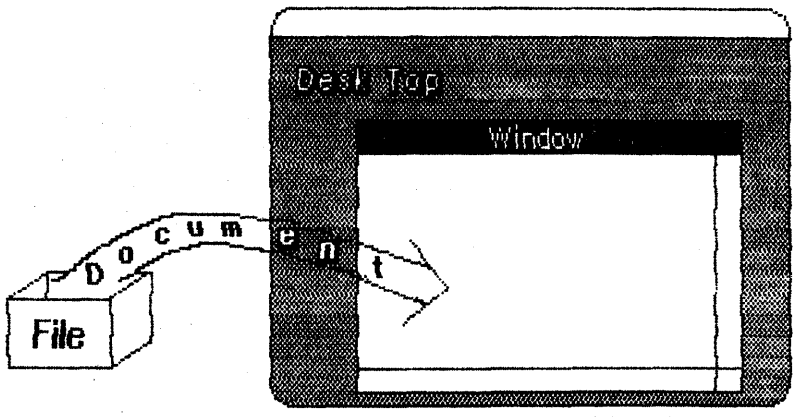


Figure 6. Conceptual Models

The desk top, documents, and windows are the most concrete of the above group: users will see these as objects and not as concepts at all.

Tools are somewhere in the middle: although they have certain distinguishable physical attributes, most of their importance is in the conceptual realm.

Files

A file is a container of information. All the texts, pictures, charts, and address lists that the user puts into Macintosh are stored in files. Files also store information that the user didn't create: information usually more intelligible to the computer than the user.

There are three general classifications for files: those containing documents, those containing tools, and those containing resources. Documents are created by users and can be viewed and modified by users. Tools are created by application programmers; the user can use them but can't modify them. Resources are also created by application programmers, but can be edited by a resource editor to change the way in which a tool communicates with the user (see RESOURCE FILES, below).

Regardless of its contents, a file has many important attributes. Every file has a type that describes its contents and determines which tools can manipulate it; a size that describes how large its contents are; a name by which the user refers to the file; and a label on which the user can put additional information about the file. It also has the dates on which it was created and last modified.

Tools

What we call a "tool" is generally known as an application program: an interactive set of procedures and data structures for manipulating information. Writing, drawing, charting, filing, analysis, and BASIC programming are the fundamental tools Macintosh provides; there are also several other "housekeeping" tools, like using a pocket calculator, note pad, and several other "mini-applications" described later in this document. A tool manifests itself in two ways: it displays a menu bar replete with menus of commands appropriate to that tool; and it places a document window on the desk through which the user can see the information contained in a file.

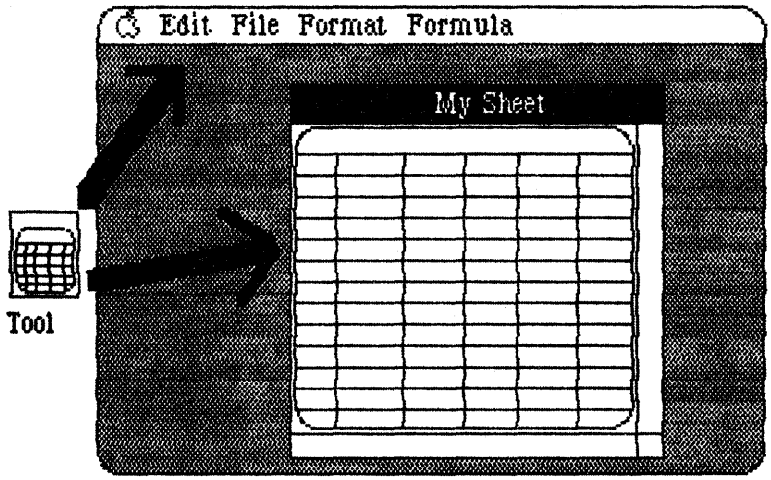


Figure 7. A Typical Tool

Tools, being themselves information (but intelligible to the computer rather than to the user), are stored in files.

(hand)
Only one tool can be in use at any given time.

Documents

Documents are the information that the user has created or wishes to manipulate. Documents can exist inside a file on a diskette or inside the memory of Macintosh. A document comprises a coherent set of different kinds of information.

- Most documents comprise only one kind of information: all text, or one picture, or a series of charts, for example. The user manipulates the information and prints it out as a whole. Every document thus has a principal "type" of information; this type is determined by the tool that formed it.

- A document can comprise more than one kind of information, but it must still form a coherent whole. The user can take information of one kind and add it to a document of another kind. But the document still retains its principal type, and the user can manipulate only the information of that type.

Associated with each kind of document is a principal tool: the tool most appropriate to manipulate that document. The principal tool of any document is usually the tool that created it. Other tools may be able to read and interpret the document; for example, the BASIC language can read word processor documents anticipating the text of a program. Such tools are the document's secondary tools. The distinction is important only when selecting files from the desk.

Resources

Some files contain information that is neither a tool nor the user's information. This information is usually fonts, system programs, configuration information, etc. Although such information may have principal tools (such as a font editor for fonts), it's most commonly used by a tool.

Files containing such information are called resource files. Tools have internal links to the resource files they need; copying a tool file, for example, automatically copies all resource files linked to it. Resource files are usually created by application programmers to accompany tools. The user can edit some resource files by using special resource editors, such as font editors or menu editors.

THE DESK TOP MODEL OF ORGANIZATION

The entire Macintosh working environment is based on familiar and intuitive concepts. The Macintosh screen represents a working surface or a desk top. Papers, writing or drawing utensils, and other common desk accessories have their place on this desk top just as on any other. Whenever possible, the objects on the desk top resemble their real-life counterparts: for example, all papers are white with black lettering.

Figure 8. The Desk Top

The Desk

The desk top metaphor is reinforced by the central tool of the Macintosh system, a tool called the Desk. While most tools manipulate the documents contained in files, the Desk manipulates the files themselves, often regardless of their contents. The basic functions of the Desk are as follows:

- Get, Print, Examine, Delete, or Copy any file or group of files;
- Initialize a diskette;
- Rename or rearrange the files on a diskette;
- Select which diskettes, network diskettes, and peripheral devices to work with.

On the Desk, files are represented by icons, with each file's name as a caption to its icon. The icons can be dragged around the desk asnd positioned in any order or arrangement. Other parts of the system are also represented by icons on the desk: disks and disk drives, printers, etc.

The central purpose of the Desk is to allow the user to manipulate files, and to call up the appropriate tools to work on the documents the files contain. The user invokes a tool from the Desk, and returns to the Desk when finished.

Once using a tool, the user can call up a subset of the standard Desk functions, to choose a new file to work with or to specify a destination for the new work. This subset as presently defined includes selecting disks and files, creating a new file, and renaming and repositioning files.

WINDOWS

Windows are objects on the desk that display information. The information can be a user's document, an error message, or a request for more information. Any number of windows can be present on the desk at any time. As on a real desk, if more windows are placed on the desk than reasonably can fit, the windows "overlap" each other: the windows in front partially or completely obscure those behind them.

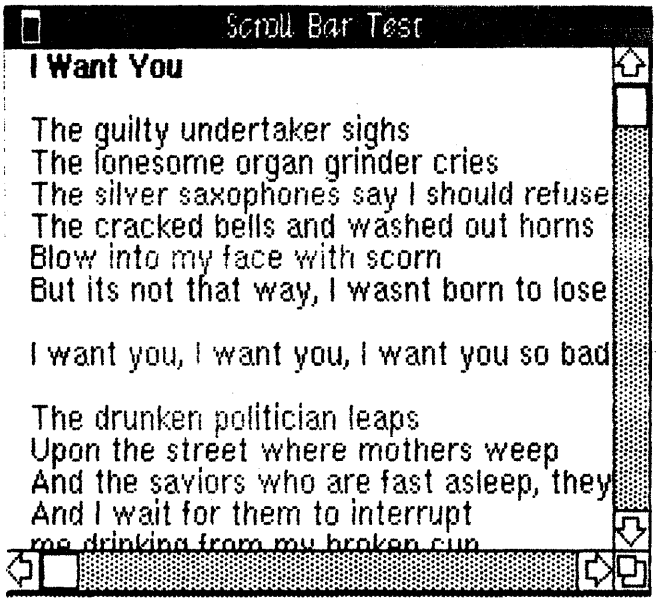


Figure 9. Windows

Each window "floats" in its own plane. Think of a number of plates of glass stacked on top of the desk: each plate contains one and only one window, and the plates can be moved to make the windows appear in different places on the desk. Each window can overlap those behind it, and can be overlapped by those in front of it. The frontmost window cannot be overlapped. Even when windows do not overlap, they retain their front-to-back ordering.

Opening and Closing Windows

Windows come up onto the screen in different ways appropriate to the purpose of the window. Some windows are created automatically: for example, when the user wants to work with a document, the tool being used creates a document window in which to present that document.

Many windows have an icon that, when double-clicked, makes the window go away: this icon is called the close box. (This icon is double-clicked, rather than singly-clicked, because of the disturbing consequences of accidentally clicking the icon). The application in control of the window determines what is done with the window when the close box is double-clicked: it can

1. make the window invisible, to be retrieved later; or

2. remove and destroy the window and any information it contained.

If an application wishes not to support closing its window with a close boxe, it should not place the box on the window.

The Active Window

At any given time, one window is of greater importance to the user than any other. Usually, the most important window is presenting the current document; at other times, an error message or information request may be more important. Thus this general rule:

- The most important window at any given time is always frontmost.

Naturally, there must be rules to determine which window is most important at any given time.

- Newly-created windows are usually brought to the front.

- If the user positions the pointer with the mouse inside any window that is not in front, and then clicks the mouse button, that window is brought to the front.

Being in front has more consequences for an window than merely being more visible. The frontmost window is said to be active, and all others inactive.

- A window's active state is visibly distinct from its inactive state; usually, the title or header of the window is highlighted.

- Clicking or dragging inside the active window may perform a useful function; clicking or dragging inside an inactive window merely brings that window to the front.

- All command and data input is handled by the program that is in control of the active window.

Document Windows

Although windows display many kinds of information and requests, the most common appearance of a window is to display the document currently being worked on. Windows displaying documents have parts not usually seen in other windows: scroll bars to move the document under the window; a size box to change the size of the window; and split bars to divide the window into several panels.

Scroll Bars

Scroll bars are used to change the user's view of a document. Only the active window has scroll bars; inactive windows leave black-bordered empty rectangles where their scroll bars will appear when the window is activated.

A scroll bar is a light gray shaft, capped on each end with square boxes labeled with arrows; inside the shaft is a white rectangle. The shaft represents one dimension of the entire document; the white rectangle (called the thumb) represents the portion of the document currently visible inside the window. As the user moves the document under the window, the position of the rectangle in the shaft moves correspondingly.

There are three ways to move the document under the window: by sequential scrolling, by "paging" screenful by screenful through the document, and by directly positioning the thumb.

Clicking a scroll arrow moves the document in the direction of the scroll arrow. For example, when the user clicks the top scroll arrow, the document moves down, bringing the view closer to the top of the document. The thumb moves towards the arrow being clicked.

Each click in a scroll arrow causes movement a distance of one unit in the chosen direction, with the unit of distance being appropriate to the tool: one line for the word processor, one row or column for the spreadsheet, etc. Pressing the scroll arrow causes continuous movement in its direction.

Clicking the mouse anywhere in the gray area of the shaft advances the document by screenfuls. The thumb moves toward the place where the user clicked, and the document moves in the opposite direction; clicking below the thumb, for example, brings the user the next screenful towards the bottom of the document. Pressing in the gray area keeps screenfuls flipping by until the user releases the button or the thumb reaches the pointer.

In both the above schemes the user moves the document incrementally until it is in the proper position under the window; as the document moves, the thumb moves accordingly. The user can also move the document directly to any position simply by moving the thumb to the corresponding position in the shaft. To move the thumb, the user presses on the thumb and drags it along the shaft; a flickering outline of the thumb follows the pointer. When the mouse button is released, the thumb jumps to the position last held by the flickering outline, and the document jumps to the position corresponding to the new position of the thumb.

If the user starts dragging the thumb, and then moves the pointer a certain distance outside the scroll bar, the thumb detaches itself from the pointer and stops following it; if the user releases the mouse button, the thumb returns to its original position and the document remains unmoved. But if the user still holds the mouse button and drags the pointer back into the shaft, the thumb reattaches itself to the pointer and can be dragged as usual.

Multiple Windows

Some tools may be able to keep several windows on the desk at the same time, as part of the same logical document. Different windows can represent:

- Different parts of the same document, such as the beginning and end of a long term paper;
- Different interpretations of the same document, such as the tabular and chart forms of a set of numerical data;
- Different parts of a logical whole, like the listing, execution, and debugging of a BASIC program;
- Separate documents being viewed and/or edited simultaneously.

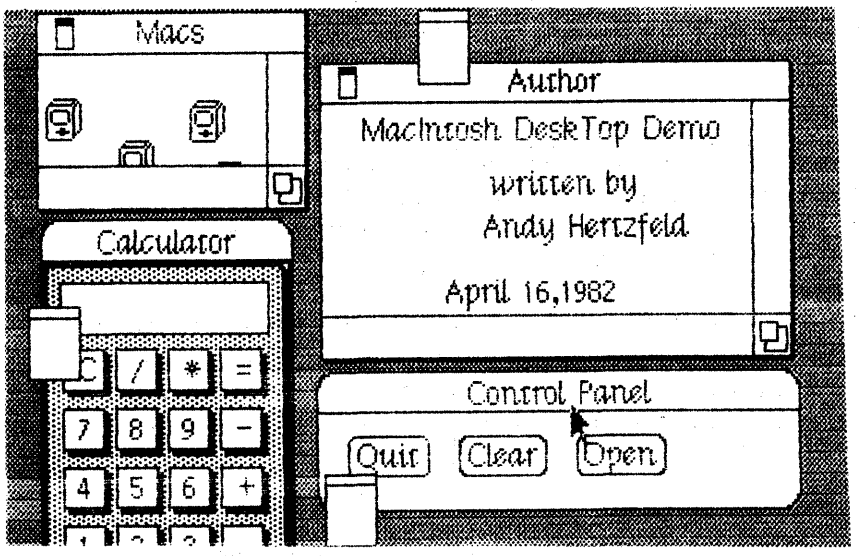


Figure 10. Multiple Windows

Each tool may deal with the meaning and creation of multiple windows in its own way.

There are occasionally better ways to perform the above functions than with multiple windows. Showing different parts of the same document can be done better by splitting the window (see below). Different interpretations of the same document occasionally merit two panes in the same window, rather than two separate windows. The implementation decision can best be made by experimentation and testing on actual users.

Moving a Window

Each tool places windows on the screen wherever it wants them. The user can move a window--to make more room on the desk or to uncover a window it's overlapping--by dragging its title bar. A flickering outline of the window follows the pointer until the user releases the

mouse button. At the release of the button the full window is drawn in its new location.

A window always moves in its own plane; while it's being dragged around, the flickering outline is visible over the windows below it but is hidden under the windows above. Notice that clicking in the title area does not make a window active or bring it to the top.

(hand)
Moving a window does not affect what portion of the document it is displaying.

A window can never be moved off the screen; specifically, it can't be moved such that the visible area of the title bar is less than four pixels square.

Moving a window is fully supported by the Window Manager, and is easily performed with one procedure call; an application program need not care where on the screen its window is placed.

Changing the Size of a Window

If a window has a certain icon in its lower right corner, where the scroll bars come together, the user can change the size of the window--enlarging or reducing it to the desired size. The box that contains the icon is called the size box.

Dragging the size box drags a flickering outline of the window. The outline's top left corner stays fixed, while the bottom right corner follows the pointer. When the mouse button is released, the entire window is redrawn in the size and form of the flickering outline.

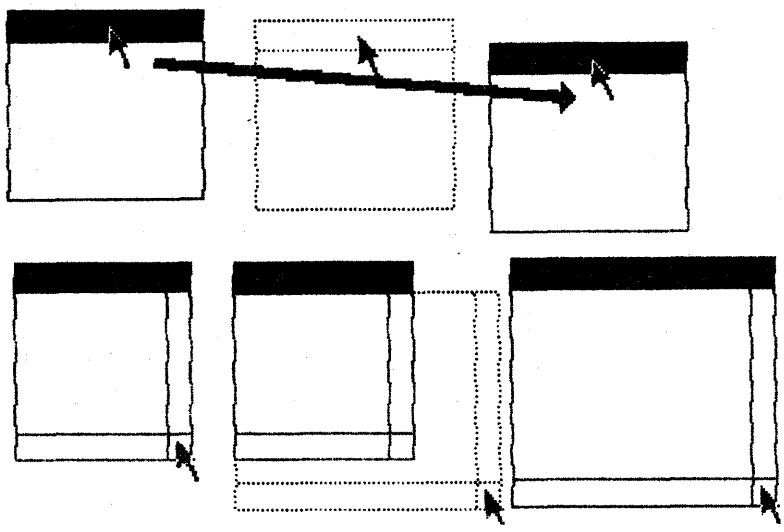


Figure 11. Moving and Sizing a Window

Sometimes it's not appropriate to size a window; some tools may not support this ability. In this case, the size box is empty and clicking

in it produces no effect. If a tool does support sizing a window, however, changing the window's size leaves the document's size unchanged; the window simply displays a larger or smaller portion of the document.

(hand)

> Sizing a window does not affect its contents, or change the position of the top left corner of the window over the document; only the portion of the view that is visible inside the window.

At its maximum size, a window is still small enough that a seven pixel square area of the size box is visible on the screen.

The minimum size window consists of only a title bar the width of the title itself, a horizontal scroll bar (or a blank rectangle of equivalent size), and the size box. If a window is made so small that its title will no longer fit in the title bar, the title is truncated to show as many of its initial characters as possible.

Sizing a window is fully supported by the Window Manager, and is easily performed with one procedure call; an application program need not care about the size of a window.

Splitting a Window

Sometimes it is desirable to be able to see disjoint parts of a document simultaneously. Tools that accommodate such a capability allow the window to be split into independently scrollable panels.

Tools that support split panes place split bars at the top of the vertical scroll bar and at the left of the horizontal one, if present. Pressing a split bar attaches it to the pointer. Dragging the split bar positions it anywhere along the nearby scroll bar; releasing the mouse button drops the split bar at its current position, splits the window at that location, and creates new scroll bars for each panel.

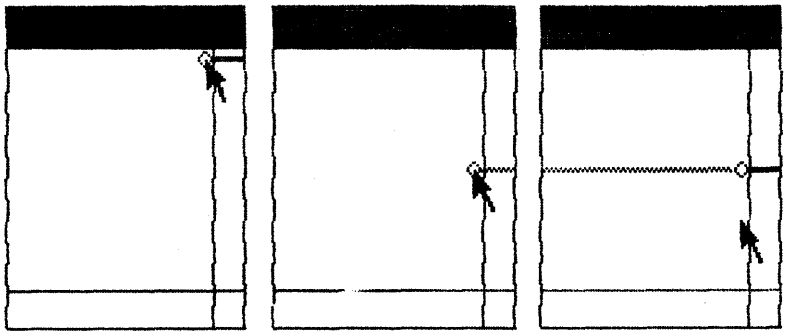


Figure 12. Split Views

The document appears the same, save for the split bar lying across it. But there are now separate scroll bars for each pane; whith these, the user can scroll each pane independently of the other.

Dragging a split bar back to its original position reunites the window in that direction; the left or top view (and its scroll bar) disappears, leaving the right or bottom view.

The number of views in a document does not alter the number of selections per document: i.e., one. The active selection appears highlighted in all views that present it.

Desk Accessories

Macintosh does not allow two tools to be running at once. However, there are several mini-applications that are available while using any tool.

At any time the user can issue a command to call up one of several desk accessories. The basic ones provided include:

Calculator
Alarm Clock
Note Pad
Telegram Form and In-Box (AppleGram)

Accessories are disk-based: only those accessories available on-line can be used. The list of accessories is expanded or reduced according to what's available at any given time. The application can support all accessories in the system with calls to the Desk Manager. On disk, accessories are stored in resource files. More than one accessory can be on the desk at any given time.

Who's on Top?

With a virtual three-dimensional screen it is essential to manage the third dimension so that important items or objects requiring immediate attention are not obscured accidentally. Hence, in order from front to back:

- The pointer
- An alert box
- A dialog box
- The menu bar and all pull-down menus
- The active window
- All other windows
- The desk top

INSIDE DOCUMENTS

(hand)

> We strongly subscribe to the doctrine of preservation of visual fidelity, i.e., what you see is what you get.

It's important that a document as seen through a window on the desk closely resemble the same document when committed to paper. The differences (and there will be differences) must be natural and unsurprising. Naturally, the ruler and graph paper used to create a report on Tuesday morning won't be distributed with that report when it's presented that afternoon; printing a document shall not carry the vestiges of the tool that created that document.

Any given tool should be able to manipulate, in some way, everything in the document it presents. Macintosh eventually will have many different tools, and we do not pretend to foresee the needs of all. However, we do provide standard means of manipulating the constituent elements of most documents.

Structure of Documents

In order to discuss the appearance of information inside documents, it is necessary first to digress a bit into the structure of documents.

A document is a collection of information. Each piece of information has its own position in the document, and its own positional relationship to the information around it.

In terms of structure, there are three principal types of documents: texts, free-form documents, and structured documents.

1. Texts consist of a string of information (in this case, characters) that appears two-dimensional but is really linearly ordered. More characters can be inserted anywhere within the text or added onto the end of the text. There is an inherent order to the characters in a text, and definite positions between characters.

2. Free-form documents start completely empty and unstructured, like a blank piece of paper. Information can be placed anywhere within the document; each piece of information has its own position. There may be large, empty spaces in the document that contain no information. There is no inherent ordering among the information in a free-form document. Pictures drawn in the graphics editor are free-form documents.

3. Structured documents have predefined cells to contain information. There is a fixed maximum number of cells per document; no cells can be added, nor can they be removed. Cells are usually arranged in rows and columns; a given cell is a member of one row and one column. There is a definite position between two adjacent cells,

and a position at the corner of a group of four cells. A spreadsheet is a structured document.

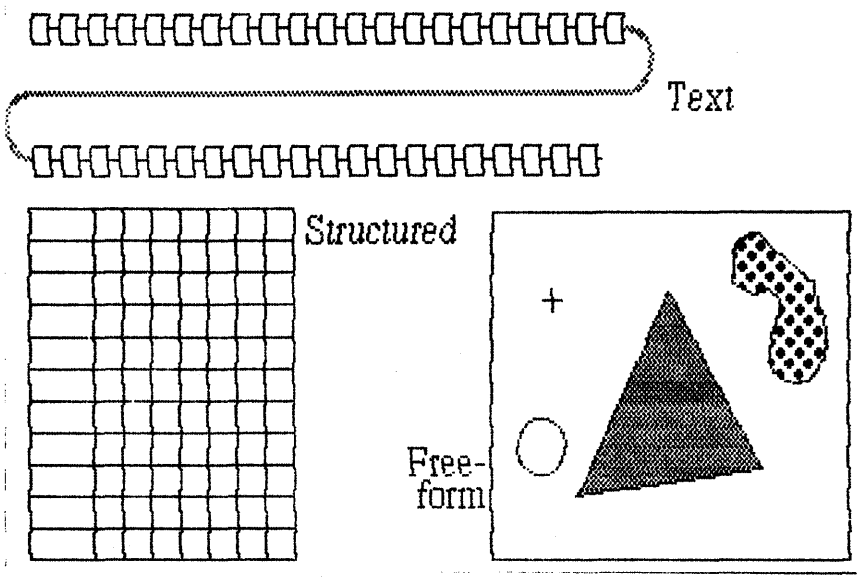


Figure 13. Types of Documents

The type of a document affects many things--mostly how a user selects information inside the document. For example, information in a text can be selected character-by-character, but information in a structured document is selected cell by cell. The exact details ot the selection process are described in the section "Selecting Information".

The Visual Structure

The structure can manifest itself visibly inside the document. For example, the rows-and-columns arrangement of a spreadsheet can be clarified by adding graphic grid lines between the cells. These lines are not part of the user's data, but they are part of the document. Such supporting graphics are usually static elements within the document, and cannot be moved or altered. Those that can be altered usually affect only the presentation of the user's data, not the data itself.

At the tool's discretion, the supporting graphics in a document may or may not appear when the document is printed. The grid lines on a spreadsheet might very well appear, while the rulers in a word processor document will probably not be printed.

Graphics in Documents

Not only does Macintosh use graphics to show the structure of a document and to otherwise communicate with its user, it also supports tools to create and manipulate graphic documents. Two such tools are planned: a graphics editor (to design and draw pictures, diagrams, illustrations, signs, etc.), and a charts and graphs package (to do bar

charts, pie charts, hi-lo graphs, etc. from a numerical data base).

Graphic documents are usually free-form: each graphic item in the document has its own position within the document, and there is no inherent relationship among the items (although the tool can define such a relationship). But there's no reason that graphic documents can't be structured. For example, a graphic programming language mught have a text-like or other structure.

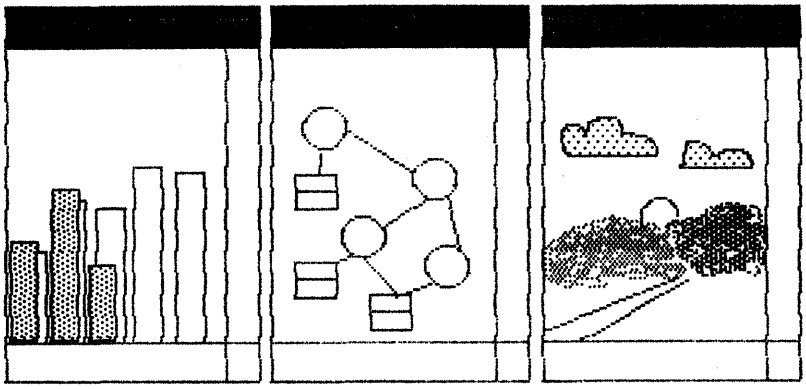


Figure 14. Graphic Documents

Graphics inside documents are produced using the QuickDraw graphics package. The package can draw seven fundamental graphic forms--lines, rectangles, ovals, rounded-corner rectangles, wedges, polygons, and arbitrary regions--either in outline or filled with a solid pattern. It can also place and manipulate images defined bit-by-bit. A tool can give the user the ability to draw anything from simple line drawings to finely textured halftone pictures.

The tool must itself determine how to respond to the mouse and keyboard in creating and manipulating graphics.

Appearance of Text

Most people, even bibliophobes, are confronted with a wide variety of printed matter on daily basis. Our eyes are so accustomed to seeing a myriad of typestyles, typesizes and typefaces used in publications to embellish or emphasize the content, that we no longer take special note. Developing eye-catching and pleasing typefaces has been an art unto itself since Gutenberg. Appropriate and aesthetically embued typesetting has been traditionally the domain of tooled craftsmen. By contrast, the repertoire of currently available computer 'typefaces' is thoroughly devoid of aesthetic nuances and provides but a bleak parody of the printed world.

Macintosh documents can contain characters in a number of different typefaces, typestyles, and typesizes. Type can abut closely or appear loosely packed; parts of some characters (such as the curl of a y) can reach back under or up over adjacent characters; and text can freely intermix with graphics. After all, text is just a specialized form of graphics.

Note that in this context, numbers are considered text: to users, the external appearance of digits is the same as that of other text characters. The following discussion thus pertains to numerical information as well as natural-language text.

- For more information on the aesthetics of type design, see a good typography book; David Gates' Type is recommended. For implementation details on how to place characters on the screen, see the Macintosh User Interface ToolBox manual QuickDraw: A Programmer's Guide.

Typefaces, Typesize, and Fonts

A typeface is a set of typographical characters composed with a coherent "feel" and consistent design. Things that relate characters in a typeface include the thickness of vertical and horizontal lines, the degree and position of curves and swirls, the use of serifs, etc. Typefaces have names, usually historical: Bodoni, Goudy, Tile, etc. The identity of a typeface is independent of its size or any particular typestyle it may conform to (see below).

Typesize in the printing world is measured in points, a point being reasonably close to 1/72 inch. The resolution (in points per inch) of the Macintosh screen is quite close to this, but not close enough to keep accurately to printers' measurements. But we do describe typesize loosely in "points", which have no correlation to the mathematical entity of a point in the QuickDraw graphics package, or to anything else for that matter. In talking about type, we use points as a rough indication of vertical size.

A font is the entire set of characters of a specific typeface and typesize. For example, Helvetica8 refers to a font that contains characters of the typeface named Helvetica at a size of 8 points. In addition to all the uppercase and lowercase letters, numerals and punctuation marks, a font may include mathematical symbols, accented letters or other special characters.

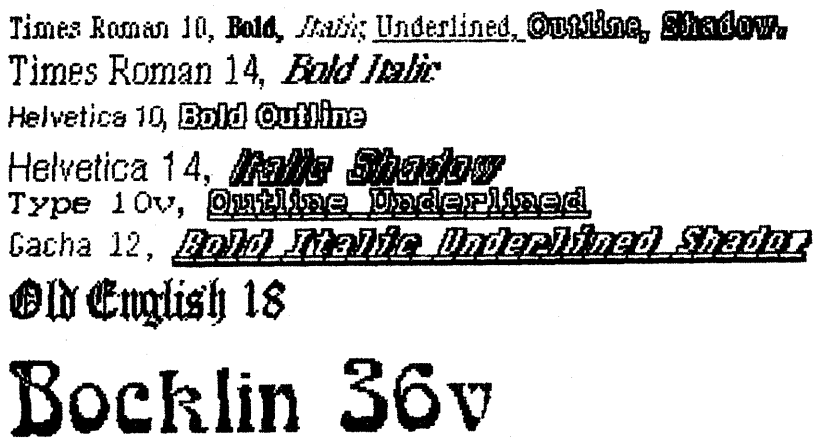


Figure 15. Type

Typestyles

Macintosh does not require the use of separate fonts to accommodate different styles of the same typeface. A character of any font may be subjected to a group of transformations that modify its general appearance: such a modification is called a typestyle. There are five fundamental typestyles: bold characters, italic (slanted) characters, outlined characters, underlined characters, and shadowed characters. Any combination of these typestyles can be used, but Macintosh cannot be held accountable for any aesthetic atrocities that may be perpetrated by an insensitive user.

Proportionally Spaced vs. Monospace Fonts

Most printing fonts are proportionally spaced (also known as variable pitch). This means that, for example, the "i" is narrower than an "m"; the "W" wider than the "J".

In a monospace (fixed pitch) font, all characters are of the same width. Monospace fonts are generally less attractive than proportionally spaced fonts. Monospace fonts are sometimes called "typewriter" fonts.

Monospace fonts are appropriate for some applications, such as COBOL coding forms, but generally discouraged in Macintosh. As monospaced fonts are merely a degenerate case of proportional fonts, they can be used just as easily as proportional fonts, when they are needed. It's necessary, for example, for proportional fonts to have monospaced numerals, so that columns of numbers line up neatly when aligned at decimal tab stops.

Standard Fonts

Macintosh uses a distinct system font when presenting its labels, messages, and lists to the user. System-provided text in this font cannot be edited. The Macintosh system font is Cream10; users and tools may not use this font.

There is always a standard font in which all information the user has entered will appear: the user font is Helvetica10, a nice, sans serif, reasonably compact face.

The use of any other fonts depends on the particular tool being used. The word processor, in all probability, will allow the user more multiple font ability than most other tools.

WORKING WITH MACINTOSH

So far, this document has described many things about the Macintosh user interface: how it accepts inpout from the user, how it displays information on its screen, and how the conceptual underpinnings of the system control the structure of interactions. But nothing has been said about how these things work together.

This section describes how input affects output: how Macintosh works. It discusses the methods the user will use to perform actions, select information, and choose commands to operate on that information.

Direct Manipulation: Controls

> "Piaget has hypothesized that infants first learn about causation by realizing that they can directly manipulate objects around them--pull off their blankets, throw their bottles, drop toys... Such direct manipulations, even on the part of infants, involve certain shared features that characterize the notion of direct causation that is so integral a part of our constant everyday functioning in our environment--as when we flip light switches, button our shirts, open doors, etc."
>
> -- Lakoff & Johnson, 198Ø

Friendly systems act on direct causation--they do what they're told. Performing actions on a system in an indirect fashion (by typing words and pressing RETURN, or by obediently choosing one item from the currently displayed list) reduces the sense of direct manipulation that is basic to the feeling of causation. To give Macintosh users the feeling that they are in control of their machines, many of a tool's features are implemented with controls: graphic objects that, when directly manipulated by the mouse, cause instant action with graphic results.

Three kinds of controls are supported by the Control Manager in the User Interface ToolBox: buttons, check-boxes, and dials.

Buttons

Buttons are small objects, usually inside a window (but occasionally on the desk top), labeled with words or an icon. Clicking or pressing a button performs the instantaneous or continuous action described by the button's label.

Buttons usually perform instantaneous actions, like opening or closing windows, or acknowledging error messages. Occasionally, they can also perform continuous action: the scroll arrows on a scroll bar are continuous-action buttons.

The Control Manager defines one kind of button, an instantaneous or continuous pushbutton, labeled with a verbal title. A tool may include a procedure to define a custom button, which can be linked in to the Control Manager and used just like the standard button.

Check-Boxes

Check-boxes are a variant of buttons. Where buttons perform instantaneous or continuous actions, check-boxes display a state that the user can change. Most commonly seen when filling out a form or setting parameters, check-boxes are small squares that appear either empty or filled in with a check-mark. The boxes are usually adjacent to a word or icon that describes the meaning of the box.

Clicking in a check-box flips its state, from checked to unchecked or vice-versa. Dragging through a field of check-boxes flips the state of the first and assigns the new state to all other boxes dragged through.

A check-box may belong to a group of boxes. If there are no interrelationships among the boxes, they are checked and unchecked as above. But if the boxes are related such that one and only one must be checked at any given time, they work like "radio buttons": clicking in an unmarked box marks that box and unmarks the previously marked box. Such groups should be labeled clearly, "Choose one of the following:". The checked appearance of this kind of box is visually distinct from normal, ungrouped check-boxes.

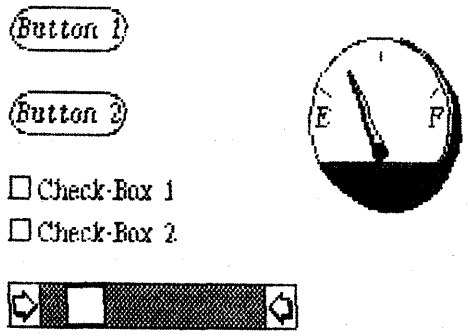


Figure 16. Buttons, Check-Boxes, and Dials

Dials

Dials display the value, magnitude, or position of something in the tool or system, and optionally allow the user to alter that value. Dials are predominantly analog devices, displaying their values graphically and allowing the user to change the value by dragging an indicator; dials may also have a digital display.

The best example of a dial is the shaft of a scroll bar. The indicator of the scroll bar is the thumb; it represents the position of the window over the length of the document. The user can drag the thumb to change that position.

Just as with buttons, there are a few standard dials defined in the ROM, but a programmer can implement a custom dial and link it in with the control mechanism.

Selecting Information

A previous section mentioned that Macintosh has one purpose only: to manipulate information. If this is true, then there is a simple operational paradigm to cover all situations:

(hand)
First select some information, then manipulate it.

This paradigm minimizes modality in basic operations. By selecting the information first, the user is free to select different information without being committed to a certain manipulation.

The following sections describe the two parts of this basic paradigm: how to select information in a document, and how to choose commands to manipulate that information.

The Selection

The selection is the collection of information that will be acted upon by the next command. There is always one and only one active selection in the active window. The selection can be so large as to enclose all the information in the document, or it can be so small as to merely indicate the position between two pieces of information, enclosing nothing at all; the latter selection is called an insertion point. The insertion point indicates the position at which newly entered information will be placed.

Positioning the pointer over the user's information in the active document and pressing the mouse button usually begins a selection. Once the button is pressed, the selection can be completed in two ways:

1. Clicking selects one piece of information or a position between pieces of information.

2. Dragging selects a group of information.

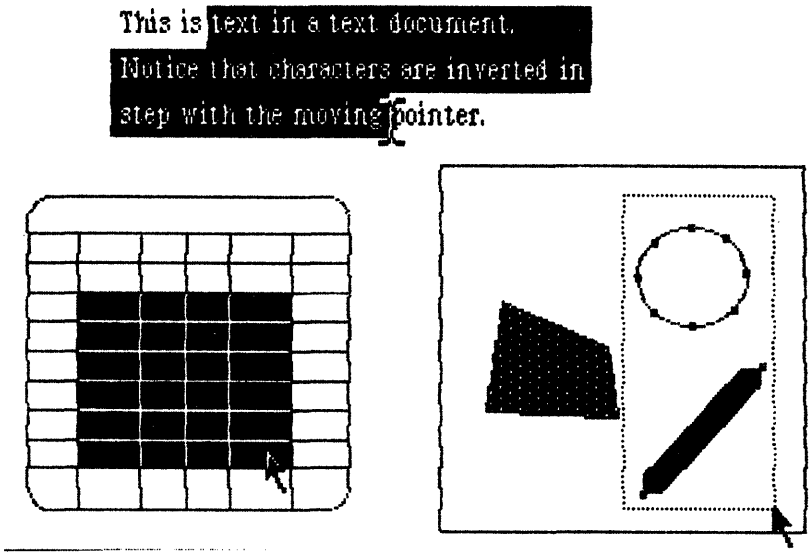


Figure 17. Selecting Information

The exact behavior of clicking and dragging to make the selection depends on the structure of the document.

- Clicking in text selects the position between the two characters nearest the pointer; this position becomes the insertion point. The insertion point in text is represented by a blinking vertical bar.

  Clicking in a structured document selects either the cell under the pointer, the position between two adjacent cells, or the corner of four cells. The latter two selections are insertion points, and are represented by blinking vertical or horizontal bars, or by a blinking cross.

  Clicking in a free-form document selects the item under the pointer. If the pointer is not over a piece of the user's information, clicking either does nothing, or selects a position in the document. This position, the insertion point, is marked by an "anchor" icon.

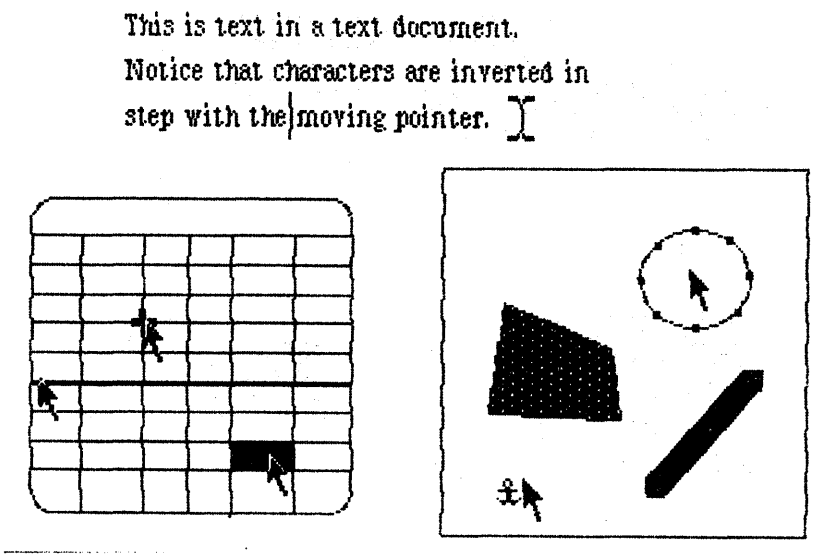


Figure 18. Selection by Clicking

Clicking in editable user information always creates a new selection; the information selected is highlighted and the previous selection is unhighlighted. Highlighted text appears white-on-black; highlighted graphics appear with "knobs".

Dragging through editable user information selects a group of information. It would seem that dragging should select all items dragged over--to select items, press the mouse button, drag across the items, then release--but experience proves that selecting only those items that were dragged over is inefficient. Instead, consider dragging as defining two points: the point where the button was pressed and the point where it was released. Dragging then selects everything between those two points, according to the structure of the document, regardless of the path of the mouse. The objects under the two points are included in the selection, as are all items between those two points.

- Dragging through text selects all characters, in textual order, from the character under the first point to the character under the last point.

  Dragging through a structured document selects all cells in the rectangle whose corners are the cell under the first point and the cell under the last point.

  Dragging through a free-form document selects all items completely enclosed by the rectangle whose corners are the first and last points.

During the dragging, the selection is visible--the items that will be selected are highlighted, in real time, according to the current

position of the pointer. But the selection is not actually confirmed until the mouse button is released. If the user moves the pointer back to the first point and releases the mouse button, the result is the same as a click at that position (see above)

The items between the two points are selected regardless of the relative orientation of the two points. Starting at the end of a sentence and dragging backwards to the beginning operates just as well as starting at the beginning and dragging to the end.

Once the selection is made, the selected items are highlighted and the items in the previous selection are unhighlighted. There is no mechanism for restoring the previous selection.

(hand)

After a selection is made, the pointer becomes invisible so as not to obscure the selection. The pointer reappears the next time the user moves the mouse.

Selection by Command

Some logical groupings of information are more commonly selected than others--columns or rows in a spreadsheet, paragraphs in a word processor, etc. And occasionally it's convenient for the tool to select a piece of information automatically--such as a word or phrase that the user is searching for.

In these cases, the invocation of a command may explicitly or implicitly make a new selection. For example, a tool may have a "Select All" command to select all information in the document; a spelling checker could have a "Select Next Misspelled Word" command, etc.

When any such command is invoked, the tool must scroll the document automatically in order to present as much as possible of the new selection.

Automatic Scrolling During Selection

The only limit on the size of the selection is the size of the document itself; the largest possible selection is the entire document.

But the normal method of selecting as outlined above can't handle selections that extend outside the window. We therefore define a way to scroll the contents of the window during selection:

- If during selection the user drags beyond the borders of the window, the contents of the window will scroll (automatically and continuously) away from that border. New information scrolled into the window becomes selected and is highlighted accordingly. Scrolling stops when the user either releases the mouse button or moves the pointer back into the window: the latter case resumes normal selection.

"Window" in the above paragraph applies to a single panel of a split window; beginning a selection in a panel and moving out of that panel scrolls only that panel.

Extending the Selection

Selection by dragging and automatic scrolling is fine for relatively small selections, but its usefulness deteriorates as the desired selection grows larger. An alternate method can be used to make a large selection: this process is called extending the selection. A selection made in this way is treated the same as any other selection.

Extending the selection merely adds to the current selection. Whereas making a normal selection removes the previous selection, making an extended selection enlarges the previous selection to extend to the newly selected position.

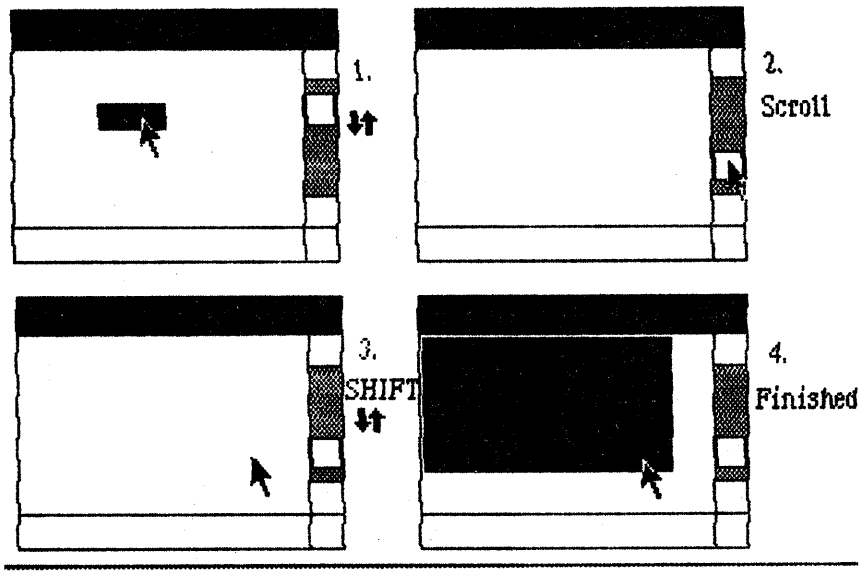


Figure 19. Extending a Selection

An extended selection is made by positioning the pointer, holding down either of the SHIFT keys on the keyboard, then pressing the mouse button. When the mouse button is pressed, all information between the original selection and the current pointer position (inclusive) becomes selected and highlighted. The user can then drag the mouse around and complete the selection as usual. The SHIFT key may be released at any time without affecting the selection.

Extended selections can be made across two panels of a split window.

Making a Discontiguous Selection

Some tools may choose to allow selections that are discontiguous: that comprise one or more unconnected pieces, that have "holes", or both. How a tool deals with operations on such selections is up to its designers; the following is merely an outline of how such selections are made.

(hand)

Discontiguous selection of text is not supported. It causes ambiguity upon insertion.

Making a discontiguous selection is like making an extended selection in that it merely augments the current selection, and also that it is invoked by holding down a keyboard key while pressing the mouse button.

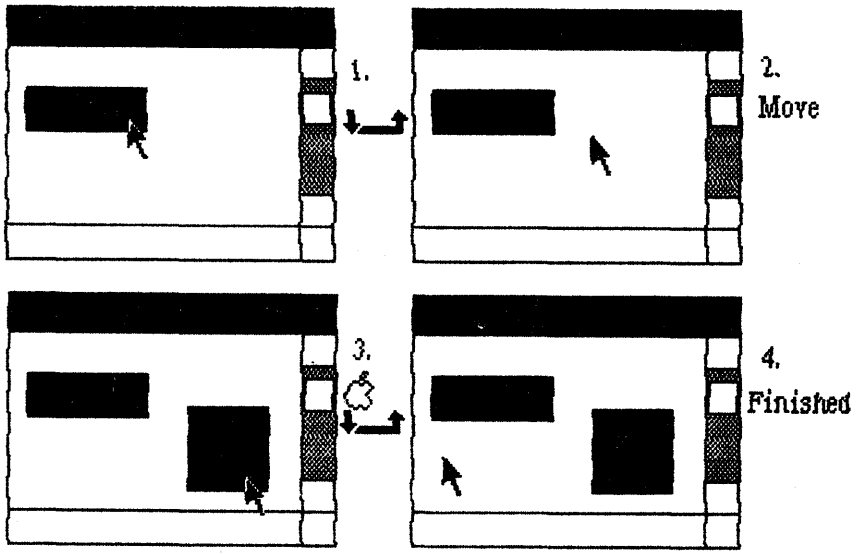


Figure 2Ø. Making a Discontiguous Selection

A discontiguous selection is made by positioning the pointer, holding down the COMMAND key, and pressing the mouse button. It continues like a normal selection: the user drags the mouse to indicate the last point, then releases the mouse button. The COMMAND key may be released at any time without affecting the selection. But the kind of selection that's being made depends upon the posiition of the pointer when the mouse button is pressed:

- If the pointer is not inside the previous selection, the operation is a normal selection that does not remove the previous selection. Both selected areas are highlighted on the screen; they are both considered parts of the selection.

- If the pointer is inside the previous selection, the operation becomes a deselection: the information "selected" becomes deselected and unhighlighted. The remaining information, even if

it contains a hole, is the selection.

With this paradigm, any arbitrary collection of items in the document may be selected. Once again, the selection comprises all highlighted items; there is one and only one selection.

Discontiguous selections can be made in any pane of a split window.

COMMANDS

Once the information to be operated on has been selected, a command to operate on that information can be chosen from lists of commands called menus.

A principal problem with menu-driven systems is that it's difficult for the menu to share the screen with the information being worked on, and especially difficult to show all menus at the same time. Most systems "solve" these problems with modal tree-structured hierarchies of menus, where menus are chosen from a menu of menus, while the user's information has disappeared from the screen. Unfriendly because it segregates information from commands, and confusing because it forces users to "walk" up and down trees of menus, this approach will not work for Macintosh. Instead, taking advantage of Macintosh's ability to overlap things on the screen, we make all menus available at all times (with the user's information still visible) by means of pull-down menus.

The Menu Bar

The menu bar is displayed at the top of the screen. It contains a number of words and phrases: these are the titles of the menus (see below) associated with the current tool. The contents of the menu bar and the corresponding menus are different for each tool. In this sense the tool is said to "own" the menu bar.

There is one and only one menu bar on the screen at any time. Exceptions may be made in special cases: full-screen games may need no menu bar, for example.

(hand)

> The titles in the menu bar, and their corresponding menus, should remain constant throughout the tool. A tool should not change the available menus or put up different menu bars at different times.

Of Mice and Menus

The user positions the pointer over a menu title on the menu bar and presses and holds the mouse button. The title becomes highlighted and a rectangular menu descends from the menu bar under the title; it remains down as long as the mouse button is held down, or until the user moves the pointer away from the menu.

The menu contains a number of items, usually stacked vertically inside the menu; each item names an operation that can be performed. The items may contain words, icons, or both. To invoke a command in the menu, the user drags the pointer down to the menu item (which becomes highlighted), then releases the mouse button. As soon as the mouse button is released, the menu item blinks briefly, the menu disappears, and the command is executed. The menu title in the menu bar remains highlighted until the command has completed execution.

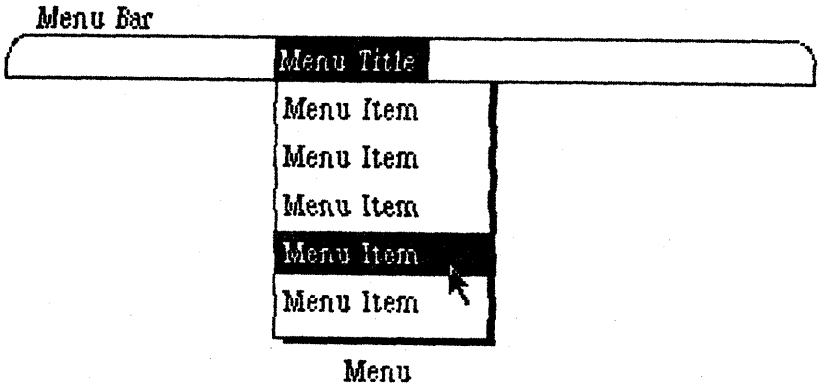


Figure 21. Pull-Down Menus

Because the user chooses a menu item only by pointing the pointer at it, and its command takes effect only when the mouse button is released, if the user drags the mouse outside the menu area (when the menus are showing) and releases the mouse button, no command is selected and no action takes place. Thus there is always recourse should the user have a change of heart after pulling down a menu, and the user is never forced to activate a command.

(hand)

The menu items, and NOT the menu titles in the menu bar, act upon selections. Users should always be able to peruse the inventory of commands by dragging the pointer across the menu bar without fear of causing something to happen.

The only way to pull down a menu is to press the mouse button while the pointer is in the menu bar. While the user is holding down the mouse button, the pointer does nothing but pull menus down and highlight their items.

If the user tries to perform an operation on a selection that is not currently visible, automatic scrolling occurs to make the selection visible before the operation is performed. The document scrolls until the selection is completely in view or, if the selection is very large, the entire window is filled with the part of the selection nearest to the current position; then the chosen operation is performed.

Notes on General Properties of Menus

Not all menu items are relevant at all times. A menu item that is inapplicable to the current selection is visually distinct from the others (perhaps grayed out) and will not highlight when a user tries to choose it. Repeated attempts by the user to choose an ineffective menu

item warrant explanations from the alert mechanism (see SPECIAL CONDITIONS).

(hand)
A menu in the menu bar can always be pulled down, even if all its menu items are ineffective; in such cases, the menu title is also grayed out. The user should always be able to survey all the available commands, even if they are inoperative.

Commands that may be invoked from the keyboard with the COMMAND key (see below) have a special notation on the right side of the menu. The notation consists at present of an apple symbol and the key that is used with COMMAND to invoke that command.

Menu items are grouped in a menu to emphasize the logical relationships among the groups. Groups are separated by a one-item-high blank space that serves to visually distinguish the groups. This space is not an item and is not highlighted when the pointer moves over it.

Experience shows us that it's easiest for users to choose the second, third, and fourth items in the menu: thy're far enough away from the menu bar to reach them without overshooting, but still not too much of a reach down. We recommend that the most common and safest commands go in these positions.

Also in regards to safety, the commands that cause the greatest effect (such as Quit) should be separated from other, less "dangerous" commands. Similarly, pairs of commands that perform similar functions with slight differences should not be adjacent; a user may choose one accidentally, intending the other, and not notice the subtle difference.

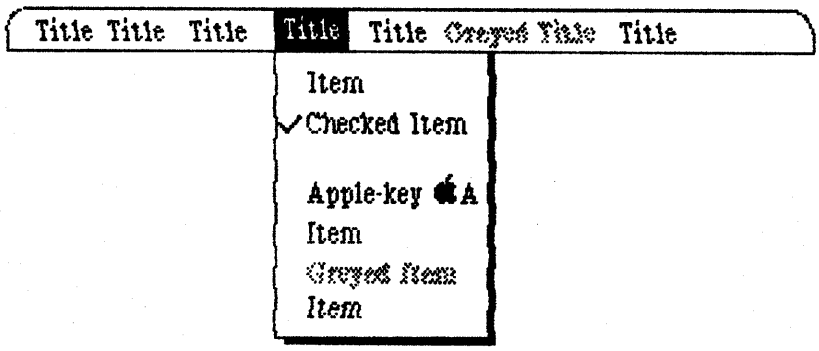


Figure 22. General Properties of Menus

Some commands come in pairs, with only one command of the pair being appropriate at any given time. Most often these pairs control the appearance of something on the desk: one command makes the object visible, and the other command makes it invisible. For example, in the Word Processor, the rulers that set margins and paragraph formatting are normally visible in the window. If the user wishes to remove the rulers, there is a command called "Hide Rulers". When the user invokes this command, the rulers disappear and remain hidden; meanwhile, that command has been replaced with its counterpart, "Show Rulers".

(hand)

These are not two different commands; they are opposite sides of the same command. The intent of this pairing method is to shorten and simplify menus. The pairing does not make a good indicator of state.

Some status information can be conveniently shown in menus, with the commands that affect that status. If all the information in the selection shares a certain characteristic, and that characteristic can be set with a menu command, that command is marked with a check-mark to show the state of the selection.

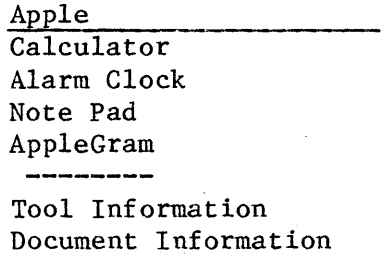
Also, in situations where commands in a menu not only perform their function on the selection, but also set a state that controls the interpretation of subsequent input (such as the Bold command), the commands whose states are currently in effect are similarly marked. In this way the menu allows the user not only to change how subsequent input will be interpreted, but also to see the interpretation before changing it.

The Standard Menus

Although the titles on the menu bar are different in each tool, the three menus at the left of the menu bar (the Apple, Edit, and File) remain the same at all times.

The commands and information in these menus pertain to functions common to all Macintosh users: inquiring the state of the current tool and data, invoking global system functions, and loading, saving, and printing documents.

The Apple Menu



Beginning the Apple menu are the names of the desk accessories currently available to the system. Choosing a name activates the corresponding accessory and places it on the desk; double-clicking the close box on the accessory makes it disappear and reactivates the previously active window. The list of available accessories changes with the availability of the accessories themselves.

The "Tool Information" and "Document Information" commands in the Apple menu let the user see information pertaining to the current state of the tool being used (its author, publisher, copyright message, version number, perhaps a hotline number) and the current document (its size, file name, label, creation and modification dates, "home" location or diskette, and any other status information).

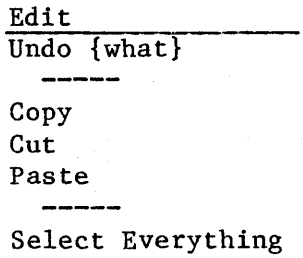
These commands, when invoked, present a window that contains the appropriate information; the window remains on the desk top until the user explicitly removes it by double-clicking its close box.

The document information window gives the user the ability to see important but little-used information about the current document, without taking up valuable screen space when the information isn't needed. The tool information is an important tool in the continued support of the customer: should anything go wrong with a tool, the users have a way to refer to the exact version number of the problematic program when seeking help from a dealer or hotline.

In tools that have a global "help" facility, the Help command appears at the bottom of the Apple menu.

The Edit Menu

The Edit menu includes all the editing commands necessary to manipulate pieces of documents.



The effects of the four editing commands are more thoroughly discussed in the BASIC EDITING PARADIGMS, below. Briefly, Cut removes the selection from the document, storing it in an intermediate window called the scrap; Paste replaces the current selection with the contents of the scrap; Copy duplicates the selection into the scrap without removing it from the document; and Undo negates the action of the immediately previous command.

Selection commands and other editing functions appropriate to the current tool may also appear in the Edit menu, but the location and order of the first four items must not change.

The File Menu

Although the exact functionality and layout of the File menu has yet to be worked out, our current thinking has it resembling this:

```
File
Quit this tool
  --------
Save this document
Print this document
  --------
Get another document
```

"Save this document" saves the current document into a file; "Get another document" gets a new document from another file; and "Print this document" invokes the printing subsystem of the tool.

"Save" and "Get" allow the user to use a limited subset of the Desk functions in selecting, creating, or naming the file associated with the document.

The "Quit" command is in the Files menu to make sure that users see their opportunity to save their work before quitting. Conversely, in the process of saving their work, they see their opportunity to leave the tool. If the user chooses to Quit before saving the document, the tool should give a gentle yet firm reminder that quitting now will cause the loss of all that information, and request confirmation before actually quitting.

Keyboard-Invoked Commands

The editing paradigms described below allow a user to perform all basic object manipulation--adding, removing, replacing, and moving--using the keyboard to enter text, the mouse to select text, and the commands in the Edit menu to manipulate it.

But this paradigm is likely to generate a lot of hand-waving--the user's hand must move from the keyboard to the mouse, and move the mouse from the document to the menu bar. As an optimization to reduce hand motion, common commands available on the three standard menus may also be invoked from the keyboard, by using the COMMAND key in combination with another key.

(hand)

> When the user holds down the COMMAND key on the keyboard and presses another key, that key is interpreted not as text entry, but as an invocation of a menu command. If the key does not correspond to any implemented command, the alert mechanism is invoked to beep at the first occurrence and give an alert message at any subsequent occurrences.

When one of these command keys is pressed, the menu title of the menu containing the corresponding command highlights while the operation is

being performed, then reverts to normal. The menu itself does not pull down.

The currently defined command keys are as follows:

COMMAND Z	Paste
COMMAND X	Cut
COMMAND C	Copy
COMMAND V	Undo
COMMAND space	Save this document and quit
COMMAND / or ?	Help

In all tools that have a Format or Typestyle menu to change the typestyle while entering text, the following command key aliases are supported:

COMMAND Q	Plain text
COMMAND W	Boldface
COMMAND E	Italic style
COMMAND R	Outline style
COMMAND T	Underlined
COMMAND Y	Shadowed

The commands, just like their counterparts in the menus, are cumulative: pressing COMMAND E while Boldface is already in effect results in bold italic text. The Plain Text command undoes all other styles.

The "OK" and "Cancel" buttons in dialog boxes (see below) also have command aliases:

COMMAND Enter	OK
COMMAND ` or	Cancel

Several emergency commands can be invoked from the keyboard. Note that rebooting the system is not among them.

COMMAND .	Stop current operation
COMMAND 1	Eject internal diskette
COMMAND 2	Eject external diskette

(hand)

The command keys are aranged positionally, not mnemonically. The command keys retain their position (not their alphabetical characters) on foreign keyboards.

What Commands Are and Aren't

- Commands, when invoked, operate immediately and return control to the user when completed.
- Commands operate on something visible in the active window, or add or remove a window on the desk.
- Commands that manipulate user information always operate upon the active selection, never upon any nonselected data.
- Commands are either verbs or verb phrases, never nouns with an implied verb.
- Most importantly, commands don't put the tool into an invisible modal state.

BASIC EDITING PARADIGMS

The Macintosh User Interface ToolBox contains a set of core editing routines that standardize the ways the user edits and manipulates text. As long as application programmers use this package properly, every piece of editable text the user sees on the Macintosh screen can be edited using the same quick, consistent methods. The paradigm below supports:

- Inserting, deleting, and replacing text;
- Moving text from one place to another in the same document;
- Carrying information between two similar or dissimilar documents.

The core editing routines also handle font changes, typestyles, and paragraph formatting; these abilities are further discussed in the documentation of those routines.

(hand)

The following discusses only the operation of Cut, Paste, Copy, Undo, insertion, and replacement on text. The same procedures should operate in a conceptually parallel manner on non-text items, i.e., graphics, spreadsheet cells, etc. It is the responsibility of the designers and programmers to maintain consistency in the editing operations on non-text items.

The Selection

As described in the section on "Inside Documents", there is always one and only one active selection in an active window that contains editable text. A selection takes one of two forms:

1. A selection between two characters that encloses no text: this appears as a blinking vertical bar and is called an insertion point.
2. A selection enclosing one or more characters of text.

The editing commands Cut, Paste, Copy, and Undo, whether invoked from the Edit menu or by the COMMAND key on the keyboard, act upon the selection. Typed characters also affect the selection.

The Scrap

The scrap goes hand in hand with the Edit menu. It is a very special kind of window with a well-defined function: it holds whatever is cut or copied from a document. It sticks around, its contents intact, when the user changes tools.

Every time the user performs a Cut or Copy on the current selection, a copy of the text in the selection replaces the previous contents of the

scrap.

The user can't select the scrap or any information inside it. But the scrap window can be dragged around by its title bar, and can be enlarged or reduced by dragging its size box. In most ways the scrap behaves just like any other window.

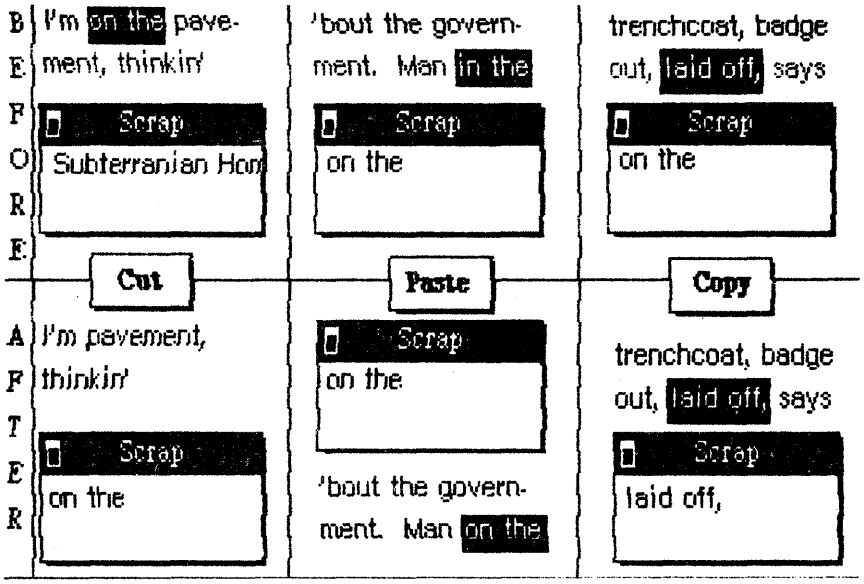


Figure 23. Use of the Scrap

There is only one scrap, which is on the desk for all tools that support Cut and Paste (it's hidden during games and such). If the user doesn't want the scrap to interfere with other things on the screen, the scrap can be shrunk to its smallest size, dragged nearly off the screen, or buried under other documents. Nothing changes the contents of the scrap except Cut, Copy, and Undo.

As the contents of the scrap remain unchanged when applications begin and terminate, the scrap can be used for transferring data among mutually compatible applications (see "Cutting and Pasting between Tools", below).

The Cut and Copy commands

The Cut command removes the current selection from the active document and puts it in the scrap. The selection completely replaces the previous contents of the scrap. The selection in the document is reduced to an insertion point.

If a Cut is attempted when the selection is an insertion point, Cut doesn't light up in the menu when chosen. This prevents people from accidentally cutting twice and losing the scrap.

Perfoming a Copy command puts a copy of the current selection into the scrap, without changing the original selection in the active document. Just as with a Cut, every Copy completely replaces the previous contents of the scrap. Also like Cut, the Copy item won't light up if

the selection is an insertion point.

Paste

The Paste command is the effective antonym of Cut: it replaces the current selection with the contents of the scrap. A Paste leaves the contents of the scrap unaltered; the selection is set to an insertion point at the end of the pasted text. With this, successive invocations of Paste replicate the contents of the scrap at the selected position in the document.

(eye)

> Notice that in a Paste over an existing selection, the contents of the selection do not go into the scrap; they can be recovered only by an immediate invocation of Undo.

Undo

Finally, the Undo command is a one-level negation of the last command. It always applies to all Edit commands; additionally, any larger scope of Undo can be added by the application. If the previous operation was an Undo, it undoes that Undo.

Inserting and Replacing Text

New text can be entered from the keyboard or numeric keypad. Typing new text operates much like a Paste command.

Typed text replaces the current selection. If the current selection is an insertion point, the typed characters appear at the insertion point and the insertion point moves past the characters. If the current selection includes text, the entire selection is automatically reduced to an insertion point, deleting the text; insertion then proceeds as described above.

(hand)

> Notice that if a selection is replaced with an entry from the keyboard, the selection does not go into scrap. Its contents can be recovered only through an immediate invocation of Undo.

Backspacing

Regardless of circumstances and context, if the selection is an insertion point, pressing the BACKSPACE key deletes one character before the insertion point and moves the insertion point to the left of the position previously held by that character. This happens during editing as well as text entry.

Pressing BACKSPACE while the selection contains characters operates much like a Cut, except that the deleted characters go into the backspace buffer (see below) rather than the scrap. The first BACKSPACE deletes the selected text, reducing the selection to an insertion point; subsequent presses of BACKSPACE operate as described

above.

Every press of BACKSPACE stores up the deleted characters in the backspace buffer. Invoking the Undo command reinserts all characters in this buffer back into the document at the insertion point. Performing any other operation, such as typing characters or invoking another command, clears this buffer; the deleted characters are then unrecoverable.

Cutting and Pasting Between Documents

Sometimes the user wants to transfer a portion of one document into another. The documents may have been created with the same tool, or with disparate tools. Macintosh allows this kind of manipulation through the mechanism of Cut/Copy and Paste.

Between Two Documents with the Same Principal Tool

Transferring information from one document to another created by the same application does not pose any difficulty. For example, the user may Copy the return address from 'Letter to Jef', Get 'Letter to Linda' and Paste in the contents of the scrap.

When the user discards a tool and returns to the Desk, the scrap retains not only its contents, but the contextual information pertinent to the tool being used. If the user retrieves that same tool, it can interpret that information, so there is little or no loss of context when carrying something in the scrap from document to document.

Between Documents with Different Principal Tools

Macintosh provides a limited but adequate scheme for transferring information from a document of one type to a document of another type.

Suppose the user wants to transfer a picture of a wolf (previously created using the Graphics Editor) into a Word Processor document named 'Letter to Grandma'. Beginning at the Desk, the user gets the wolf picture, automatically entering the Graphics Editor. There the picture is selected and Cut or Copied into the scrap; then the user returns to the Desk. The picture remains in the scrap.

Now the user calls up the letter to Grandma and enters the Word Processor. Upon selecting a position and attempting to Paste, the Word Processor examines the scrap and determines whether it is palatable. As Graphics Editor pictures are implemented with the QuickDraw picture structure, the Word Processor has no problem interpreting and displaying the picture, and graciously pastes it into the letter. However, in the letter the wolf and the rectangular area around it are selectable only as a single unit; the individual parts of the wolf are not editable. To the Word Processor the wolf is static data.

Each tool may have its own appropriate level of interpretation of the scrap. If the user tries to Paste the scrap in a tool that does not understand it, the tool presents an alert message to inform the user of

the undigestability of the scrap.

SPECIAL CONDITIONS

The <noun>+<verb> syntax is wonderful and clean when the operations are simple and act on only one object. But occasionally a command will require more than one object, or will need additional parameters in order to be most useful to the user. And sometimes a command won't be able to carry out its normal function, or will be be befuddled as to the user's real intent. For these special circumstances we have included two mechanisms: the Dialog Box to garner additional information, and the Alert mechanism to signal error or warning conditions.

Dialog Boxes

Commands in menus normally act upon only one or two objects: the current selection, the scrap, or a default object. If a command needs more information before it can be performed, it presents a Dialog Box to gather the additional information from the user.

A Dialog Box is a rectangle that may contain text, buttons, dials, and icons. It is slightly below the menu bar, a bit narrower than the screen, and as tall as its contents require. It is clearly labelled with the name of the command whose invocation prompted the appearance of the box.

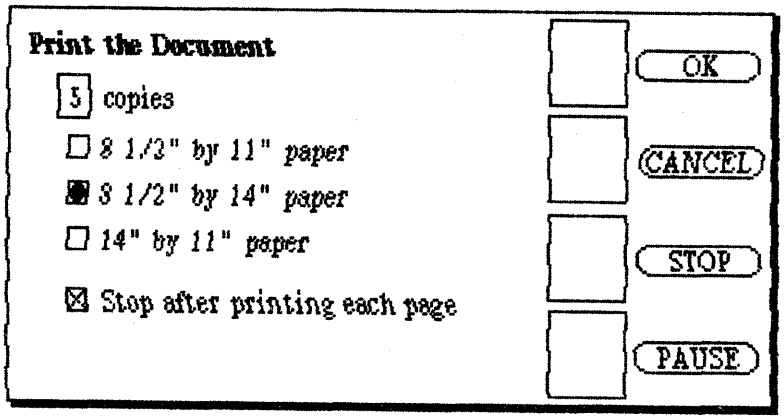


Figure 24. A Dialog Box

Some dialog boxes may affect several properties at the same time or show several choices of the same property. In such cases, the choices have check-boxes next to them. The boxes next to properties that are currently in force are checked. Clicking on a check box or the text accompanying it puts a check-mark in the box; this may also cause other boxes to become unchecked.

If the information requested by the dialog box is textual, the user can enter and edit that text just like any other editable text. If the information has a default value (which it should have, if possible), the default text appears selected in the dialog box. If the user starts typing, the selected value will be replaced with what the user types. For boxes with many text items, the first one is selected when the box appears. After editing an item,

- Pressing ENTER, TAB, or RETURN accepts the changes made to the item, and selects the next item in sequence.

- Clicking in another item accepts the changes made to the previous item and selects the newly clicked item.

There are, at the absolute minimum, two buttons in the Dialog Box--"OK" and "Cancel". "OK" enforces the modifications in the properties included in the Dialog Box, removes the Dialog Box from the screen, and performs the command originally issued. "Cancel" dismisses the Dialog Box without effecting any changes.

The "OK" and "Cancel" buttons should always appear in the same relative orientation in the Dialog Box to preserve a consistent feel to the interaction. They should be near the title of the dialog box to remind the user of what command they will perform or cancel. They may be marked with reinforcing icons, e.g., thumbs-up and thumbs-down.

A Dialog Box may include a "Stop" button, marked with an octagonal stop sign, for stopping operations that are in progress, such as printing.

When a command requires some time to execute, its Dialog Box may contain a dial that indicates the level of completion of the task in progress.

The Alert Mechanism

Every user of every application is liable to do something that the application won't understand. From simple typographical errors to slips of the mouse to trying to write on a protected diskette, users will constantly do things an application can't cope with in a normal manner. The Alert mechanism gives applications a way to respond to errors not only in a consistent manner, but in steps according to the severity of the error, the user's level of expertise, and the particular history of the error.

There are three levels of alerts:

1. Note: Probably a minor slip that's signaled by an audible warning.

2. Caution: A condition in which the application can't understand the user's input, and must request that the user change something.

3. Stop: A situation that requires definitive action on the part of the user, such as inserting another diskette.

These are ranked in ascending order of importance. Not only are program errors ranked in this manner, but repeating an error increases its importance: receiving the same Note alert several times, for example, turns it into a Caution, which warrants further explanation and assistance.

Note alerts are signaled by a beep from the speaker; if the speaker volume is turned off, the beep is inaudible. Caution and Stop alerts warrant an alert box (see below).

Alert Boxes

Alert Boxes are similar in appearance to Dialog Boxes. Alert Boxes are intended to give the user warnings and error messages. Before describing Alert Boxes it is worth while mentioning a few words about alert messages in general.

Alert Boxes are displayed to:

- Clarify the system's response to users' actions, (e.g., "This text is not editable"),
- Lead the user through a series of actions required for the completion of certain tasks, (e.g. "Please insert a diskette to be copied to"),
- Inform of a state that might affect users' future activities ("The document is getting too long to hold in memory. You may want to break it up into pieces"),
- Warn the user against doing something irrevocable or dangerous ("You will lose the contents of this diskette if you proceed with initialization. Do you still want to initialize?"), giving an opportunity to cancel the command, and
- Delay while a lengthy operation is being concluded.

How to Phrase an Alert Message

It is important to phrase messages in Alert Boxes so that users are not left guessing the real meaning. Do not use computer jargon. Sometimes it is difficult for the jaded to recognize jargon even as they use it. If you have any doubts of the lucidity of a message, try it on an unsuspecting naive friend.

Use icons whenever possible. Graphics can better describe some error situations than words, and familiar icons help users distinguish their alternatives better. The thumbs-up icon should always lead to the safest route out of a situation.

Generally, it is better to be polite than abrupt, even if it means lengthening the message. The role of the Alert Box is to be helpful and make constructive suggestions, not to give out orders. But its focus is to help the user solve the problem, not to give an interesting

but academic description of the problem itself.

Under no circumstances should an Alert message refer the user to external documentation for further clarification. It should provide a complete encapsulation of the information needed by the user to take appropriate action.

(hand)

> The best way to make an Alert message understandable is to think carefully through the error condition itself. Can the application handle this without an error? Is the error specific enough so that the user can fix the situation? What are the recommended solutions? Can the exact item causing the error be displayed in the alert message?

Be as specific as you can when signaling an error condition.

Appearance of Alert Boxes

An Alert Box is a rectangle just a little narrower than the screen and of variable height. It may contain text, icons, dials and buttons. It appears in a slightly lower position from where Dialog Boxes appear, to emphasize that the alert message is more important.

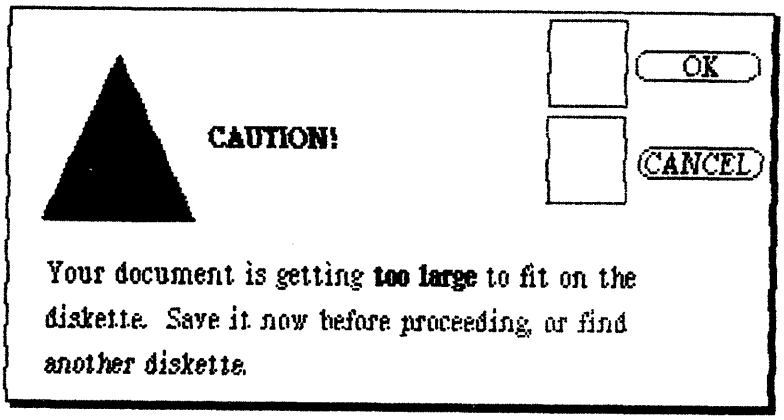


Figure 25. An Alert Box

All Alert Boxes have a "Cancel" button that dismisses the box. Alert Boxes that require confirmation to perform an action have an additional "OK" button. Some Alert Boxes may include a "Stop" button to allow the user to interrupt an ongoing operation. As in Dialog Boxes, the relative orientation of these buttons should remain the same from box to box.

If there are a small but finite number of ways to solve the problem, the box may contain descriptions of those ways, each marked by

check-boxes. The user checks the desired solution and presses the "OK" button.

Alert Boxes that require immediate attention contain a stop sign in the upper-left corner of the box to emphasize the severity of the warning.

Alert Boxes that inform the user about a process' status may display dials to indicate the level of completion of a task, much as in Dialog Boxes.

APPENDIX A: THOU-SHALT-NOTS OF A FRIENDLY USER INTERFACE

Here are six things to avoid when designing a friendly user interface.

1. Assigning more than one consequence to the same action.

2. Giving the user several ways to perform the same function. Generally, it is much easier for users to learn a task when there is only one obvious way of accomplishing it. Too many alternatives in an unfamiliar environment may paralyze the user.

3. Overloading an application with too many esoteric features. Before introducing another nifty feature, ask yourself how the feature will affect the overall complexity of the application, and how many users will benefit from the feature.

(hand)

Featurism is the single major contributor to system complexity and user intimidation.

4. Changing the state of the world while the user is not looking. One way to make a user comfortable with a system is to create an environment that is predictable and consistent. For example, if the contents of a menu change from one invocation to another, the user comes to think that the machine has a mind of its own, and feels that control of it will always be elusive.

5. Cluttering the screen. A cluttered and busy screen is frequently a symptom of an application design that is not carefully thought out. Reevaluating the reasons for different features (always keeping the end user in mind) will generally result in a simpler, more elegant program and visually more streamlined interface.

6. Overenthusiastic use of modes. It is highly desirable, if not always possible, to allow the user to go from one activity to another without feeling trapped in a mode. For an eloquent discussion of modes, the reader is referred to "The Smalltalk Environment", an article by Larry Tesler in the August, 1981 issue of BYTE magazine.

APPENDIX B: POINTER SHAPES

Certain pointer shapes have been standardized to imply that specific actions will occur when the mouse button is pushed.

(I-beam)
Text selection 

(Hollow Cross)
Selection in a structured document 

(Plus sign)
Drawing graphics 

(Hourglass)
Long operation in progress (sometimes associated with a dial in a dialog box) 

(Arrow)
All remaining cases, including menus, desk top, graphics selection, button-pushing and dial-dragging, dead data, etc. 

APPENDIX C: THE PHYSICAL BOX

The following summarizes Macintosh's salient hardware features.

Physical box:

- A main unit with a built-in 9" CRT and a built-in minifloppy drive;
- A detached keyboard;
- A mouse.

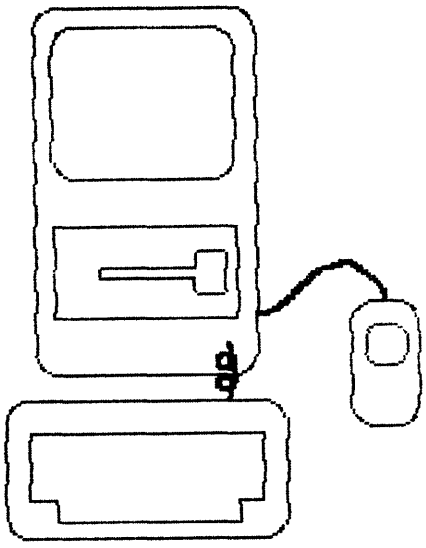


Figure 26. Macintosh

Memory capacity:

- 131,Ø72 bytes (128K) of user and program memory, 21,888 bytes (21 3/8 K) of which are dedicated to the video display;
- 65,536 bytes (64K) permament (ROM) storage;
- 86Ø,16Ø bytes (84ØK) storage on the built-in disk drive.

Microprocessor:

- Sixteen-bit Motorola MC68ØØØ with eight 32-bit data registers, seven 32-bit address registers, and two stack pointers.
- 56 instructions in 14 addressing modes; microprocessor runs at 8 million cycles per second (8MHz).

Display:

- 512 dots wide, 342 dots tall, black and white dots on a square grid. Dots displayed at 8Ø dots per inch on a 9" screen.

This is the only configuration of Macintosh. There are no other memory sizes, no different ROMs, no other video displays. The consistency of the Macintosh user interface is based on the consistency of the hardware: as every Macintosh ever sold is guaranteed to contain the above, every application program written for this configuration will run on 1ØØ% of the installed base.

The only options available are:

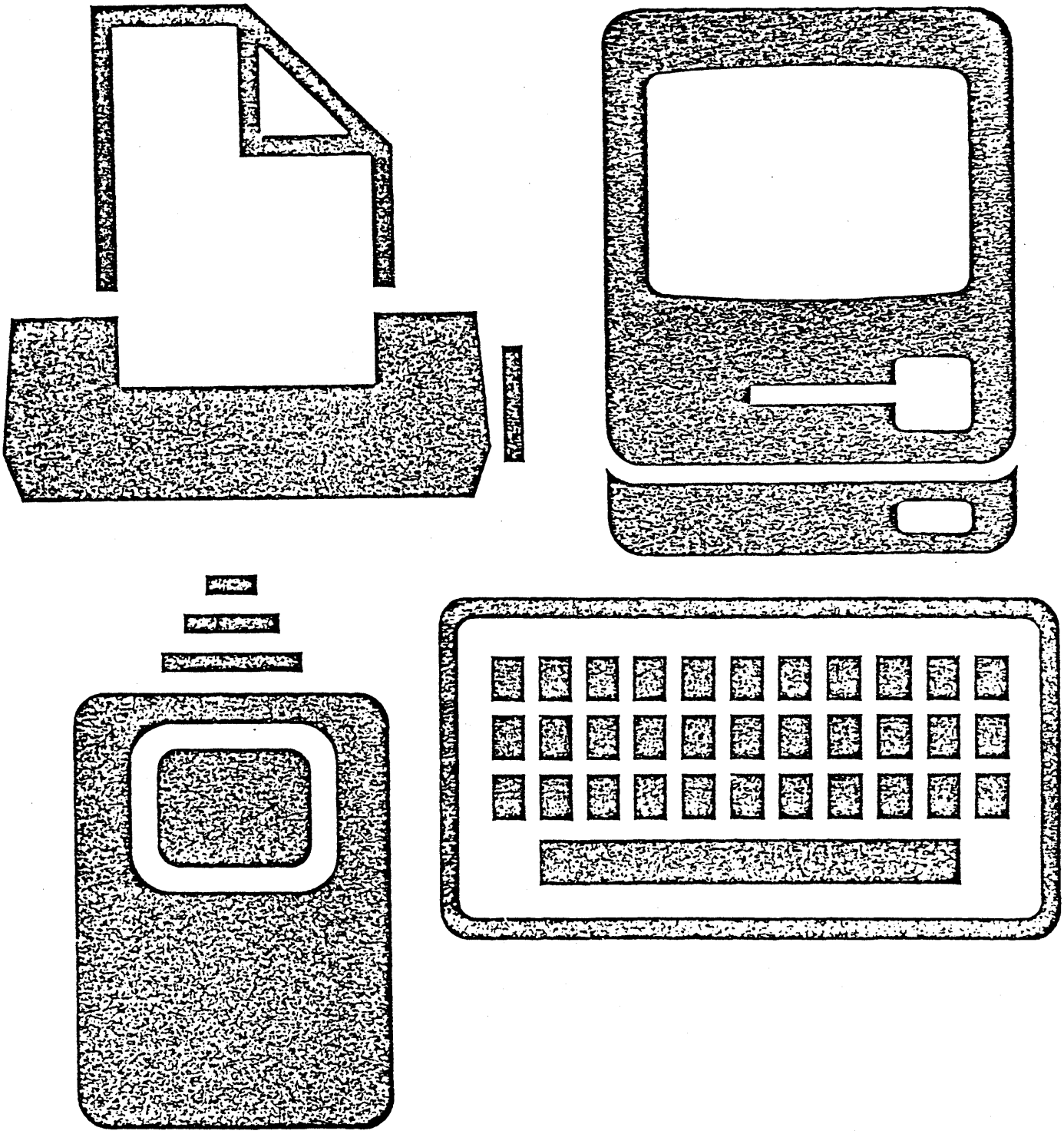
- A second 84ØK floppy disk drive;
- An 18-key numeric keypad;
- A dot-matrix or letter-quality printer;
- Connection to a RS-232, RS-422, or network communication device.

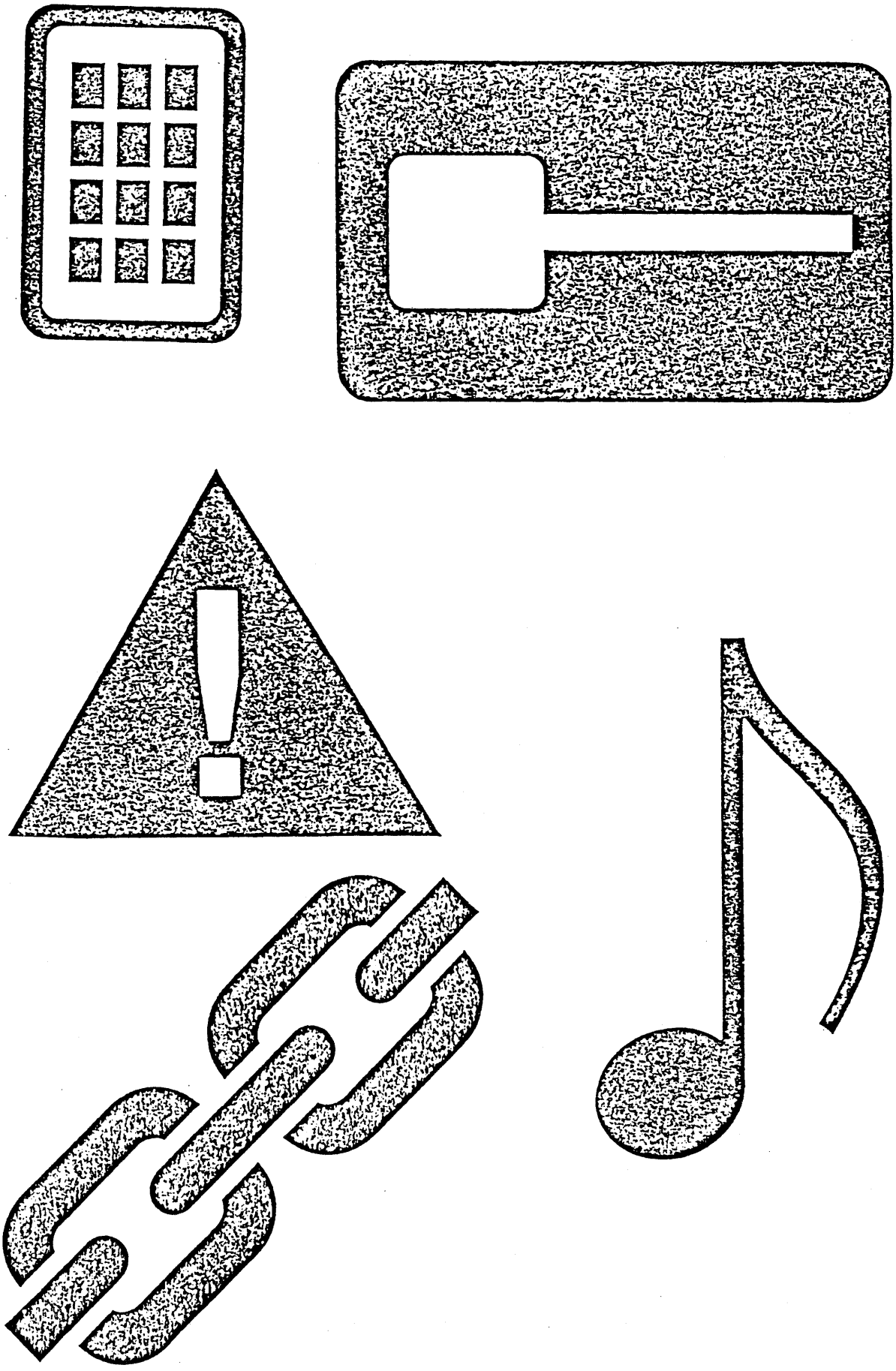
APPENDIX D: KEYBOARD LAYOUTS AND CHARACTER ASSIGGNMENTS

Here are the keyboard layouts and ASCII character assignments for the standard character sets in Macintosh:

APPENDIX E: GUIDE TO ICONS

Here are the standard icons as used on our packing materials, on the back of the Macintosh itself, and appearing in Macintosh software:





APPENDIX F: UNRESOLVED ISSUES

- What does the Close box do in the main document window for a tool? Does it put away the document, unload the tool, and return the user to the Desk? As Larry's tests show that users occasionally hit the Close box when intending to drag the title bar (or pull down a menu), is it proper for such a commonly-misused icon to perform such a time-consuming task?

- When inactive windows in Lisa are dragged, they are brought to the top afterward. We don't do this.

- Do Show Scrap/Hide Scrap exist? Where? And is the scrap called the Clipping?

- How do Macintosh command-key assignments differ from those on Lisa, and will we have a real Apple key rather than the word COMMAND?

- Do Randy's Core Editing or Word Processor routines support backspace-by-word, or unbackspace?

- There's a clash between the use of the stop sign as a warning icon in Dialog and Alert Boxes and its use as an icon on the interrupt button in the same place.

- COMMAND-Click and SHIFT-click, and their conflict in the Graphics Editor, is unresolved.

- The 1/4"-grey-around-the-edges was dropped in this draft. It is superfluous, hard to code, and adds little to the illusion.

TECHNICAL LEXICON

These terms are defined here in their technical meaning and relationship to one another. Users will never encounter some of the terms mentioned here; neither will they read the descriptions as phrased here. For a users'-eye-view of Macintosh terminology, please see the glossaries in the Macintosh User Style Guide and in the Macintosh Introduction manual.

Active Selection (Noun) See Selection, Active

Active Window (Noun) See Window, Active

Alarm Clock (Noun) A desk accessory that displays the current date and time, as well as allowing the user to set an alarm date and time and an alarm message.
Usage: Same as Desk Accessory

Alert Box (Noun) A window containing warnings and cautions, which appears when a tool encounters an unsolvable error or a dangerous situation. An alert box always contains two buttons, labeled OK and Cancel.
See Also: Alert Message
Usage: Present an A.B.

Alert Message (Noun) An audible or visible message or warning generated by the computer to signal input errors, problems interpreting data, or situations threatening the safety of the user's data.
See Also: Alert box

Automatic Scrolling (Noun) See Scrolling, Automatic

Back (Noun) The position or orientation of objects on the desk furthest from and least visible to the user; objects in front overlap and obscure objects in the back.
See Also: Front Window Behind
Usage: Send to the b. In b. of another

Behind (Adverb) In the position or orientation towards the back. An object on the desk is behind all the objects that are in front of it.

Button (Noun) A control that causes an action when clicked or pressed. Buttons highlight when pressed.
Usage: Press Click

Button, Mouse (Noun) See Mouse Button.

Calculator (Noun) A desk accessory that emulates a four-function desk calculator. Calculation results can be cut and pasted between the calculator and the user's document.
Usage: Same as Desk Accessory

Cancel button (Noun) A button that, when pressed, cancels a proposed action or action in progress. The cancel button is labeled "Cancel" and is marked with a thumbs-down icon.
See Also: OK button
Usage: Same as button

Check Box (Noun) A control in the shape of a square box, which may or may not have a check mark in it. Clicking in a check box toggles its state, and may affect the state of related check boxes.
Usage: Check Click

Choose (Verb) To pick a menu item from a menu.
Usage: Choose a command Choose a menu item

Click (Verb) To position the pointer and briefly press and release the mouse button without moving the mouse.
See Also: Drag Double-Click
Usage: Click an object Click the mouse button

Close (Verb) To remove the window from a document; you close a window to reduce it to an icon that represents the document.
Usage: Close a window (never close a file)

Close Box (Noun) The box on the left side of the title bar of a document window that, when clicked, closes the window. The close box contains an icon of a document that "winks" when ckicked.
Usage: Click the close box

Closed (Adjective) The state of a window when the document it contains is not visible. Documents whose windows have been closed are represented by icons.

Command (Noun) A word (usually appearing as a menu item) that describes an action that a Macintosh tool can perform; or the action itself.
Usage: Choose a command from a menu The command takes effect

Control	(Noun) An object on the screen that causes an action when clicked or dragged; buttons, dials, and scroll bars are the most common controls. Usage: Use only when necessary.
Control Panel	(Noun) A desk accessory full of controls. With it, the user can change the speaker volume, the keyboard repeat speed and delay, system paranoia level, etc. Usage: Same as Desk Accessory
Desk	(Noun) The tool that deals with coying, moving, creating, deleting, and changing the names of files. Also refers to the smaller version used within applications. Usage: On the desk (?)
Desk Accessories	(Noun) Mini-tools generally available at all times. A pocket calculator, note pad, telegram form, alarm clock, and the control panel are the currently imagined desk accessories. Usage: Get a D.A. Use the D.A.
Desktop	(Noun) The metaphor for the Macintosh working environment. See Also: Desk
Dial	(Noun) A control that acts as a pseudo-analog output and/or input device. See Also: Scroll Bar Usage: Adjust a dial
Dialog Box	(Noun) A window opened by a tool that requests the user for entry or confirmation of information. A dialog box is presented when a chosen command needs more information in order to take effect. See Also: Alert Box Usage: Present a d.b. Close the d.b.
Discontiguous Selection	(Noun) See Selection, Discontiguous
Disk	(Noun) Any kind of rotating magnetic storage device. See Also: Diskette Disk Drive Usage: Save on a d. Get from a d.
Disk Drive	(Noun) The mechanism that stores and retrieves the information on a disk. See Also: Diskette

Diskette
(Noun) A thin, plastic disk.
See Also: Disk Drive
Usage: Insert the d. Eject the d. On the d.

Document
(Noun) A collection of information intelligible to a user.
See Also: File Window Tool
Usage: Get a d. Save a d. Scroll a d.

Document Panel
(Noun) The pane of a document window that presents the document itself, as opposed to status panes, formula panes, etc.
See Also: Panel
Usage: Avoid if possible.

Document Window
(Noun) A window that displays a document. Document windows usually come equipped with a title bar, one or two scroll bars, a size box, and a close box.
Usage: Use only when "window" is ambiguous.

Double-Click
(Verb) To click the mouse button again shortly after a previous click. Double-clicking an object enhances or expands the action normally caused by singly clicking that object.
Usage: D.C. an object D.C. the mouse button

Drag
(Verb) To press and hold the mouse button while moving the mouse. Dragging either selects items (when done inside the window) or drags a flickering outline of an object (outside the window).
See Also: Click Select Choose Size Window Split a Window
Usage: D. an object D. the mouse D. out a rectangle D. across the text

Enter
(Verb) To insert or add information into the computer, usually by typing on the keyboard. Entries are usually terminated by a press of the ENTER key.
Usage: E. the name

Extend (the Selection)
(Verb) To make the active selection larger by holding down the COMMAND key while making another selection. The two selections and all items in between become the new selection.
See Also: Select Selection
Usage: Extend the Selection Make an extended selection

File
(Noun) A storage container for information.
See Also: Document Tool Window Resource File

Usage: Delete a f. Copy a f. Move a f. Rename a f.

File — (Verb) To put a document into a file, or get a document from a file.

File Name — (Noun) The name attached to a file by its creator.

Font — (Noun) A set of characters of the same typeface and size.
See Also: Typestyle
Usage: Appears in the f.

Front — (Noun) The position or orientation of objects on the desk that are closest and most visible to the user; the active window is always in front of any other windows.
See Also: Back Behind
Usage: Bring to the f. In f. of others Frontmost

Highlight — (Verb) To emphasize something by making it visually distinct from its normal appearance; by inverting it, underlining it, making it blink, or appear in boldface, etc.
See Also: Invert Select Front Window
Usage: H. the text Title bar is highlighted

Icon — (Noun) "1. An image; representation. 2. A similie or symbol." (AHD) A graphic representation of a material object, a concept, or a message. Icons may be objects on the desk.
Usage: Click an i. Drag an i. Labeled with an i.

Inactive Selection — (Noun) See Selection, Inactive

Inactive Window — (Noun) See Window, Inactive

Insertion Point — (Noun) A selection enclosing nothing; indicates the position between two items in a document, or an absolute position in that document. Indicates the point at which newly inserted items will be placed.
See Also: Select
Usage: Make an I.P. At the I.P.

Invert — (Verb) To invert the black-and-white polarity of an image; inverting is the most common form of highlighting.
Usage: Inversely highlighted

Item

(Noun) A single piece of information in a document. Each character in a text, each shape or line in a picture, and each cell in a spreadsheet is an item.
See Also: Select Drag Extend (the Selection)
Usage: Between two items Click an i. Drag over items

Key

(Noun) A button on the keyboard. Character keys are typed; modifier keys are held; special keys are pressed.
Usage: Press a k. Hold down a k.

Keyboard

(Noun) The device used for entering text and numeric data. The keyboard has 48 character keys, 6 modifier keys, and 4 special keys.
See Also: Press Type Hold
Usage: Type on the k.

Menu

(Noun) A rectangular list of menu items, which is pulled down from the menu bar; the user chooses a menu item by pressing on a menu title, dragging through the menu, and releasing on a menu item.
See Also: Command
Usage: Choose from a m. Pull down a m.

Menu Bar

(Noun) The horizontal strip at the top of the screen that contains the menu titles.

Menu Item

(Noun) One item in a menu. A menu item may contain words, an icon, or both. Menu items usually describe commands. A menu item is highlighted when the pointer is over it.
See Also: Choose
Usage: Choose a m.i.

Menu Title

(Noun) A word or phrase in the menu bar that designates one menu. Pressing on the menu title pulls down ite menu; dragging through the menu highlights menu items.
Usage: Press on the m.t.

Mouse

(Noun) A small device the size of a deck of cards that rolls around on your desk. Moving the mouse causes corresponding motion of the pointer on the screen.
See Also: Mouse button Drag
Usage: Move the m. Drag the m.

Mouse Button

(Noun) A rectangular button on the top of the mouse. Pressing the button initiates some action at the position of the pointer;

releasing the button confirms the action.
See Also: Click Double-click Drag
Usage: Press the m.b. Release the m.b.
Click the m.b.

Note Pad

(Noun) A desk accessory that works as a mini word processor, allowing the user to enter and edit small amounts of text while working on another document.
Usage: Same as Desk Accessory

Numeric Keypad

(Noun) An auxilliary keyboard containing keys for digits and arithmetic operators, used for numeric input. The numeric keypad contains sixteen character keys and two special keys.
See Also: Press Type
Usage: Same as Keyboard

Object

(Noun) Anything distinguishable image on the desk. Windows, the menu bar, and icons are objects.
See Also: Drag Click Front
Usage: Click an o. Drag an o. Select an o.

OK button

(Noun) A button that, when pressed, confirms a proposed action. The OK button is labeled "OK" and is marked with a thumbs-up icon.
See Also: Cancel button
Usage: Same as button

Open (a Window)

(Verb) To create a window onto a document in order to view the information.
See Also: Close Closed
Usage: Open a window

Pane

(Noun) A portion of a window with a different function or purpose than other panes of the same window. The tool defines the panes in the window it presents.
See Also: Panel
Usage: One pane of the window

Panel

(Noun) A user-definable subdivision of a pane. The user creates panels in the document pane by using a split bar.
See Also: Splitting a Window
Usage: One panel of the window

Pointer

(Noun) A small object, usually a north-northwest arrow, that hovers above all other objects on the screen. It moves around as you move the mouse around.
See Also: Click Drag Press
Usage: Position the p. by moving the mouse

Press
(Verb) 1. To depress the mouse button. 2. To depress a special key on the keyboard. 3. To position the pointer with the mouse and depress and hold the mouse button.
See Also: Click Drag Key Release
Usage: P. the RETURN key P. the mouse button P. on a menu title

Principal Tool
(Noun) See Tool, Principal.

Release
(Verb) To cease pressing. Releasing the mouse button quickly, without moving the mouse, results in a click.
See Also: Drag Double-Click
Usage: R. the mouse button

Resource File
(Noun) A file containing information relevant to or necessary for the operation of the Macintosh or an individual tool.
Usage: Same as file

Scroll
(Verb) To move a document so that a different part of it is visible in the window.
See Also: Scroll Bar Scrolling
Usage: S. the document

Scroll Arrow
(Noun) A button at either end of a scroll bar, with a picture of an arrow on it. Pressing a scroll arrow scrolls the document in its direction, and moves the thumb closer to the arrow.
See Also: Scroll
Usage: Same as button

Scroll Bar
(Noun) A rectangular bar along the right or bottom edge of a window. Clicking and dragging in various parts of the scroll bar moves the document in the window, and moves the thumb accordingly.
See Also: Scroll Arrow Shaft Thumb
Usage: Use the s.b.

Scrolling, Automatic
(Noun) The process of scrolling while making a selection.
See Also: Scroll

Select
(Verb) To click or drag in a collection of items or objects, in order to designate them to be acted upon by a subsequent command.
See Also: Selection Insertion Point Extend the Selection
Usage: S. the text S. a file

Selection

(Noun) A collection of text or objects designated to be acted upon by a subsequent command. The selection appears highlighted in the document.
See Also: Select Highlight Active/Inactive Selection Insertion Point
Usage: Edit the s. Make a s.

Selection, Active

(Noun) The selection that will be influenced by the next command. The active selection is always highlighted.
See Also: Selection, Inactive
Usage: Use "selection" unless ambiguous.

Selection, Discontiguous

(Noun) A selection whose items are not contiguous: that have other, nonselected items between two selected items. A discontiguous selection is made with the assistance of the SHIFT key.
See Also: Selection Extending the Selection
Usage: Make a d.s.

Selection, Inactive

(Noun) A selection in an inactive window, or in a pane of the active window other than the pane containing the active selection.
See Also: Selection, Active
Usage: Refer to this only when necessary.

Shaft

(Noun) The long, thin gray area of the scroll bar in which the thumb appears. Clicking in the shaft to either side of the thumb moves the document one page.
See Also: Thumb
Usage: Click in the s.

Size

(Verb) To change the size of a window by dragging its size box.
Usage: Size a window

Size box

(Noun) A rectangle in the bottom right corner of a window containing an icon. Dragging this box allows the user to alter the size of the window by repositioning its bottom right corner.
See Also: Size
Usage: Drag the s.b.

Split a Window

(Verb) To drag a split bar in order to divide a window into two panels.
Usage: Split the Window

Split bar

(Noun) A small black bar at one end of a scroll bar. Dragging a split bar into the window causes the window to be split into two

panels at the point where the mouse button was released.
See Also: Split a Window
Usage: Same as object

Telegram Form (Noun) A desk accessory that allows the user to send or receive messages over the AppleNet network.
Usage: Same as Desk Accessory

Thumb (Noun) The indicator of a scroll bar. The position of the thumb within the shaft represents the position of the window over the length (or breadth) of the document.
See Also: Thumb (verb)
Usage: Drag the t.

Thumb (Verb) To move to a different part of the document by dragging the thumb, clicking or pressing the scroll arrows, or clicking or pressing in the shaft.
See Also: Scroll
Usage: T. through the document

Title Bar (Noun) The horizontal bar at the top of a document window. It contains the name of the file from which the document in that window was taken. On the left side of the title bar is the close box.
Usage: Same as object

Tool (Noun) A manipulator of information, otherwise known as an Application Program.
See Also: Document File Resource File Tool, Principal
Usage: Get a t. Use a t.

Tool, Principal (Noun) The tool most strongly associated with a given document.

Type (Verb) To press and release one or more character keys on the keyboard or numeric keypad. The user types information on the keyboard.
Usage: Type the following Type on the keybaord

Typeface (Noun) A collection of letters, digits, punctuation marks, and other typographical symbols with a coherent "look" and consistent design.
See Also: Typestyle Font
Usage: Of a t.

Typestyle	(Noun) A stylistic variation that can be applied to any typeface; examples are boldface, italic, underlined, shadowed, and outlined. See Also: Font Highlight Usage: In a t.
Window	(Noun) A presenter of information. A window is an object on the desk. See Also: Document Window Tool Usage: Open a w. Close a w. Size a w. Drag a w. Bring the w. to the front
Window, Active	(Noun) The frontmost window, which will receive commands and data entered. See Also: Window, Inactive Usage: Use "window" unless ambiguous.
Window, Inactive	(Noun) Any visible window other than the active window. See Also: Window Usage: Refer to this only when necessary.

MSG#:B41194
 IN#: 111
  TO: MAC
FROM: SUPT   MAC
SENT: 30 NOV 83 16:17:25
READ: 05 DEC 83 09:59:19

FILE MENU AND FILING COMMANDS

The File menus should read:

Application	Finder
New	Open
Open...	Duplicate
Close	Get Info
Save	Put Back
Save As...	
Revert to Saved	Close
Page Setup...	Close All
Print...	Print
‹ your items here›	
Quit	Eject

New... in a one-document application, is disabled while a document is open. When chosen, opens a new document window with the title "Untitled". Get the word "Untitled" from the System Resource file.

Open... brings up the GetFile dialog showing the contents of the disk (default to first on-line volume in the volume queue, usually the boot volume). User can use the Drive and Eject buttons to switch disks or drives. The disk directory shows only documents that the current application can open (the application passes a type mask, or can install itself in a filterProc to select which names to display). The user can select one and only one document from the list. Pressing "Open" attempts to load that document (the GetFile dialog will check to see that it's there, readable, the right kind, etc.) If your application has trouble loading the selected file, it should alert the user and cancel the command. It should not automatically return to the GetFile dialog.

Place the name of the opened file in the title bar of the document window. Record the volume number and version byte to be used in saving the document.

Open: is disabled if no more documents can be opened at the moment. The user must manually close an (or the) open document before opening a new one.

Save: attempts to save the current document with the same name, volume number, and version number as given when it was opened. If the document name is "Untitled", Save drops into Save As.

Save As... calls the PutFile dialog, prompting "Save document as:" with the current name as the default (unless the current name is "Untitled", in which case the default is a null string). The default volume is the first on-line volume in the volume queue (usually the boot volume), not the volume the file came from (because it may be off-line). When the user clicks Save, PutFile verifies that the file is writable, and does overwrite warnings. If you get an error while writing, or the disk is full, etc., loop back to the PutFile dialog until a save is successful, or cancelled.

Once the file is written, change the document's name in the title bar to that of the file written. Remember the volume number and version byte for the next Save command.

Close: If the frontmost window is a desk accessory, a modeless dialog, or an accessory window, Close closes that window. If the frontmost window is a document window, Close does an implicit Save before closing the window. Most applications should be able to function without an open document window (so the user can open another document, use desk accessories, etc. without leaving the application); those that can't should either force an Open or Quit after closing the document window.

Page Setup: brings up the first printing dialog, with document-specific information that should be saved on disk with the documents. This includes page size and orientation, and any other information desired by the application.

Print... brings up the printing dialog for the configured printer. Settings stick to the printer. See Owen for details of implementation.

Revert to Saved: confirms that the user really wants to revert to the saved copy. If OK, it then confirms that the old copy exists and is OK before dumping the current document and loading the old copy. The name remains the same.

Quit... confirms that the user really wants to Quit. If OK, then does an implicit Close (and Save, if dirty) of all open documents, followed by closing all other windows. It then returns to the Finder.

An implicit Save begins with checking to see if the document is dirty; the Save is skipped if it isn't. If the document is dirty, the user is asked to confirm whether to save it or not. Pressing Don't Save skips the save; pressing OK drops into the Save code.

OTHER FILING ISSUES

Changing Volumes: A Drive button on the GetFile and PutFile dialogs allows the user to cycle through the mounted volumes, showing the disk names (and, in Get, the contents). Any resulting filename is prefixed with the volume name (unless the user typed one in). This does not set the working volume. Drive does not appear on one-drive systems, and is disabled on two-drive systems with only one volume on-line.

Ejecting Disks: An Eject button on the GetFile and PutFile dialogs allows the user to eject the current volume. In the GetFile dialog, Eject cycles to the next volume, if any; if there's no next volume, the volume name and directory go blank and remain blank until another disk is inserted.

Remounting Disks: When your application gets a Disk Remount event (event bit 7) with an I/O error code, trap to package 3, which will allow the user to initialize the disk, or eject it if it's a mistake. NOTE that you should do this on every remount event with an I/O error, even during modey dialogs; be sure to check for remount errors in your modey dialog filterProcs.

Save: is an optimization of Save As to reduce button clicks, and also to work around the volume name ambiguity in saving to an off-line disk. Although we strongly recommend you support it, it is optional.

Revert to Saved: is an optimization for a "global Undo"; the user can just do a close and open. We recommend it for its clarity and speed, but it is optional.

To: All Developers
From: Cary Clark Re: Above Note

Hard Copy of the above information will be included in the next mailing.

Let me know if the info is sufficiently clear.

END/CRC

MACINTOSH USER EDUCATION

INSIDE MACINTOSH: A ROAD MAP /ROAD.MAP/ROAD

See Also: Pascal Reference Manual for the Lisa
Macintosh User Interface Guidelines
Macintosh Operating System Reference Manual
The Resource Manager: A Programmer's Guide
QuickDraw: A Programmer's Guide
The Font Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Window Manager: A Programmer's Guide
Macintosh Control Manager Programmer's Guide
The Menu Manager: A Programmer's Guide
The Dialog Manager: A Programmer's Guide
TextEdit: A Programmer's Guide
CoreEdit: A Programmer's Guide
The Desk Manager: A Programmer's Guide
The Scrap Manager: A Programer's Guide
The Toolbox Utilities: A Programmer's Guide
The Memory Manager: A Programmer's Guide
The Segment Loader: A Programmer's Guide
Putting Together a Macintosh Application
Index to Technical Documentation

Modification History:			
	First Draft (ROM 4.4)	C. Rose	8/8/83
	Second Draft (ROM 7)	C. Rose	12/22/83

ABSTRACT

This manual introduces you to the "inside" of Macintosh: the Operating System and other routines that your Macintosh application program will call. It will help you figure out which software you need to learn more about and how to proceed with the rest of the technical documentation.

Summary of significant changes and additions since last version:

- The Toolbox overview has been rewritten, and the Operating System overview has been added.
- "About Using Assembly Language" has been removed; it will be replaced by other documentation.
- "Where to Go From Here" has been updated.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual introduces you to the "inside" of Macintosh: the Operating System, the User Interface Toolbox, and other routines that your application program may call. It will help you figure out which software you need to learn more about and how to proceed with the rest of the technical documentation. *** Eventually it will be an introductory chapter in a comprehensive manual that describes everything in detail. ***

You should already be familiar with the Macintosh User Interface Guidelines. All Macintosh programmers should follow these guidelines to ensure that the end user is presented with a consistent interface. It would also be helpful for you to be familiar with an existing Macintosh application.

This manual begins with a general overview of the software your application program will use, followed by individual overviews of the User Interface Toolbox and the Operating System. Following these overviews is a section that tells you how to proceed with reading the rest of the Toolbox and Operating System documentation. Finally, there's a glossary of terms used in this manual.

GENERAL OVERVIEW

The routines available for use in Macintosh application programs are divided into functional units, many of which are called "managers" of the application feature that they support. As shown in Figure 1 on the following page, most units are part of either the Operating System or the User Interface Toolbox and are in the Macintosh ROM.

The Operating System is at the lowest level; it does basic tasks such as interrupt handling, memory management, and I/O. The User Interface Toolbox is a level above the Operating System; it exists to help you implement the standard Macintosh user interface in your application. The Toolbox calls the Operating System when necessary to do low-level operations, and you'll also call the Operating System directly yourself.

Other software is available for performing specialized operations that aren't integral to the user interface but may be useful to some applications. This includes routines for doing printing and floating-point arithmetic. Such software isn't located in the Macintosh ROM, nor are certain special-purpose Toolbox units (such as CoreEdit, for doing sophisticated text editing). The entire Operating System and all the commonly used Toolbox units are in ROM.

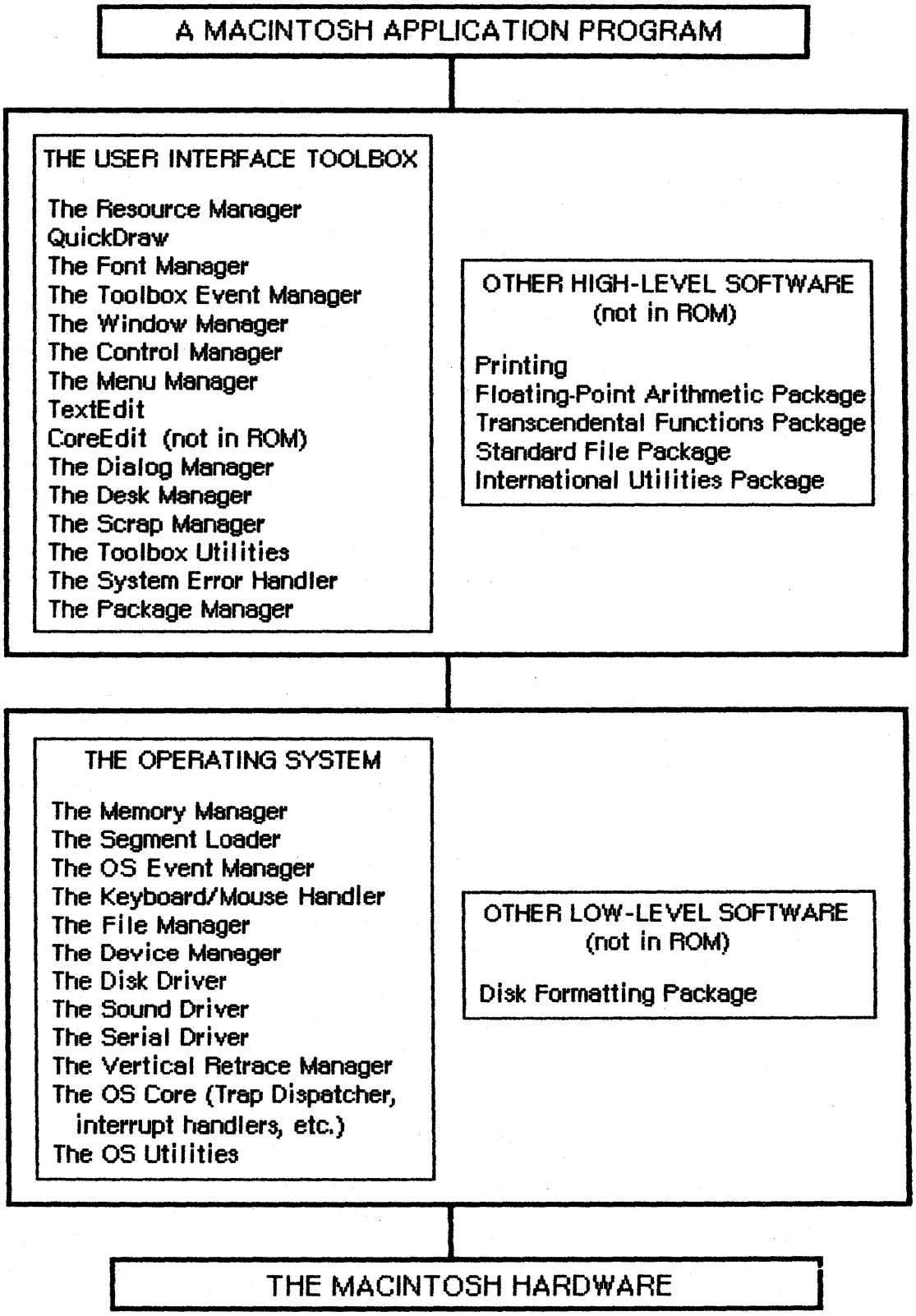


Figure 1. Overview

Macintosh applications can be written most easily in Pascal, since all units have a Pascal interface *** or will eventually ***. For greater efficiency, however, you may want to use assembly language or a combination of Pascal and assembly language. *** Currently you must develop your application on a Lisa computer and convert it to a Macintosh disk before trying it out. Eventually development will be possible on the Macintosh itself. ***

ABOUT THE USER INTERFACE TOOLBOX

The Macintosh User Interface Toolbox provides a simple means of constructing application programs that conform to the Macintosh User Interface Guidelines. By offering a common set of routines that every application calls to implement the user interface, the Toolbox not only ensures consistency but also helps reduce the application's code size and development time. At the same time, it allows a great deal of flexibility: an application can use its own code instead of a Toolbox call wherever appropriate, and can define its own types of windows, menus, controls, and desk accessories.

Figure 2 shows the Toolbox units in rough order of their level, from the lowest level at the bottom to the highest level at the top. There are many interconnections between these units; the lower-level ones are in many cases called by those at the higher levels.

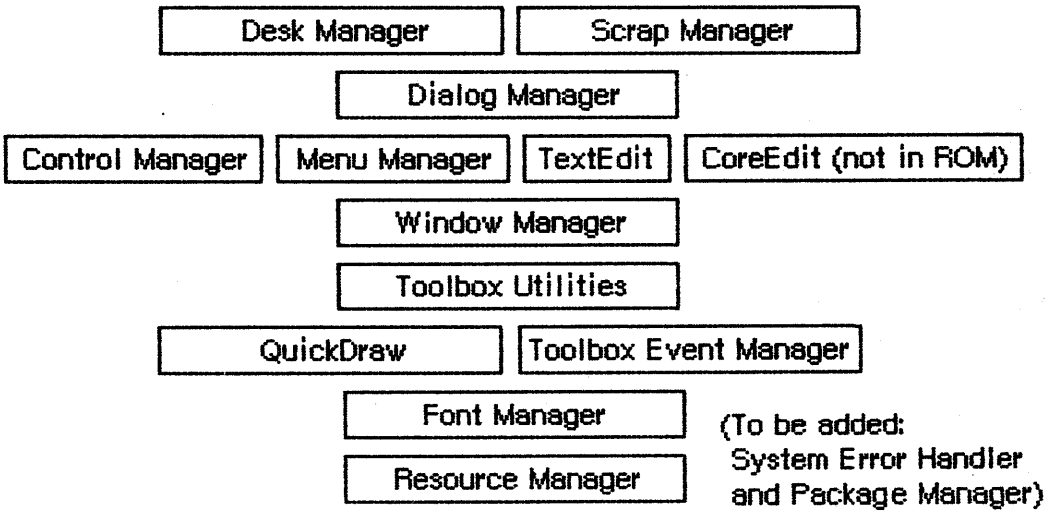


Figure 2. Toolbox Units

To keep the data of an application separate from its code, making the data easier to modify and easier to share among applications, the Toolbox includes the Resource Manager. The Resource Manager lets you, for example, store menus separately from your code so that they can be edited or translated without requiring recompilation of the code. It also allows you to get standard data, such as the wristwatch graphic that means "wait", from a shared system file. When you call other Toolbox units that need access to the data, they call the Resource Manager. Although most applications never need to call the Resource

Manager directly, an understanding of the concepts behind it is essential.

Graphics are an important part of every Macintosh application. All graphic operations on the Macintosh are performed by the QuickDraw unit. To draw something on the screen, you'll often call one of the other Toolbox units, but that unit will in turn call QuickDraw. You'll also call QuickDraw directly, usually to draw inside a window. QuickDraw's underlying concepts, like those of the Resource Manager, are important for you to understand.

Graphics include text as well as pictures. To draw text, QuickDraw calls the Font Manager, which does the background work necessary to make a variety of character fonts available in various sizes and styles. Unless an application includes a font menu, it usually need not be concerned with the Font Manager.

An application decides what to do from moment to moment by examining input from the user, in the form of mouse and keyboard actions. It learns of such actions by repeatedly calling the Toolbox Event Manager (which in turn calls another, lower-level Event Manager in the Operating System). The Toolbox Event Manager also reports occurrences within the application that may require a response, such as when a window that was overlapped becomes exposed and needs to be redrawn.

All information presented by a standard Macintosh application appears in windows. To create windows, activate them, move them, resize them, or close them, you'll call the Window Manager. It keeps track of overlapping windows, so you can manipulate windows without concern for how they overlap. The Window Manager, for example, tells the Toolbox Event Manager when to inform your application that a window has to be redrawn. Also, when the user presses the mouse button, you call the Window Manager to learn which part of which window it was pressed in, if any, or whether it was pressed in the menu bar or a desk accessory.

Any window may contain controls, such as buttons, check boxes, and scroll bars. You create and manipulate controls with the Control Manager. When you learn from the Window Manager that the user pressed the mouse button inside a window containing controls, you call the Control Manager to find out which control it was pressed in, if any.

A common place for the user to press the mouse button is, of course, in the menu bar. You set up menus in the menu bar by calling the Menu Manager. When the user gives a command, either from a menu with the mouse or from the keyboard with the Command key, you call the Menu Manager to find out which command was given.

To accept text typed by the user and allow the standard editing capabilities, such as cutting and pasting within a document via the Clipboard, your application can call either TextEdit or CoreEdit. TextEdit is especially easy to use but doesn't support advanced editing and formatting features such as fully justified text, tabbing, or recognition of word boundaries during cutting and pasting; for these, you'll have to use CoreEdit. Bear in mind, however, that CoreEdit is

not in the Macintosh ROM; instead, it occupies over 6K of your application's available memory.

When an application needs more information from the user about a command, it presents a dialog box. In case of errors or potentially dangerous situations, it gives the user an alert, in the form of an alert box or sound from the Macintosh's speaker (or both). To create and present dialogs and alerts, and find out the user's responses to them, you call the Dialog Manager.

Every Macintosh application should support the use of desk accessories. The user opens desk accessories through the Apple menu, which you set up by calling the Menu Manager. When you learn that the user has pressed the mouse button in a desk accessory, you pass that information on to the accessory by calling the Desk Manager. The Desk Manager also includes routines that you must call to ensure that desk accessories behave properly.

As mentioned above, you can use TextEdit or CoreEdit to implement the standard text editing capability of cutting and pasting via the Clipboard in your application. However, to extend the use of the Clipboard to allow cutting and pasting between your application and another application or a desk accessory, you need to call the Scrap Manager.

Finally, some generally useful operations such as fixed-point arithmetic, string manipulation, and logical operations on bits may be performed with the Toolbox Utilities.

*** To be added: System Error Handler, Package Manager, and other high-level software ***

ABOUT THE OPERATING SYSTEM

The Macintosh Operating System provides the low-level support that applications need in order to use the Macintosh hardware. As the Toolbox is your program's interface to the user, the Operating System is its interface to the Macintosh.

The Memory Manager dynamically allocates and releases memory for use by applications and by the other parts of the Operating System. Most of the memory that your program uses is in an area called the heap; the code of the program itself occupies space in the heap. Memory space in the heap must be obtained from the Memory Manager.

The Segment Loader is the part of the Operating System that loads the program code into memory to be executed. Your program can be loaded all at once as a single unit, or you can divide it up into dynamically loaded segments to economize on memory usage.

Low-level, hardware-related events such as mouse-button presses and keystrokes are reported by the Operating System Event Manager. (The

Toolbox Event Manager then passes them along to the application, along with higher-level, software-generated events added at the Toolbox level.) The Operating System Event Manager learns of mouse and keyboard actions in particular from the Keyboard/Mouse Handler. Your program will ordinarily deal only with the Toolbox Event Manager and rarely call the Operating System Event Manager or the Keyboard/Mouse Handler directly.

File I/O is supported by the File Manager, and device I/O by the Device Manager. The task of making the various types of devices present the same interface to the application is performed by specialized device drivers. The Operating System includes three built-in drivers:

- The Disk Driver controls data storage and retrieval on 4ØØK-byte 3 1/2-inch disks.
- The Sound Driver controls sound generation, including music composed of four simultaneous tones.
- The Serial Driver reads and writes asynchronous data through the two serial ports, providing communication between applications and serial peripheral devices such as a modem or printer.

Other drivers can be added independently or built on the existing drivers. For example, a printer driver can be built on the Serial Driver or a music driver built on the Sound Driver.

The Macintosh video circuitry generates a vertical retrace interrupt (also known as the vertical blanking or VBL interrupt) sixty times a second while the beam of the display tube returns from the bottom of the screen to the top to display the next frame. The system uses this interrupt as a convenient time to perform recurrent tasks such as checking the state of the mouse button. An application can also schedule routines to be executed at regular intervals based on this "heartbeat" of the system. The Vertical Retrace Manager handles the scheduling and execution of tasks during the vertical retrace interrupt.

At the very lowest level is the Operating System Core, which does the actual interrupt handling, initialization, and other important background work necessary to keep the Macintosh functioning. Via the Trap Dispatcher, it provides the connection between your request for a Toolbox or Operating System service and the physical code that performs that service.

Finally, there are miscellaneous Operating System Utilities for doing such things as setting the date and time or finding out the user's preferred speaker volume.

*** To be added: other low-level software (Disk Formatting Package) ***

WHERE TO GO FROM HERE

*** This section will be considerably rewritten for the final comprehensive manual. ***

The technical documentation will eventually be ordered in such a way that you can follow it if you read it sequentially. The proposed order for the documentation that's already written is given below. Before you begin, you should be familiar with Lisa Pascal, as described in the Pascal Reference Manual for the Lisa. You should also know a little bit about the Macintosh's memory management--heaps, handles, and the like. For now you can read about these in the Memory Manager manual, from "About the Memory Manager" through "Utility Data Types"; eventually there will be a separate overview of memory management.

The Resource Manager: A Programmer's Guide
QuickDraw: A Programmer's Guide
The Font Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Window Manager: A Programmer's Guide
Macintosh Control Manager Programmer's Guide
The Menu Manager: A Programmer's Guide
TextEdit: A Programmer's Guide
CoreEdit: A Programmer's Guide
The Dialog Manager: A Programmer's Guide
The Desk Manager: A Programmer's Guide
The Scrap Manager: A Programmer's Guide
The Toolbox Utilities: A Programmer's Guide
The Memory Manager: A Programmer's Guide
Macintosh Operating System Reference Manual
The Segment Loader: A Programmer's Guide
Putting Together a Macintosh Application

(hand)

The Macintosh Operating System Reference Manual is very out-of-date, incomplete, and in a different format from the other manuals. It will eventually be completely replaced by up-to-date documentation in the usual format.

(hand)

Anything not listed above hasn't been documented yet by Macintosh User Education, although programmer's notes or other preliminary documentation may be available. Check with Macintosh Technical Support.

The individual manuals identify any special-purpose information that can possibly be skipped. Most likely you won't need to read everything in each manual and can even skip entire manuals. You should at least read the manuals on the Toolbox units that deal with the fundamental aspects of the user interface: the Resource Manager, QuickDraw, the Toolbox Event Manager, the Window Manager, and the Menu Manager. Read the other manuals if you're interested in what they discuss, which you should be able to tell from the above overviews and from the

introductions to the manuals themselves. Each manual's introduction will also tell you what you should already know before reading that manual.

The documentation is oriented toward Pascal programmers. If you want to program in assembly language, read the "Using QuickDraw from Assembly Language" section of the QuickDraw manual. (Eventually that section will be removed and there will be a separate, more detailed discussion of using assembly language.) There are also notes for assembly-language programmers throughout every manual.

Read the manual "Putting Together a Macintosh Application" when you're ready to try something out. Currently the documentation doesn't include any sample programs, but you can get some through Macintosh Technical Support in the meantime.

GLOSSARY

Control Manager: A Toolbox unit that provides routines for creating and manipulating controls (such as buttons, check boxes, and scroll bars).

CoreEdit: A Toolbox unit that handles sophisticated text editing and formatting, including fully justified text, tabbing, and recognition of word boundaries during cutting and pasting.

Desk Manager: A Toolbox unit that supports the use of desk accessories from an application.

device driver: A piece of Operating System software that controls a peripheral device and makes it present the standard interface to the application.

Device Manager: The part of the Operating System that supports device I/O.

Dialog Manager: A Toolbox unit that provides routines for implementing dialogs and alerts.

Disk Driver: The device driver that controls data storage and retrieval on 4ØØK-byte 3 1/2-inch disks.

Event Manager: See Toolbox Event Manager or Operating System Event Manager.

File Manager: The part of the Operating System that supports file I/O.

Font Manager: A Toolbox unit that supports the use of various character fonts for QuickDraw when it draws text.

heap: An area of memory in which space can be allocated and released on demand, using the Memory Manager.

Keyboard/Mouse Handler: The part of the Operating System that controls communication with the keyboard and the mouse.

Memory Manager: The part of the Operating System that dynamically allocates and releases memory space in the heap.

Menu Manager: A Toolbox unit that deals with setting up menus and letting the user choose from them.

Operating System: The lowest-level software in the Macintosh. It does basic tasks such as interrupt handling, memory management, and I/O.

Operating System Core: The part of the Operating System that does the actual interrupt handling, initialization, and other important background work necessary to keep the Macintosh functioning.

Operating System Event Manager: The part of the Operating System that reports hardware-related events such as mouse-button presses and keystrokes.

Operating System Utilities: Operating System routines that perform miscellaneous tasks such as setting the date and time or finding out the user's preferred speaker volume.

QuickDraw: The Toolbox unit that performs all graphic operations on the Macintosh screen.

resource: Data used by an application (such as menus, fonts, and icons), and also the application code itself.

Resource Manager: The Toolbox unit that reads and writes resources.

Scrap Manager: The Toolbox unit that enables cutting and pasting between applications, desk accessories, or an application and a desk accessory.

Segment Loader: The part of the Operating System that loads the code of an application into memory, either as a single unit or divided into dynamically loaded segments.

Serial Driver: The device driver that controls communication, via serial ports, between applications and serial peripheral devices.

Sound Driver: The device driver that controls sound generation in an application.

TextEdit: A Toolbox unit that supports the basic text entry and editing capabilities of a standard Macintosh application.

Toolbox: Same as User Interface Toolbox.

Toolbox Event Manager: A Toolbox unit that allows your application program to monitor the user's actions with the mouse, keyboard, and keypad.

Toolbox Utilities: A Toolbox unit that performs generally useful operations such as fixed-point arithmetic, string manipulation, and logical operations on bits.

Trap Dispatcher: The part of the Operating System Core that provides the connection between your request for a Toolbox or Operating System service and the physical code that performs that service.

User Interface Toolbox: A set of routines and data types that help you implement the standard Macintosh user interface in your application.

vertical retrace interrupt: An interrupt generated sixty times a second by the Macintosh video circuitry while the beam of the display tube returns from the bottom of the screen to the top; also known as the vertical blanking or VBL interrupt.

Vertical Retrace Manager: The part of the Operating System that schedules and executes tasks during the vertical retrace interrupt.

Window Manager: A Toolbox unit that provides routines for creating and manipulating windows.

COMMENTS?

Macintosh User Education encourages your comments on this manual.

- What do you like or dislike about it?

- Were you able to find the information you needed?

- Was it complete and accurate?

- Do you have any suggestions for improvement?

Please send your comments to the author (indicated on the cover page) at 10460 Bandley Drive M/S 3-G, Cupertino CA 95014. Mark up a copy of the manual or note your remarks separately. (We'll return your marked-up copy if you like.)

Thanks for your help!

MACINTOSH USER EDUCATION

Programming Macintosh Applications in Assembly Language /INTRO/ASSEM

See Also: Macintosh Memory Management: An Overview
The Memory Manager: A Programmer's Guide
The Segment Loader: A Programmer's Guide
The Operating System Utilities: A Programmer's Guide
Putting Together a Macintosh Application

Modification History: First Draft S. Chernicoff 2/27/84

ABSTRACT

This manual tells you what you need to know to write all or part of your Macintosh application program in assembly language. The emphasis here is on general principles and methods; details on specific OS and Toolbox routines are given elsewhere.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual tells you what you need to know to write all or part of your Macintosh application program in assembly language. The emphasis here is on general principles and methods; details on specific OS and Toolbox routines are given elsewhere.

The manual assumes you already know how to write assembly language for the Motorola MC68ØØØ (or just "68ØØØ" for short), the microprocessor used in the Macintosh. It also assumes you're familiar with Lisa Pascal and its associated software development tools, particularly the Assembler, the Pascal Compiler, and the Linker. *** (Currently, all software for the Macintosh must be developed on a Lisa computer and written on a Macintosh-formatted disk for execution on the Macintosh. Eventually development tools will be available on the Macintosh itself.) ***

The manual begins by discussing the various files of definitions pertaining to the OS and Toolbox, and what they contain. Then it describes the Macintosh's memory layout and organization. This is followed by a description of the dispatch table and the trap mechanism, which allow your program to use the OS and Toolbox while remaining independent of specific addresses in the Macintosh ROM. Next is a discussion of the calling conventions for using the OS and Toolbox from assembly language and for mixing Pascal and assembly language in your own programs. Finally, there's a glossary of terms used in this manual.

DEFINITION FILES

The primary aids to assembly-language programmers are a set of definition files that define various symbolic names for use in assembly-language programs. By naming the definition files in an .INCLUDE directive, you make the definitions available to your program.

The most important of the definition files are the equates files, which use .EQU directives to define values for symbolic names. There are separate system, QuickDraw, and Toolbox equates files for definitions related to the Operating System, QuickDraw, and the User Interface Toolbox. There are also a number of specialized equates files, such as the memory equates file, which contains definitions pertaining to memory allocation. These specialized files are discussed in the individual manuals that apply to them (for instance, the memory equates file is covered in the Memory Manager manual).

The equates files define a variety of symbolic names for various purposes, such as:

- Useful numeric quantities. For example, the constant maxMenu stands for the maximum number of menus in a menu bar.

- Fixed memory addresses. For example, sysCom is the starting address of the system communication area.

- Addresses of system variables. For example, ticks is the address of a long-word integer variable containing the elapsed time in ticks (sixtieths of a second) since the system was last started up. Often the global variable in turn contains an address: for example, sysEvtBuf is the address of a pointer to the system event buffer (**not** the address of the buffer itself!).

- Masks. For example, tagMask is a mask for extracting the tag field from the header of a memory block.

- Bit numbers. For example, lock is the bit number of the lock bit in the first byte of a master pointer, defined for use with the bit manipulation instructions BTST (Bit Test), BSET (Bit Set), BCLR (Bit Clear), and BCHG (Bit Change).

- Codes. For example, inMenuBar is the code returned by the Window Manager function FindWindow when the user presses the mouse button inside the menu bar.

- Offsets into data structures. For example, wVisible is the offset of a window's "visible" flag relative to the beginning of the window record.

It's a good idea always to use the symbolic names defined in an equates file in place of the corresponding numerical values (even if you know them), since some of these values may be subject to change. One thing to watch out for is that the names of the offsets for a data structure don't always match the field names in the corresponding Pascal definition. In the OS and Toolbox documentation, the definitions are normally shown in their Pascal form; the corresponding offset constants for assembly-language use are listed in the summary at the end of each manual.

In addition to the equates files, there's also a system errors file, which defines symbolic names for all error codes returned by Operating System routines. Finally, there are the system, QuickDraw, and Toolbox macro files, which define the macros used to call OS and Toolbox routines from assembly language.

MEMORY ORGANIZATION

In its current configuration, the Macintosh has 128K bytes of volatile read/write memory (RAM) and 64K bytes of permanent read-only memory (ROM). The ROM contains the built-in code of the Operating System and User Interface Toolbox, which is available for use by any application program. In the 68ØØØ's 16-megabyte address space, RAM occupies addresses $Ø-$1FFFF and ROM is at addresses $4ØØØØØ-$4ØFFFF.

In addition, the various built-in input/output devices are "memory-mapped", meaning that they appear to the processor as addressable memory locations with special properties. The 6522 VIA (Versatile Interface Adapter) occupies addresses in the range $EØØØØØ-$EFFFFF, the 853Ø SCC (Serial Communications Controller) $9ØØØØØ-$9FFFFF and $BØØØØØ-$BFFFFF, and the IWM ("Integrated Woz Machine") disk interface $DØØØØØ-$DFFFFF. You won't ordinarily need to know any details about these memory-mapped devices, since you'll deal with them exclusively through the Operating System.

(warning)

> All specific memory addresses given in this section refer to the first-release, 128K Macintosh. The Lisa 2 Macintosh emulator uses a different memory layout, as will future versions of Macintosh with different memory capacities. For compatibility, always refer to these RAM addresses by their symbolic names (given in a table below) rather than their numeric values. For calls to OS and Toolbox routines located in ROM, use the 68ØØØ's unimplemented instruction trap, as described below under "The Trap Mechanism". This ensures compatibility by making all ROM references indirectly, through a dispatch table kept in RAM.

The organization of RAM is shown in Figure 1. The first $1ØØ bytes (addresses $Ø-$FF) are reserved by the 68ØØØ hardware for use as exception vectors. The next $3ØØ bytes ($1ØØ-$3FF), referred to as the "system communication area", contain global variables used by various parts of the Macintosh system software. The next $4ØØ bytes ($4ØØ-$7FF) contain the dispatch table for OS and Toolbox routines, discussed below under "The Dispatch Table". This is followed by $3ØØ bytes ($8ØØ-$AFF) of additional system globals.

At (almost) the very end of memory are the main sound buffer ($1FDØØ-$1FFE3), used by the Sound Driver to control the sounds emitted by the built-in speaker, and the main screen buffer ($1A7ØØ-$1FC7F), which holds the bit image to be displayed on the Macintosh screen. If an interactive debugger such as MacsBug is installed, it immediately precedes the screen buffer. Then comes an area reserved for the application's parameters and global variables, which normally also includes a block of global variables belonging to QuickDraw. When the Segment Loader starts up an application, it adjusts the size of this area according to the application's needs and sets register A5 to point to the boundary between the application's parameters and globals. (This subject is covered in more detail in the Segment Loader manual.)

(note)

> For special applications, there are an alternate screen buffer ($127ØØ-$17C7F) and an alternate sound buffer ($1A1ØØ-$1A3E3). If you use either or both of these, the application parameters (or the debugger, if there is one) end at $126FF or $1AØFF instead of the normal $1A6FF, and the space available for dynamic allocation (see below) is reduced accordingly.

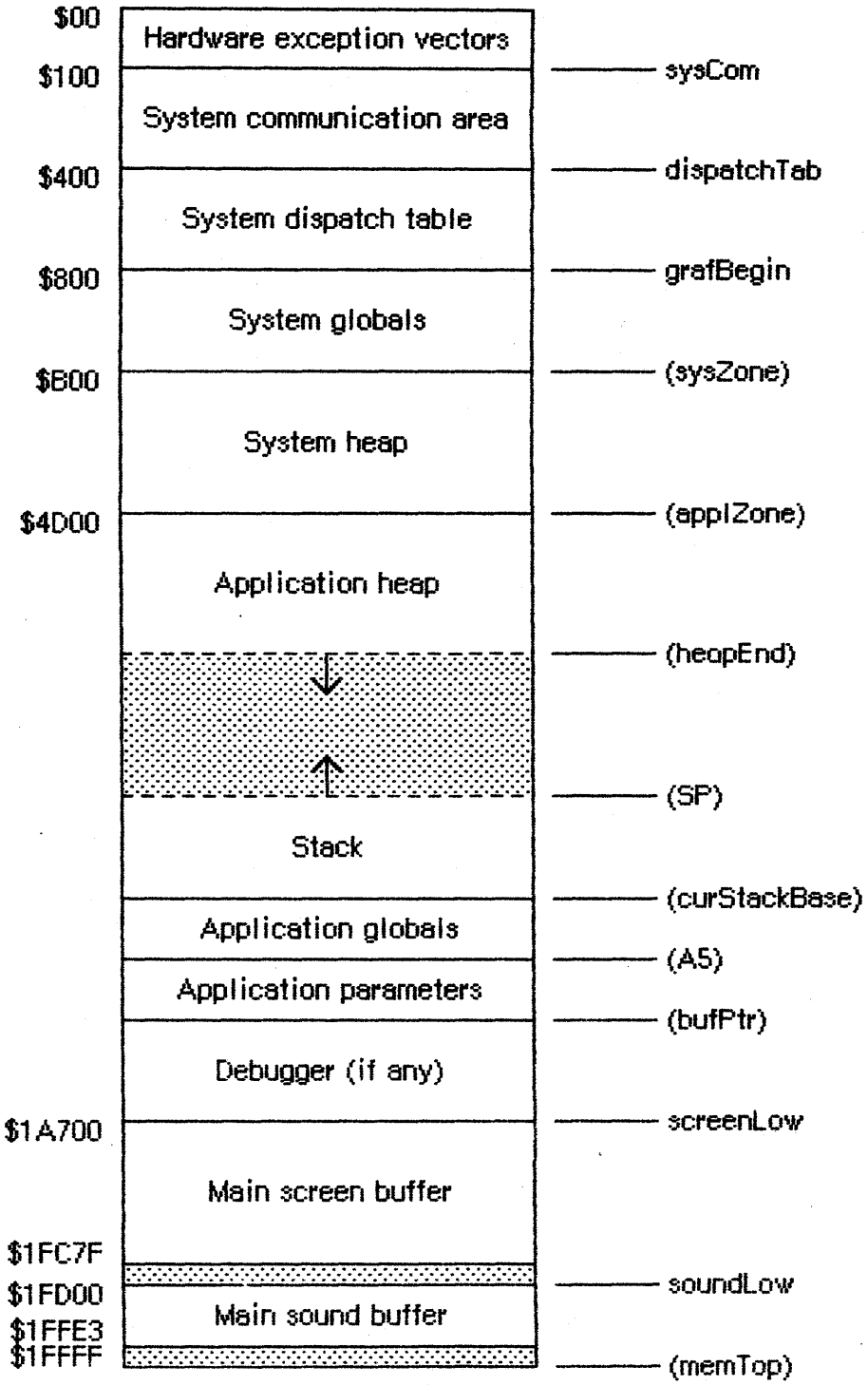


Figure 1. RAM Organization

All remaining space, between the end of the system globals ($BØØ) and the beginning of the application globals, is available for dynamic allocation by the running program. This space is shared between the stack and the heap, with the heap growing forward from the beginning of the space and the stack growing backward from the end. (The stack and the heap are discussed in general terms in the document "Macintosh Memory Management: An Overview" *** which will be the chapter preceding this one in the eventual "Inside Macintosh" manual *** and in greater detail in the Memory Manager manual.)

Immediately following the system globals is the system heap, which is initialized to a fixed size (currently 16.5K, or $42ØØ bytes) when the system is started up. The system heap is intended for the system's own private use; your application program should use the application heap for all of its heap allocation. (In particular, the code of the application itself resides in the application heap.) The application heap is initialized at the start of each new application program (currently to 6K, or $18ØØ bytes), and can then expand as required to accommodate the application's needs. The stack grows and shrinks from the other end of the space.

(warning)

> Although the 68ØØØ hardware provides for separate user and supervisor stacks, each with its own stack pointer, the Macintosh maintains only one stack. All application programs run in supervisor mode and share the same stack with the system; the user stack pointer isn't used.

The boundaries between the various areas of RAM are marked by global constants and variables defined in the equates files. In the following table (as well as in Figure 1), names not shown in parentheses are constants that are equated directly to the designated address; those in parentheses are variables containing long-word pointers that in turn point to the address. Names identified as marking the end of an area actually refer to the address **following** the last byte in that area.

Name	Meaning
sysCom	Start of system communication area
dispatchTab	Start of system dispatch table
grafBegin	Start of additional system globals
(sysZone)	Start of system heap
(applZone)	Start of application heap
(heapEnd)	End of application heap
(curStackBase)	Base (end) of stack; start of application globals
(bufPtr)	End of application parameters
screenLow	Start of main screen buffer
(scrnBase)	Start of current screen buffer
soundLow	Start of main sound buffer
(soundBase)	Start of current sound buffer
(memTop)	End of RAM
romStart	Start of ROM

THE DISPATCH TABLE

The bulk of the Operating System and Toolbox resides in read-only memory (ROM). However, to allow flexibility for future development, application code must be kept free of any specific ROM addresses. So all references to OS and Toolbox routines are made indirectly, through a dispatch table in RAM containing the addresses of the routines. As long as the location of the dispatch table is known, the routines themselves can be moved to different locations in ROM without disturbing the operation of programs that depend on them.

Information about the locations of the various OS and Toolbox routines is encoded in compressed form in the ROM itself. When the system is started up, this encoded information is expanded to form the dispatch table. Because the dispatch table resides in RAM (locations $4ØØ-$7FF), individual entries can be "patched" to point to addresses other than the original ROM address. This allows changes to be made in the ROM code by loading corrected versions of individual routines into RAM at system startup and patching the dispatch table to point to them. It also allows an application program to replace specific OS and Toolbox routines with its own "custom" versions. A pair of utility routines for manipulating the dispatch table, GetTrapAddress and SetTrapAddress, are described in the Operating System Utilities manual.

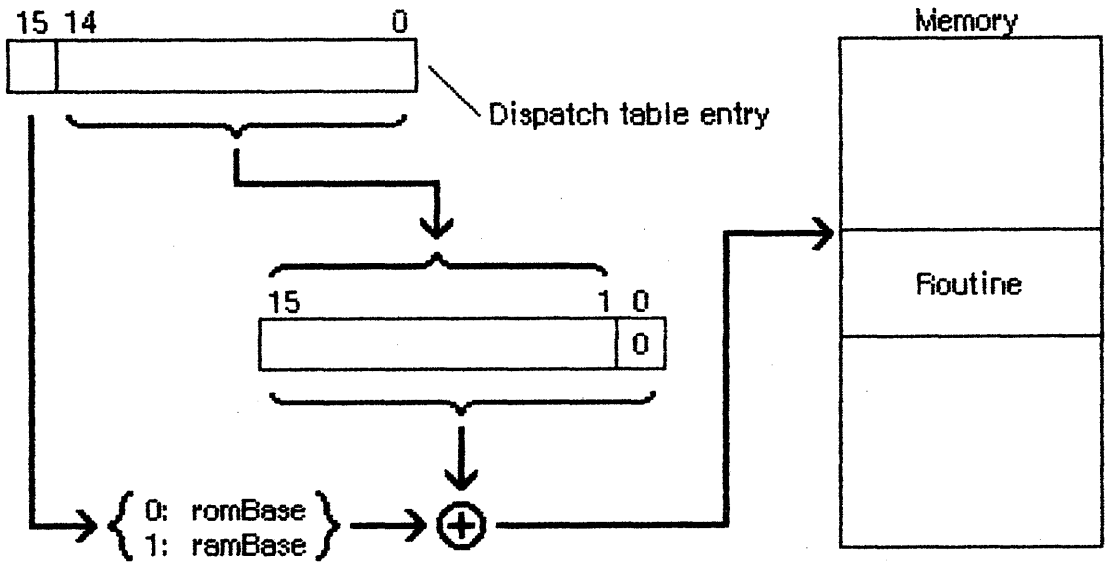


Figure 2. Dispatch Table Entry

For compactness, entries in the dispatch table are encoded into one word each, instead of a full long-word address (see Figure 2). Since the dispatch table is 1Ø24 ($4ØØ) bytes long, it has room for 512 word-length entries. The high-order bit of each entry tells whether the routine resides in ROM (Ø) or RAM (1). The remaining 15 bits give the offset of the routine relative to a base address. For routines in ROM, this base address is the beginning of the ROM, address $4ØØØØØ; for routines in RAM, it's the beginning of the system heap, currently at address $BØØ.

(note)

The two base addresses are kept in a pair of global variables named romBase and ramBase.

The offset in a dispatch table entry is expressed in words instead of bytes, taking advantage of the fact that instructions must always fall on word boundaries (even byte addresses). To find the absolute address of the routine, the system checks the high-order bit of the dispatch table entry to find out which base address to use, doubles the offset to convert it from words to bytes, and adds the result to the designated base address.

Using 15-bit word offsets, the dispatch table can address locations within a range of 32K words, or 64K bytes, from the base address. Starting from romBase, this range is big enough to cover the entire ROM; but only half of the 128K RAM lies within range of ramBase. Since all RAM-based code resides in the heap, ramBase is set to the beginning of the system heap to maximize the amount of useful space within range.

Locations below the start of the heap ($BØØ) are used to hold global system data (including the dispatch table itself), and can never contain executable code; but if the heap is big enough, it's possible for some of the application's code to lie beyond the upper end of the dispatch table's range ($1ØAFF). Any such code is inaccessible through the dispatch table.

(note)

This problem will become particularly acute on the Lisa 2 and on future versions of Macintosh with more than 128K of RAM. To make sure they lie within range of ramBase, patches to OS and Toolbox routines are typically placed in the system heap rather than the application heap.

THE TRAP MECHANISM

Calls to the OS and Toolbox via the dispatch table are implemented by means of the 68ØØØ processor's "1Ø1Ø emulator" trap. To issue such a call in assembly language, you use one of the trap macros defined in the system, QuickDraw, and Toolbox macro files. When you assemble your program, the macro generates a trap word in the machine-language code. A trap word always begins with the hexadecimal digit $A (binary 1Ø1Ø); the rest of the word identifies the routine you're calling, along with some additional information pertaining to the call.

Instruction words beginning with $A don't correspond to any valid machine-language instruction, and are known as unimplemented instructions. They're used to augment the processor's native instruction set with additional operations that are "emulated" in software instead of being executed directly by the hardware. On the Macintosh, the additional operations are the OS and Toolbox routines. Attempting to execute an unimplemented instruction causes a trap to the Trap Dispatcher, which examines the bit pattern of the trap word to determine what operation it stands for, looks up the address of the corresponding routine in the dispatch table, and jumps to the routine.

Format of Trap Words

As noted above, a trap word always begins with the digit $A in bits 12-15, the mark of an unimplemented instruction. Bit 11 tells whether the call is to the Operating System (Ø) or the Toolbox (1). The format of the rest of the word depends on whether it's an OS or a Toolbox call.

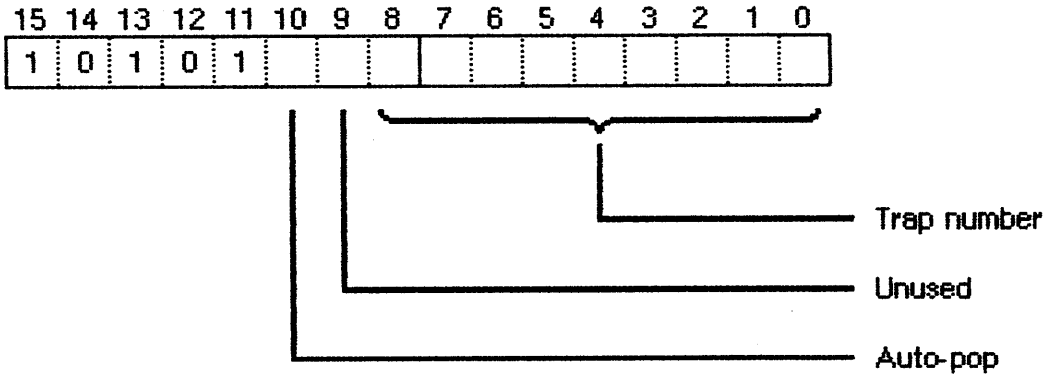


Figure 3. Trap Word Format for Toolbox Calls

Figure 3 shows the trap word format for Toolbox calls. Bits Ø-8 form a 9-bit trap number identifying the particular Toolbox routine being called. Bit 9 is unused; bit 1Ø is called the "auto-pop" bit and is discussed below under "Pascal Interface to the OS and Toolbox".

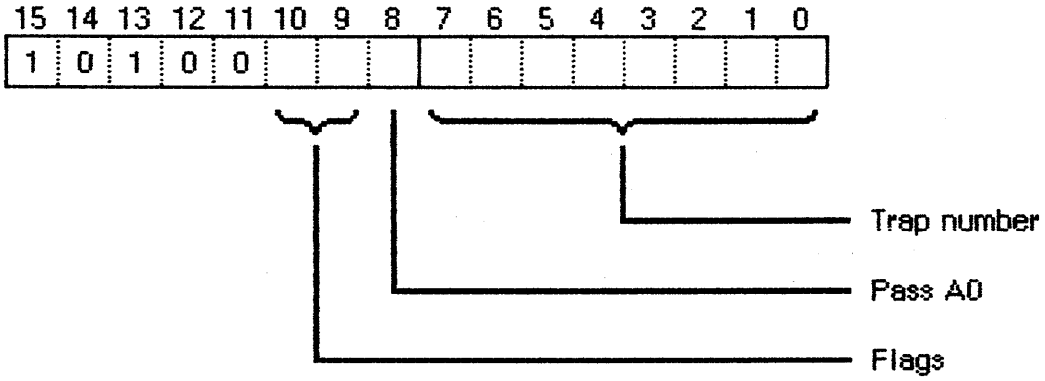


Figure 4. Trap Word Format for OS Calls

For Operating System calls, only the low-order 8 bits (bits Ø-7) are used for the trap number (see Figure 4). Thus of the 512 entries in the dispatch table, only the first 256 can be used for OS traps. Bit 8 of an OS trap has to do with register usage and is discussed below under "Register-Saving Conventions". Bits 9 and 1Ø have specialized meanings depending on which OS routine you're calling, and are covered where relevant in other manuals.

Trap Macros

The names of all trap macros begin with the underscore character (_), followed by the name of the corresponding routine. As a rule, the macro name is the same as the name used to call the routine from Pascal, as given in the OS and Toolbox documentation. For example, to call the Window Manager routine NewWindow, you would use an instruction with the macro name _NewWindow in the op code field. There are a few exceptional cases, however, in which the spelling of the macro name differs from the name of the routine itself; these exceptions are noted in the documentation for the individual routines.

Trap macros for Toolbox calls take no arguments; those for OS calls may have as many as three optional arguments. The first argument, if present, is used to load a register with a parameter value for the routine you're calling, and is discussed below under "Register-Based Calls". The remaining arguments control the settings of the various flag bits in the trap word. The form of these arguments varies with the meanings of the flag bits, and is described in the manuals on the relevant parts of the Operating System.

CALLING CONVENTIONS

The calling conventions for Operating System and Toolbox routines fall into two categories: register-based and stack-based. As the terms imply, register-based routines receive their parameters and return their results in the processor's registers; stack-based routines communicate via the stack, following the same conventions used by the Pascal Compiler for routines written in Pascal. Before calling any OS or Toolbox routine, you have to set up the parameters in the way the routine expects.

(note)

As a general rule, Operating System routines are register-based and Toolbox routines stack-based, but there are exceptions on both sides. Throughout this documentation, register-based calling conventions are given for all routines that have them; if none is shown, then the routine is stack-based.

Register-Based Calls

By convention, register-based routines normally use register AØ for passing addresses (such as pointers to data objects) and DØ for other data values (such as integers). Depending on the routine, these registers may be used to pass parameters to the routine, result values back to the calling program, or both. For routines that take more than two parameters (one address and one data value), the parameters are normally collected in a parameter block in memory and a pointer to the parameter block is passed in AØ. However, not all routines obey these

conventions; for example, some expect parameters in other registers, such as A1. See the documentation on each individual routine for details.

Whatever the conventions may be for a particular routine, it's up to you to set up the parameters in the appropriate registers before calling the routine. For instance, the Memory Manager utility procedure BlockMove, which copies a block of consecutive bytes from one place to another in memory, expects to find the address of the first source byte in register A0, the address of the first destination location in A1, and the number of bytes to be copied in D0. So to move 20 bytes beginning at address srcAddr to locations beginning at destAddr, you might write something like

```
LEA       srcAddr,A0      ;source address in A0
LEA       destAddr,A1     ;destination address in A1
MOVEQ     #20,D0          ;byte count in D0
_BlockMove                ;trap to routine
```

Because many register-based routines expect to find an address of some sort in register A0, the trap macros allow you to specify the contents of that register as an argument to the macro instead of explicitly setting up the register yourself. The first argument you supply to the macro, if any, represents an address to be passed in A0. The macro will load the register with an LEA (Load Effective Address) instruction before trapping to the routine. So, for instance, to perform a Read operation on a file, you could set up the parameter block for the operation and then use the instruction

```
_Read     paramBlock      ;trap to routine with
                          ; pointer to parameter
                          ; block in A0
```

This feature is purely a convenience, and is optional: if you don't supply any arguments to a trap macro, or if the first argument is null, the LEA to A0 will be omitted from the macro expansion. Notice that A0 is loaded with the actual address denoted by the argument, not the contents of that address.

(note)

You can use any of the 68000's addressing modes to specify this address, with one exception: you can't use the two-register indexing mode ("address register indirect with index and displacement"). An instruction such as

```
_Read     offset(A3,D5)
```

won't work properly, because the comma separating the two registers will be taken as a delimiter marking the end of the macro argument.

Many register-based routines return a 16-bit result code in the low-order half of register DØ to report successful completion or failure due to some error condition. A negative result code always signals an error of some kind; a code of Ø denotes successful completion. (Some routines also use DØ to return an actual data result. In these cases, any nonnegative value in the low-order half of the register represents a true result and implies successful completion of the routine.) The system errors file defines symbolic names for all result codes reported by the various OS routines.

Just before returning from a register-based call, the Trap Dispatcher tests the low-order half of DØ with a TST.W instruction to set the processor's condition codes. You can then check for an error by branching directly on the condition codes, without any explicit test of your own: for example,

```
_PurgeMem               ;trap to routine
BMI     Error           ;branch on error

. . .                   ;no error--actual result
                        ; in low half of DØ
```

(warning)
Not all register-based routines return a result code. Some leave the contents of DØ unchanged; others use the full 32 bits of the register to return a long-word result. See the documentation of individual routines for details.

Stack-Based Calls

To call a stack-based routine from assembly language, you have to set up the parameters on the stack in the same way the compiled object code would if your program were written in Pascal. The number and types of parameters expected on the stack depend on the routine being called. The number of bytes each parameter occupies depends on its type:

Parameter type	Number of bytes	Contents
BOOLEAN	1 byte	Low-order bit = Ø (FALSE) or 1 (TRUE)
CHAR	1 byte	ASCII character code
INTEGER	2 bytes	Twos-complement integer
LongInt	4 bytes	Twos-complement integer
REAL	4 bytes	Sign bit, 8-bit biased exponent, 23-bit mantissa
String	4 bytes	Pointer to string; first byte pointed to gives length of string in characters
Record, array	1-4 bytes	Contents of structure if <= 4 bytes; otherwise pointer to structure
Pointer	4 bytes	Address of value
Handle	4 bytes	Address of master pointer
VAR parameter	4 bytes	Address of variable, regardless of type

If the routine you're calling is a function, the first step is to reserve space on the stack for the function result. Then, for both functions and procedures, push the parameters onto the stack in the order they occur in the routine's Pascal definition. Finally, call the routine by executing the corresponding trap macro. The trap pushes the return address onto the stack, along with an extra word of processor status information. The Trap Dispatcher removes this extra status word, leaving the stack in the state shown in Figure 5 on entry to the routine. The routine itself is responsible for removing its own parameters from the stack before returning. If it's a function, it leaves its result on top of the stack; if it's a procedure, it restores the stack to the same state it was in before the call.

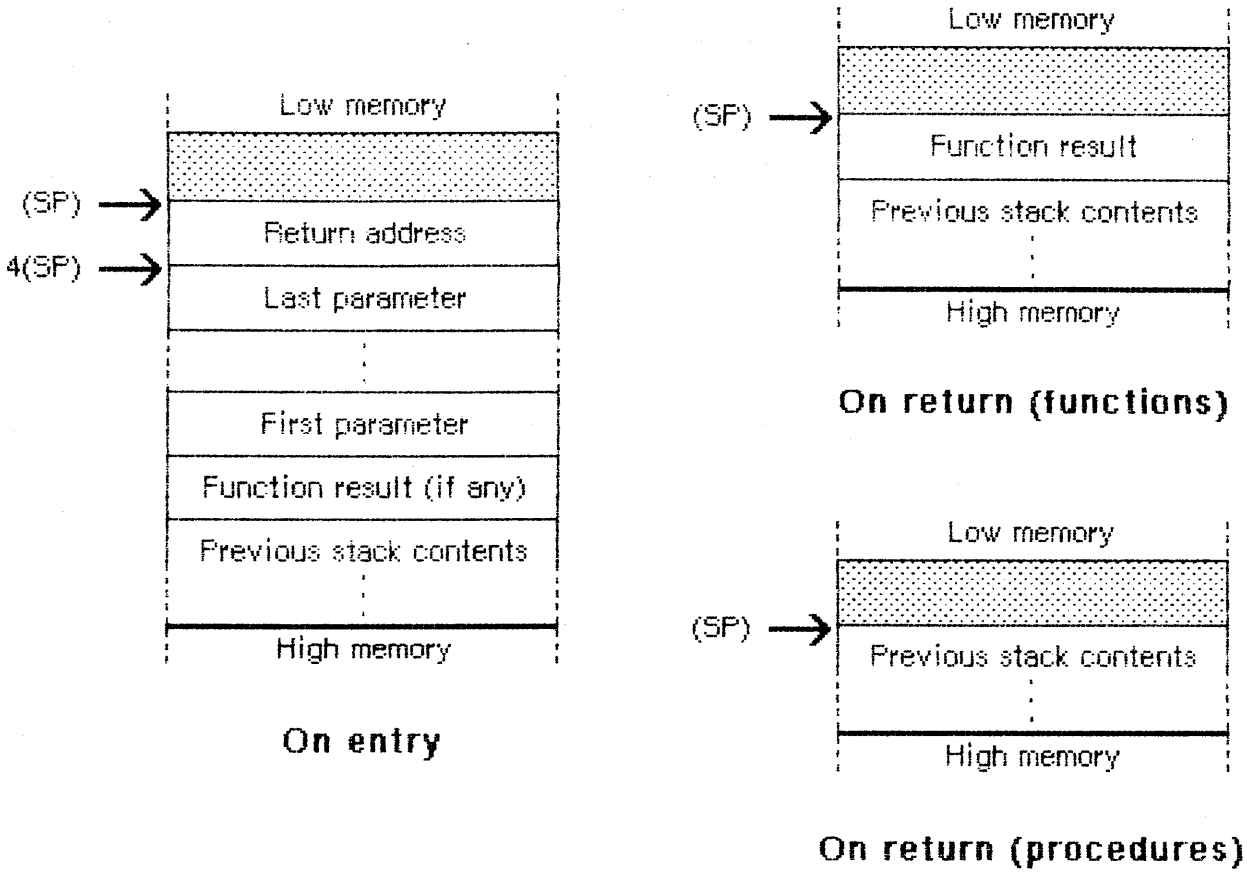


Figure 5. Stack Format for Stack-Based Calls

For example, the Window Manager function GrowWindow is defined in Pascal as follows:

```
FUNCTION GrowWindow (theWindow: WindowPtr; startPt: Point;
            sizeRect: Rect) : LongInt;
```

To call this function from assembly language, you'd write something like the following:

```
SUBQ.L  #4,SP               ;make room for LongInt result
MOVE.L  theWindow,-(SP)     ;push window pointer
MOVE.L  startPt,-(SP)       ;a Point is a 4-byte record,
                            ;  so push actual contents
PEA     sizeRect            ;a Rect is an 8-byte record,
                            ;  so push a pointer to it
_GrowWindow                 ;trap to routine
MOVE.L  (SP)+,D3            ;pop result from stack
```

(warning)

Don't forget that the stack pointer must always be aligned on a word boundary (that is, at an even byte address). When pushing a value with an odd number of bytes (such as a Boolean or a character), you have to add a byte of "padding" to keep the stack pointer even. Because all Macintosh application code runs in the 68ØØØ's supervisor mode, an odd stack pointer will cause a "double bus fault": a catastrophic system failure from which the only escape is to turn the power off and restart the machine.

(note)

To keep the stack pointer properly aligned, the 68ØØØ automatically adjusts the pointer by 2 instead of 1 when you move a byte-length value to or from the stack. This special case applies only when three conditions are met: a one-byte value is being transferred; either the source or the destination is specified by predecrement or postincrement addressing; and the register being decremented or incremented is the stack pointer (A7). For example, you can push the Boolean value TRUE onto the stack with the instruction

```
ST.B    -(SP)           ;byte-length
                        ; predecrement to
                        ; stack pointer
```

and an extra, unused byte will automatically be added to keep the stack pointer even.

However, when you use any other method to manipulate the stack pointer, it's your responsibility to make sure the pointer stays properly aligned. For instance, to reserve space on the stack for a Boolean function result, you have to remember to decrement explicitly by two bytes instead of one:

```
SUBQ.L  #2,SP           ;make room for
                        ; Boolean result
```

The function will return its result in the high-order (even-addressed) byte of the two; the other byte is just padding and should be ignored.

Register-Saving Conventions

All OS and Toolbox routines follow Lisa Pascal's register-saving conventions, which require the routine to preserve the contents of all registers except AØ, A1, and DØ-D2 (and of course A7, which is special). In addition, for register-based routines, the Trap Dispatcher saves some of the remaining registers before dispatching to the routine and restores them before returning to the calling program.

Registers A1, D1, and D2 are always saved and restored in this way, so their contents are unaffected by a register-based trap even though the routine itself is allowed to "trash" them. A7 and DØ are never restored: whatever the routine leaves in these registers is passed back unchanged to the calling program, allowing the routine to manipulate the stack pointer as appropriate and to return a result code.

Whether the Trap Dispatcher preserves register AØ depends on the setting of bit 8 of the trap word. If this bit is Ø, AØ is saved and restored; if it's 1, AØ is passed back from the routine unchanged. Thus bit 8 of the trap word should be set to 1 only for those routines that return a result in AØ, and to Ø for all other routines. The trap macros automatically set this bit correctly for each routine, so you never have to worry about it yourself.

Notice, however, that the Trap Dispatcher preserves these other registers only on register-based traps. Stack-based traps preserve only those registers required by the Pascal conventions (A2-A6, D3-D7). If you want to preserve any of the other registers, you have to save them yourself before trapping to the routine--typically on the stack with a MOVEM (Move Multiple) instruction--and restore them afterward.

Pascal Interface to the OS and Toolbox

Lisa Pascal doesn't know anything about the Macintosh trap mechanism. When you call an OS or Toolbox routine from Pascal, you're actually calling an interface routine that performs the trap for you. For register-based calls, the interface routine fetches the parameters from the stack where the Pascal calling program left them, puts them in the registers where the routine expects them, then traps to the routine. On return, it moves the routine's result, if any, from a register to the stack and then returns to the calling program. (For routines that return a result code, the interface routine also moves the result code to a global variable, where it can later be accessed with a special Pascal utility routine.) For stack-based calls, there's nothing for the interface routine to do except trap to the routine and then return to the calling program.

Ordinarily this would mean that each stack-based interface routine would be two instructions long: a trap word and an RTS (Return from Subroutine) instruction. However, to save code, the interface routines to the Toolbox dispense with the RTS and instead use the "auto-pop" bit, bit 1Ø of the trap word for Toolbox traps. When this bit is set to 1, the Trap Dispatcher doesn't return control to the interface routine after the trap. Instead, it just removes the trap's return address from the stack and returns directly to the calling program. This halves the amount of memory space taken up by the Toolbox interface routines--from two words per routine to only one, the trap word itself. When you trap to a Toolbox routine from assembly language, the trap macro sets the auto-pop bit to Ø, so that control will return normally.

MIXING PASCAL AND ASSEMBLY LANGUAGE

You can mix Pascal and assembly language freely in your own programs, calling routines written in either language from the other. The Pascal and assembly-language portions of the program have to be compiled and assembled separately, then combined with the Lisa Pascal Linker. For convenience in this discussion, we'll refer to such separately compiled or assembled portions of a program as "modules", although this term isn't actually used in Lisa Pascal. You can divide a program into any number of modules, each of which may be written in either Pascal or assembly language.

References in one module to routines defined in another are called external references. The Linker resolves external references by matching them up with their definitions in other modules. You have to identify all the external references in each module so they can be resolved properly. To call an assembly-language routine from Pascal, you name the routine in a .DEF, .PROC, or .FUNC directive in the module where it's defined and declare it with an EXTERNAL declaration in the Pascal module that refers to it. To call a Pascal routine from assembly language, you declare it in the INTERFACE section of a Pascal unit to make it available to other modules and name it in a .REF directive in the assembly-language module that uses it. The actual process of linking the modules together is covered in the document "Putting Together a Macintosh Application".

All calls from one language to the other, in either direction, must obey Pascal's stack-based calling conventions (see "Calling Toolbox Routines", above). To call a Pascal routine from assembly language, you push the parameters onto the stack before the call and (if the routine is a function) look for the result on the stack on return. In an assembly-language routine to be called from Pascal, you look for the parameters on the stack on entry and leave the result (if any) on the stack before returning.

Under stack-based calling conventions, a convenient way to access a routine's parameters on the stack is with a frame pointer, using the 68ØØØ's LINK and UNLK (Unlink) instructions. You can use any address register for the frame pointer (except A7, which is reserved for the stack pointer), but on the Macintosh register A6 is conventionally used for this purpose. The instruction

```
LINK    A6,#-12
```

at the beginning of a routine saves the previous contents of A6 on the stack and sets A6 to point to them. The second operand specifies the number of bytes of stack space to be reserved for the routine's local variables: in this case, 12 bytes. The LINK instruction offsets the stack pointer by this amount after copying it into A6.

(warning)
The offset is **added** to the stack pointer, not subtracted from it. So to allocate stack space for local variables,

you have to give a **negative** offset; the instruction won't work properly if the offset is positive. Also, to keep the stack pointer correctly aligned, be sure the offset is even. For a routine with no local variables on the stack, use an offset of #Ø.

Register A6 now points to the routine's stack frame; the routine can locate its parameters and local variables by indexing with respect to this register (see Figure 6). The register itself points to its own saved contents, which are often (but needn't necessarily be) the frame pointer of the calling routine. The parameters and return address are found at positive offsets from the frame pointer.

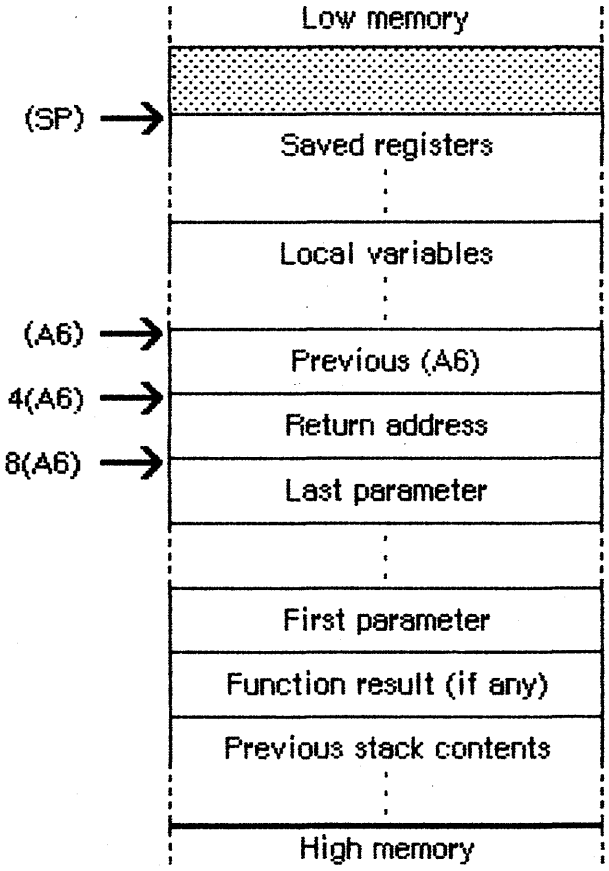


Figure 6. Frame Pointer

Since the saved contents of the frame pointer register occupy a long word (4 bytes) on the stack, the return address is located at 4(A6) and the last parameter at 8(A6). This is followed by the rest of the parameters in reverse order, and finally by the space reserved for the function result, if any. The proper offsets for these remaining parameters and for the function result depend on the number and types of the parameters, according to the table above under "Stack-Based Calls". If the LINK instruction allocated stack space for any local variables, they can be accessed at negative offsets from the frame pointer, again depending on their number and types.

At the end of the routine, the instruction

```
UNLK    A6
```

reverses the process: first it releases the local variables by setting the stack pointer equal to the frame pointer (A6), then pops the saved contents back into register A6. This restores the register to its original state and leaves the stack pointer pointing to the routine's return address.

A routine with no parameters can now just return to the caller with an RTS (Return from Subroutine) instruction. But if there are any parameters, it's the routine's responsibility to "strip" them from the stack before returning. The usual way of doing this is to pop the return address into an address register, increment the stack pointer to remove the parameters, then exit with an indirect jump through the register.

Another point to remember is that any routine that's called from Pascal must observe Pascal register conventions and preserve registers A2-A6 and D3-D7. This is usually done by saving those registers the routine will be using on the stack with a MOVEM (Move Multiple) instruction, then restoring them before returning. Any routine you write that will be accessed via the trap mechanism--for instance, your own version of an OS or Toolbox routine that you've patched into the dispatch table--should observe the same conventions.

Putting all this together, the routine should begin with a sequence like

```
MyRoutine LINK     A6,#-dd              ;set up frame pointer--
                                        ; dd = number of bytes
                                        ; of local variables

          MOVEM.L  A2-A5/D3-D7,-(SP)    ;...or whatever subset of
                                        ; these registers you use
```

and end with something like

```
          MOVEM.L  (SP)+,A2-A5/D3-D7    ;restore registers
          UNLK     A6                   ;restore frame pointer

          MOVE.L   (SP)+,A1             ;save return address in a
                                        ; "trashable" register
          ADD.W    #pp,SP               ;strip parameters--
                                        ; pp = number of bytes
                                        ; of parameters
          JMP      (A1)                 ;return to caller
```

Notice that A6 doesn't have to be included in the MOVEM instructions, since it's saved and restored by the LINK and UNLK.

(warning)

Recall that the Segment Loader, when it starts up an application, sets register A5 to point to the boundary between the application's globals and parameters. Certain parts of the system (notably QuickDraw and the File Manager) rely on finding A5 set up properly--so you have to be a bit more careful about preserving this register. The safest policy is never to touch A5 at all. If you must use it for your own purposes, just saving its contents at the beginning of a routine and restoring them before returning isn't enough: you have to be sure to restore it before any call that might depend on it. The correct setting of A5 is always available in the long-word global variable currentA5.

GLOSSARY

application heap: The portion of the heap available to the running application program for its own memory allocation.

dispatch table: A table in RAM containing the addresses of all Operating System and Toolbox routines in encoded form.

external reference: A reference to a routine or variable defined in a separate compilation or assembly.

frame pointer: A pointer to a routine's stack frame, held in an address register and manipulated with the LINK and UNLK instructions.

heap: The area of memory in which space is dynamically allocated and released on demand, using the Memory Manager.

interface routine: A routine called from Pascal whose purpose is to trap to a certain Operating System or Toolbox routine.

IWM ("Integrated Woz Machine"): The Macintosh's built-in custom disk interface.

parameter block: A table of parameter values to an Operating System routine, stored in memory and located by means of a pointer passed in an address register.

QuickDraw equates file: The file defining global constants and variables pertaining to QuickDraw.

QuickDraw macro file: The file defining trap macros for calling QuickDraw routines.

register-based: Said of an Operating System or Toolbox routine that receives its parameters and returns its results in the processor's registers.

result code: A code returned by an Operating System routine to report successful completion or failure due to some error condition.

SCC (Serial Communications Controller): The Macintosh's built-in 853Ø serial communication interface.

stack: The area of memory in which space is allocated and released in LIFO (last-in-first-out) order, used primarily for routine parameters, return addresses, local variables, and temporary storage.

stack-based: Said of an Operating System or Toolbox routine that receives its parameters and returns its results on the stack.

stack frame: The area of the stack used by a routine for its parameters, return address, local variables, and temporary storage.

system communication area: An area of memory containing global variables used by the Macintosh system software.

system equates file: The file defining global constants and variables pertaining to the Operating System.

system errors file: The file defining all result codes returned by Operating System routines.

system heap: The portion of the heap reserved for use by the Macintosh system software.

system macro file: The file defining trap macros for calling Operating System routines.

Toolbox equates file: The file defining global constants and variables pertaining to the User Interface Toolbox.

Toolbox macro file: The file defining trap macros for calling Toolbox routines.

trap macro: A macro that assembles into a trap word, used for calling an Operating System or Toolbox routine from assembly language.

trap number: The identifying number of an Operating System or Toolbox routine.

trap word: An unimplemented instruction representing a call to an Operating System or Toolbox routine.

unimplemented instruction: An instruction word that doesn't correspond to any valid machine-language instruction but instead causes a trap; used for calling Operating System and Toolbox routines via the 68ØØØ's trap mechanism.

VIA (Versatile Interface Adapter): The Macintosh's built-in 6522 parallel communication interface.

COMMENTS?

Macintosh User Education encourages your comments on this manual.

- What do you like or dislike about it?

- Were you able to find the information you needed?

- Was it complete and accurate?

- Do you have any suggestions for improvement?

Please send your comments to the author (indicated on the cover page) at 10460 Bandley Drive M/S 3-G, Cupertino CA 95014. Mark up a copy of the manual or note your remarks separately. (We'll return your marked-up copy if you like.)

Thanks for your help!

MACINTOSH USER EDUCATION

The Resource Manager: A Programmer's Guide /RMGR/RESOURCE

See Also: Macintosh User Interface Guidelines
Macintosh Operating System Reference Manual
QuickDraw: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Window Manager: A Programmer's Guide
The Font Manager: A Programmer's Guide
The Menu Manager: A Programmer's Guide
Macintosh Control Manager Programmer's Guide
The Dialog Manager: A Programmer's Guide
The Desk Manager: A Programmer's Guide
Putting Together a Macintosh Application

Modification History:			
	First Draft (ROM 2.Ø)	Caroline Rose	2/2/83
	Second Draft (ROM 4)	Caroline Rose	6/21/83
	Third Draft (ROM 7)	Caroline Rose	1Ø/3/83
	Errata added	Caroline Rose	3/8/84

ABSTRACT

Macintosh applications make use of many resources, such as menus, fonts, and icons. These resources are stored in resource files separately from the application code, for flexibility and ease of maintenance. This manual describes resource files and the Resource Manager routines.

Errata:

The low-order bit of the resource attribute byte is no longer available for use by your application; it's now reserved for internal use by the Resource Manager.

There's a new function:

```
FUNCTION SizeResource (theResource: Handle) : INTEGER;
```

Given a handle to a resource, SizeResource returns the size of the resource in bytes. If the resource isn't in memory, the size is read from the resource file. If the given handle isn't a handle to a resource, SizeResource will return -1 and the ResError function will return the error code resNotFound.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Resource Manager, the part of the Macintosh User Interface Toolbox through which an application accesses various resources that it uses, such as menus, fonts, and icons. *** Eventually it will become part of a large manual describing the entire Toolbox. *** It discusses resource files, where resources are stored. Resources form the foundation of every Macintosh application; even the application's code is a resource. In a resource file, the resources used by the application are stored separately from the code for flexibility and ease of maintenance.

- You can use an existing program for creating and editing resource files, or write one of your own. These programs will call Resource Manager routines.

- Usually you'll access resources indirectly through other Toolbox units, such as the Menu Manager and the Font Manager, which in turn call the Resource Manager to do the low-level resource operations. In some cases, you may need to call a Resource Manager file-opening routine and possibly other routines to access resources directly.

(hand)

This manual describes version 7 of the ROM. If you're using a different version, the Resource Manager and the file system may not work as discussed here.

Like all documentation about Toolbox units, this manual assumes you're familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's Memory Manager. You should also be familiar with the following:

- The basic functions of the Finder, which are performed with the help of the Resource Manager. (To the user, the Finder is known as the Desktop Manager.)

- The Operating System error codes.

- The Macintosh file system, as documented *** though probably not up-to-date *** in the Macintosh Operating System Reference Manual. You need to know about this only if you want to understand exactly how resources are implemented internally; you don't have to know it to be able to use the Resource Manager.

If you're going to write your own program to create and edit resource files, you also need to know the exact format of each type of resource. The documentation for the Toolbox unit that deals with a particular type of resource will tell you what you need to know for that resource.

This manual begins with an introduction to the Resource Manager and resources, an overview of resource files, and a discussion of resource specification, all of which offer useful general information. The next

section deals with resource references; you can skip it if you're only going to access resources through other Toolbox units.

Next, a section on using the Resource Manager introduces you to its routines and tells how they fit into the flow of your application. This is followed by detailed descriptions of all Resource Manager procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that will not interest all readers. A discussion of how resources point to each other is followed by a section giving the exact format of a resource file. *** Also, to be removed eventually: notes for programmers who will use the Resource Manager routines from assembly language. ***

Finally, there's a summary of the Resource Manager data structures and routine calls and a summary of the resource file format, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE RESOURCE MANAGER

Macintosh applications make use of many resources, such as menus, fonts, and icons, which are stored in resource files. For example, an icon resides in a resource file as a 32-by-32 bit image, and a font as a large bit image containing the characters of the font. In some cases the resource consists of descriptive information (such as, for a menu, the menu title, the text of each command in the menu, whether the command is checked with a check mark, and so on). The Resource Manager keeps track of resources in resource files and provides routines that allow applications and other Toolbox units to access them.

There's a resource file associated with each application, containing the resources specific to that application; these resources include the application code itself. There's also a system resource file, which contains standard resources shared by all applications (also called system resources).

The resources used by an application are created and changed separately from the application's code. This separation is the main advantage to having resource files. A change in the title of a menu, for example, won't require any recompilation of code, nor will translation to a foreign language.

The Resource Manager is initialized by the system when it starts up, and the system resource file is opened as part of the initialization. Your application's resource file is opened when the application starts up. When instructed to get a certain resource, the Resource Manager normally looks first in the application's resource file and then, if the search isn't successful, in the system resource file. This makes it easy to share resources among applications and also to override a system resource with one of your own (if you want to use something other than a standard icon in an alert box, for example).

You refer to a resource by passing the Resource Manager a resource specification, which consists of a type and either an ID number or a name. Any resource type is valid, whether one of those reserved by the Toolbox (such as for menus and fonts) or a type created for use by your application. Given a resource specification, the Resource Manager will read the resource into memory and return a handle to it.

(eye)

> The Resource Manager knows nothing about the formats of the individual types of resources. Only the routines in other Toolbox units that call the Resource Manager have this knowledge.

While most access to resources is read-only, certain applications may want to modify resources. You can change the content of a resource or its ID number, name, or other attributes--everything except its type. For example, you can designate whether the resource should be kept in memory or whether, as is normal for large resources, it can be removed from memory and read in again when needed. You can change existing resources, remove resources from the resource file altogether, or add new resources to the file.

Not only can an application's resource file contain resources themselves, but it may also contain references to resources in the system resource file. These references need not be in the application's resource file in order for the system resources to be found, because the system resource file will be searched anyway as part of the normal search process; however, the references do serve other purposes. One is that a reference can have a different name than the system resource itself, thus providing an "alias" for the resource. But more important, these references let the Finder know what resources the application uses, thus ensuring that those resources will accompany the application if you should copy it to a disk that has a different system resource file on it. References to system resources can be added or removed with Resource Manager routines.

Resource files are not limited to applications; anything stored in a file can have its own resources. For example, documents usually have resource files containing references to the system resources they use, such as fonts and icons. As in an application's resource file, these references tell the Finder what resources the document uses. An unusual font used in only one document can be included in the resource file for that document rather than in the system resource file.

(hand)

> Although shared resources are usually stored in the system resource file, you can have other resource files that contain resources shared by two or more applications (or documents, or whatever). In this case, however, the Finder will know nothing about the connection between the shared resources and the files that use them.

A number of resource files may be open at one time; the Resource Manager always searches the files in the reverse of the order that they

were opened. Since the system resource file is opened when the Resource Manager is initialized, it's always searched last. Usually the search starts with the most recently opened resource file, but you can change it to start with a file that was opened earlier. (See Figure 1.)

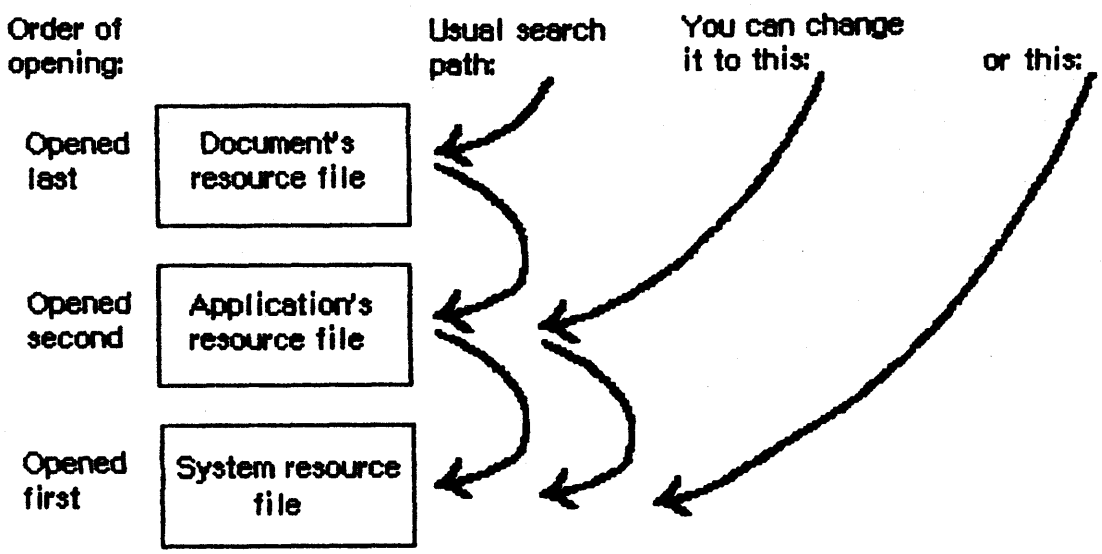


Figure 1. Resource File Searching

OVERVIEW OF RESOURCE FILES

Resources may be put in a resource file with the aid of the Resource Editor, which is documented *** nowhere right now, because it isn't yet available. Meanwhile, you can use the Resource Compiler. You describe the resources in a text file that the Resource Compiler uses to generate the resource file. The exact format of the input file to the Resource Compiler is given in the manual "Putting Together a Macintosh Application". ***

A resource file is not a file in the strictest sense. Although it's functionally like a file in many ways, it's actually just one of two parts, or "forks", of a file. (See Figure 2.) Every file has a resource fork and a data fork (either of which may be empty). The resource fork of an application file contains not only the resources used by the application but also the application code. The code is divided into different segments, each of which is a resource; this allows various parts of the program to be loaded and purged dynamically. The data fork of an application file initially contains nothing, but the application may store data there if desired, by using the Operating System file I/O routines. All data related to resources is stored in the resource fork via the Resource Manager.

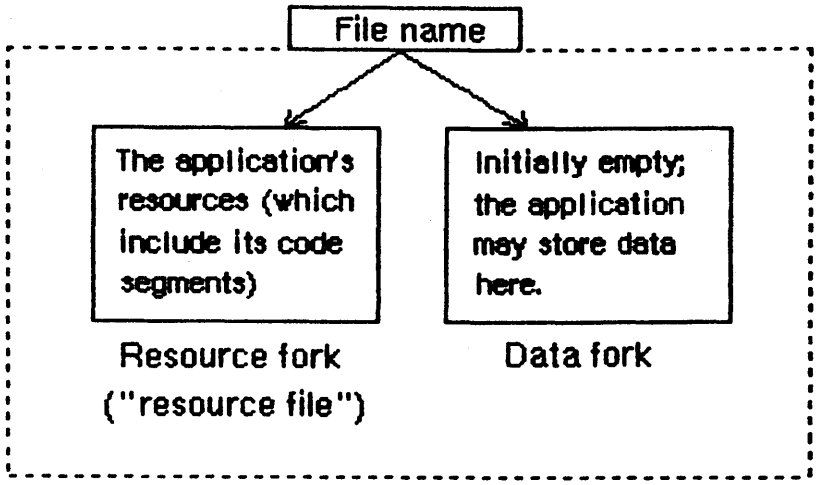


Figure 2. An Application File

As shown in Figure 3, the system resource file has this same structure. The resource fork contains the system resources and the data fork contains the RAM-based Operating System routines. Figure 3 also shows the structure of a file containing a document; the resource fork contains the document's resources and the data fork contains the data that comprises the document.

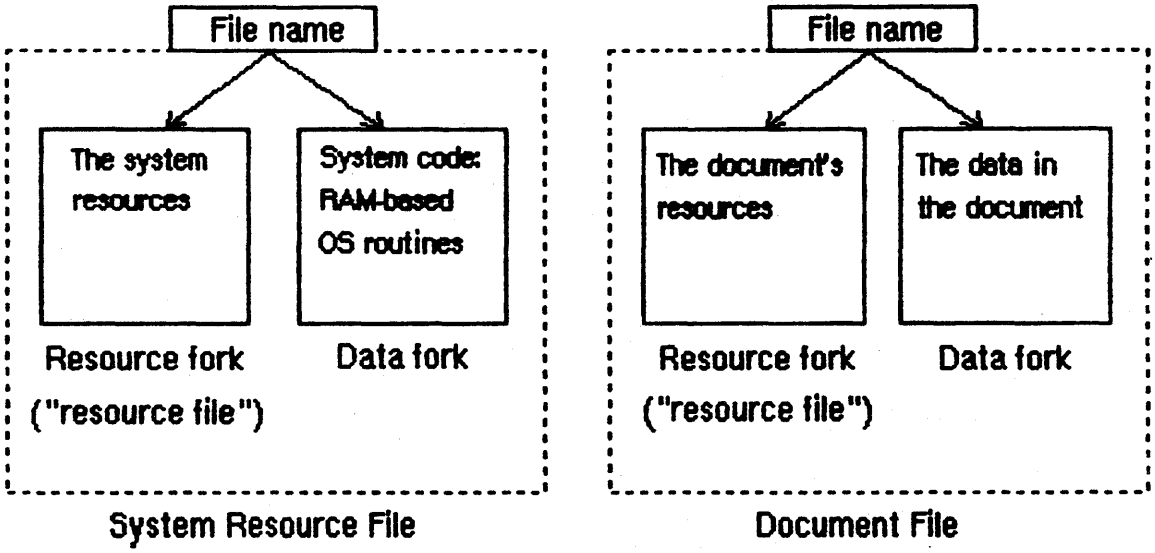


Figure 3. Other Files

To open a resource file, the Resource Manager calls the appropriate Operating System routine and returns the reference number it gets from the Operating System. This is a number greater than Ø by which you can refer to the file when calling other Resource Manager routines. Most of the routines, however, don't have such a parameter; instead, they assume that the current resource file is where they should perform their operation (or begin it, in the case of a search for a resource). The current resource file is the last one that was opened unless you

specify otherwise.

A resource file consists primarily of resource data and a resource map. The resource data consists of the resources themselves (for example, the bit image for an icon or the descriptive information for a menu). The resource map provides the connection between a resource specification and the corresponding resource data. It's like the index of a book; the Resource Manager looks up the resource you specify in the resource map and learns where its resource data is located. The resource map leads to a resource in the same file as the map or provides a reference to a system resource.

The resource map is read into memory when the file is opened and remains there until the file is closed. Notice that although we say the Resource Manager searches resource files, it actually searches the resource maps that were read into memory, and not the resource files on the disk.

Resource data is normally read into memory when needed, though you can specify that it be read in as soon as the resource file is opened. Once read in, the data for a particular resource may or may not be kept in memory, depending on an attribute of that resource that's specified in the resource map. Resources consisting of a relatively large amount of data are usually designated as purgeable, meaning they may be removed from the heap (purged) when space is required by the Memory Manager. Before accessing such a resource through its handle, you can ask the Resource Manager to read the resource into memory again if it was purged.

(hand)

> Programmers concerned about the amount of available memory should be aware that there's a 12-byte overhead in the resource map for every resource and an additional 12-byte overhead for memory management if the resource is read into memory.

To modify a resource, you change the resource data or resource map in memory. The change becomes permanent only at your explicit request, and then only when the application terminates or when you call a routine specifically for updating or closing the resource file.

Each resource file also contains a partial copy of the file's directory entry, written and used by the Finder, and up to 128 bytes of any data the application wishes to store there.

RESOURCE SPECIFICATION

In a resource file, every resource is assigned a type, an ID number, and optionally a name. When calling a Resource Manager routine to access a resource, you specify the resource by passing its type and either its ID number or its name. This section gives some general information about resource specification.

The resource type is a sequence of four characters. Its Pascal data type is:

```
TYPE ResType = PACKED ARRAY [1..4] OF CHAR;
```

The standard resource types recognized by the Macintosh User Interface Toolbox are as follows:

Resource type	Meaning
'CODE'	Application code segment
'WIND'	Window template
'WDEF'	Window definition function
'MENU'	Menu
'MDEF'	Menu definition procedure
'MBAR'	Menu bar
'CNTL'	Control template
'CDEF'	Control definition function
'DLOG'	Dialog template
'ALRT'	Alert template
'DITL'	Item list in a dialog or alert
'ICON'	Icon
'FONT'	Font
'FWID'	Font widths
'CURS'	Cursor
'PICT'	Picture
'PAT '	Pattern (The space is required.)
'PAT#'	Pattern list
'STR '	String (The space is required.)
'DRVR'	Desk accessory or other I/O driver
'KEYC'	Keyboard configuration
'PACK'	Package
'ANYB'	Any bytes

In addition, the type 'DSAT' is reserved for system use.

(eye)
Uppercase and lowercase letters are distinguished in resource types. For example, 'Menu' will not be recognized as the resource type for menus.

Notice that some of the resources listed above are "templates". A template is a list of parameters used to build a Toolbox object; it is not the object itself. For example, a window template contains information specifying the size and location of the window, its title, whether it's visible, and so on. The Window Manager uses this information to build the window in memory and then never accesses the template again.

You can use any four-character sequence (except those listed above) for resource types specific to your application.

Every resource has an ID number, or resource ID. The resource ID must be unique within each resource type, but resources of different types may have the same ID. The standard resources in the system resource

file are usually numbered starting from Ø. The exact range of ID numbers reserved for system resources varies according to resource type. To be safe, if you want the ID numbers of your own resources not to conflict with those of the system resources, you should start numbering from at least 256 (or call a Resource Manager routine that will return an unused resource ID).

(hand)
For assembly-language programmers, the file ResEqu.Text contains predefined constants for the various resource types and for the ID numbers of standard resources.

A resource may optionally have a resource name. Like the resource ID, the resource name must be unique within each type. When comparing resource names, The Resource Manager uses the standard Operating System string comparison routine, which doesn't distinguish between uppercase and lowercase and interprets diacritical marks in foreign names properly.

RESOURCE REFERENCES

The connection between a resource specification and the corresponding resource data is provided by the resource map, via resource references. As illustrated in Figure 4, there are two kinds of resource reference:

- Local references, which are references to resources in this resource file. These point to the resource data in the file and contain a handle to the data if it's in memory.

- System references, which are references to system resources. These provide a resource specification for the resource in the system resource file, which in turn leads to a local reference to the resource in that file.

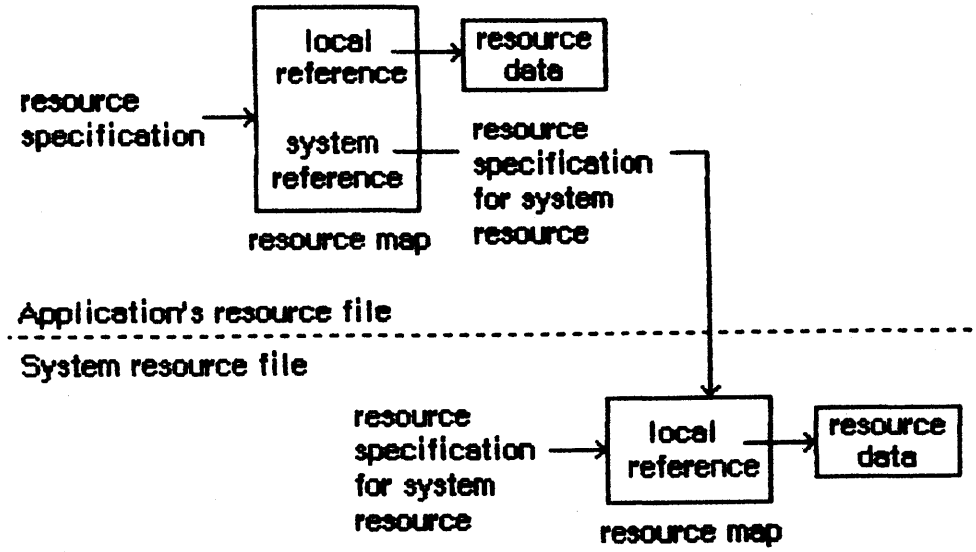


Figure 4. Resource References in Resource Maps

Every resource reference has its own type, ID number, and optional name. In the case of local references, the ID number and name are those of the resource itself. A system reference, on the other hand, may have its own ID number and name, different from those of the resource it refers to in the system resource file.

Suppose you're accessing a resource for the first time. You pass a resource specification to the Resource Manager, which looks for a match among all the references in the resource map; if none is found, it looks at the references in the resource map of the next resource file to be searched. (Remember, it looks in the resource map in memory, not in the file.) Eventually it gets to a local reference to the resource, which tells it where the resource data is in the file. After reading the resource data into memory, the Resource Manager stores a handle to that data in the local reference (again, in the resource map in memory) and returns the handle so you can use it to refer to the resource in subsequent routine calls.

Every resource reference also has certain resource attributes that determine how the resource should be dealt with. In the routine calls for setting or reading them, each attribute is specified by a bit in the low-order byte of a word, as illustrated in Figure 5.

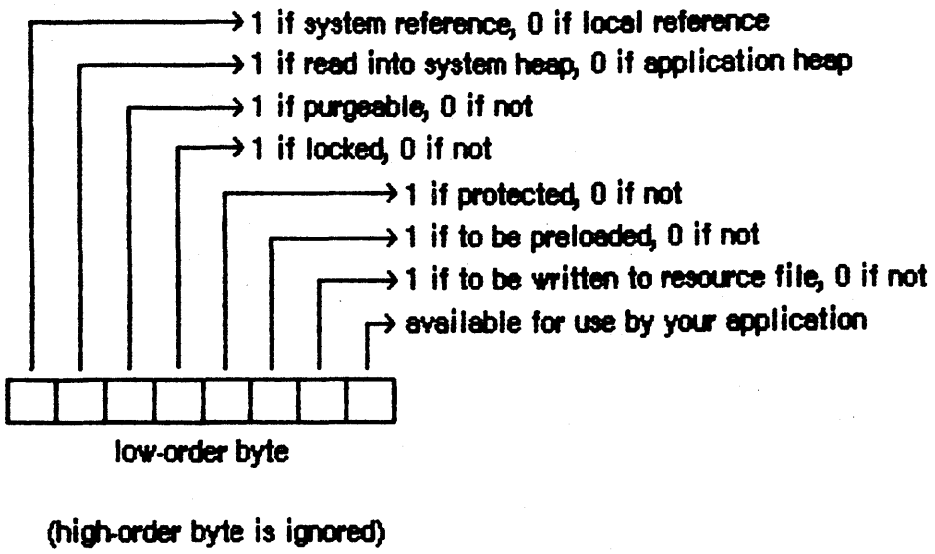


Figure 5. Resource Attributes

The Resource Manager provides a predefined constant for each attribute, in which the bit corresponding to that attribute is set.

```
CONST resSysRef     = 128;  {set if system reference}
      resSysHeap    = 64;   {set if read into system heap}
      resPurgeable  = 32;   {set if purgeable}
      resLocked     = 16;   {set if locked}
      resProtected  = 8;    {set if protected}
      resPreload    = 4;    {set if to be preloaded}
      resChanged    = 2;    {set if to be written to resource file}
      resUser       = 1;    {available for use by your application}
```

(eye)

Your application should not change the setting of the resSysRef attribute, nor should it set the resChanged attribute directly. (ResChanged is set as a side effect of the procedure you call to tell the Resource Manager that you've changed a resource.)

Normally the resSysHeap attribute is set for all system resources; however, if the resource is too large for the system heap, this attribute will be Ø, and the resource will be read into the application heap.

Since a locked resource is neither relocatable nor purgeable, the resLocked attribute overrides the resPurgeable attribute; when resLocked is set, the resource will not be purgeable regardless of whether resPurgeable is set.

If the resProtected attribute is set, the application can't use Resource Manager routines to do any of the following to the resource: set the ID number or name in the resource reference; remove the resource from the resource file; or remove the system reference to it,

if it's a system resource. The routine that sets the resource attributes may be called, however, to remove the protection or just change some of the other attributes.

The resPreload attribute tells the Resource Manager to read this resource into memory immediately after opening the resource file. This is useful, for example, if you immediately want to draw ten icons stored in the file; rather than read and draw each one individually in turn, you can have all of them read in when the file is opened and just draw all ten.

The resChanged attribute is used only while the resource map is in memory, and must be Ø in the resource file. It tells the Resource Manager whether this resource has been changed.

USING THE RESOURCE MANAGER

This section discusses how the Resource Manager routines fit into the general flow of an application program and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

Resource Manager initialization happens automatically when the system starts up: the system resource file is opened and its resource map is read into memory. Your application's resource file is opened when the application starts up; you can call CurResFile to get its reference number. You can also call OpenResFile to open any resource file that you specify by name, and CloseResFile to close any resource file. A function named ResError lets you check for errors that may occur during execution of Resource Manager routines.

(hand)

> These are the only routines you need to know about to use the Resource Manager indirectly through other Toolbox units; you can skip to their descriptions in the next section.

Normally when you want to access a resource for the first time, you'll specify it by type and ID number (or type and name) in a call to GetResource (or GetNamedResource). In special situations, you may want to get every resource of each type. There are two routines which, used together, will tell you all the resource types that are in all open resource files: CountTypes and GetIndType. Similarly, CountResources and GetIndResource may be used to get all resources of a particular type.

If you don't specify otherwise, GetResource, GetNamedResource, and GetIndResource read the resource data into memory and return a handle to it. Sometimes, however, you may not need the data to be in memory. You can use a procedure named SetResLoad to tell the Resource Manager not to read the resource data into memory when you get a resource; in this case, the handle returned for the resource will be an empty handle

Many register-based routines return a 16-bit result code in the low-order half of register DØ to report successful completion or failure due to some error condition. A negative result code always signals an error of some kind; a code of Ø denotes successful completion. (Some routines also use DØ to return an actual data result. In these cases, any nonnegative value in the low-order half of the register represents a true result and implies successful completion of the routine.) The system errors file defines symbolic names for all result codes reported by the various OS routines.

Just before returning from a register-based call, the Trap Dispatcher tests the low-order half of DØ with a TST.W instruction to set the processor's condition codes. You can then check for an error by branching directly on the condition codes, without any explicit test of your own: for example,

```
_PurgeMem               ;trap to routine
BMI     Error           ;branch on error

. . .                   ;no error--actual result
                        ; in low half of DØ
```

(warning)

Not all register-based routines return a result code. Some leave the contents of DØ unchanged; others use the full 32 bits of the register to return a long-word result. See the documentation of individual routines for details.

Stack-Based Calls

To call a stack-based routine from assembly language, you have to set up the parameters on the stack in the same way the compiled object code would if your program were written in Pascal. The number and types of parameters expected on the stack depend on the routine being called. The number of bytes each parameter occupies depends on its type:

- CloseResFile, which updates the resource file before closing it.

- UpdateResFile, which simply updates the resource file.

- WriteResource, which writes the resource data for a specified resource to the resource file.

RESOURCE MANAGER ROUTINES

This section describes all the Resource Manager procedures and functions. They are presented in their Pascal form; for information on using them from assembly language, see "Using the Toolbox from Assembly Language" *** for now, see "Using QuickDraw from Assembly Language" in the QuickDraw manual and also "Notes For Assembly-Language Programmers" in this manual ***.

(hand)

Assembly-language programmers: Except for LoadResource, all Resource Manager routines preserve all registers except AØ and DØ. LoadResource preserves AØ and DØ as well.

Initializing the Resource Manager

Although you don't call these initialization routines (because they're executed automatically for you), it's a good idea to familiarize yourself with what they do.

```
FUNCTION InitResources : INTEGER;
```

InitResources is called by the system when it starts up, and should not be called by the application. It initializes the Resource Manager, opens the system resource file, reads the resource map from the file into memory, and returns a reference number for the file.

(hand)

The application doesn't need the reference number for the system resource file, because every Resource Manager routine that has a reference number as a parameter interprets Ø to mean the system resource file.

```
PROCEDURE RsrcZoneInit;
```

RsrcZoneInit is called automatically when your application starts up, to initialize the resource map read from the system resource file; normally you'll have no need to call it directly. It "cleans up" after any resource access that may have been done by a previous application. First it closes all open resource files except the system resource file. Then, for every system resource that was read into the

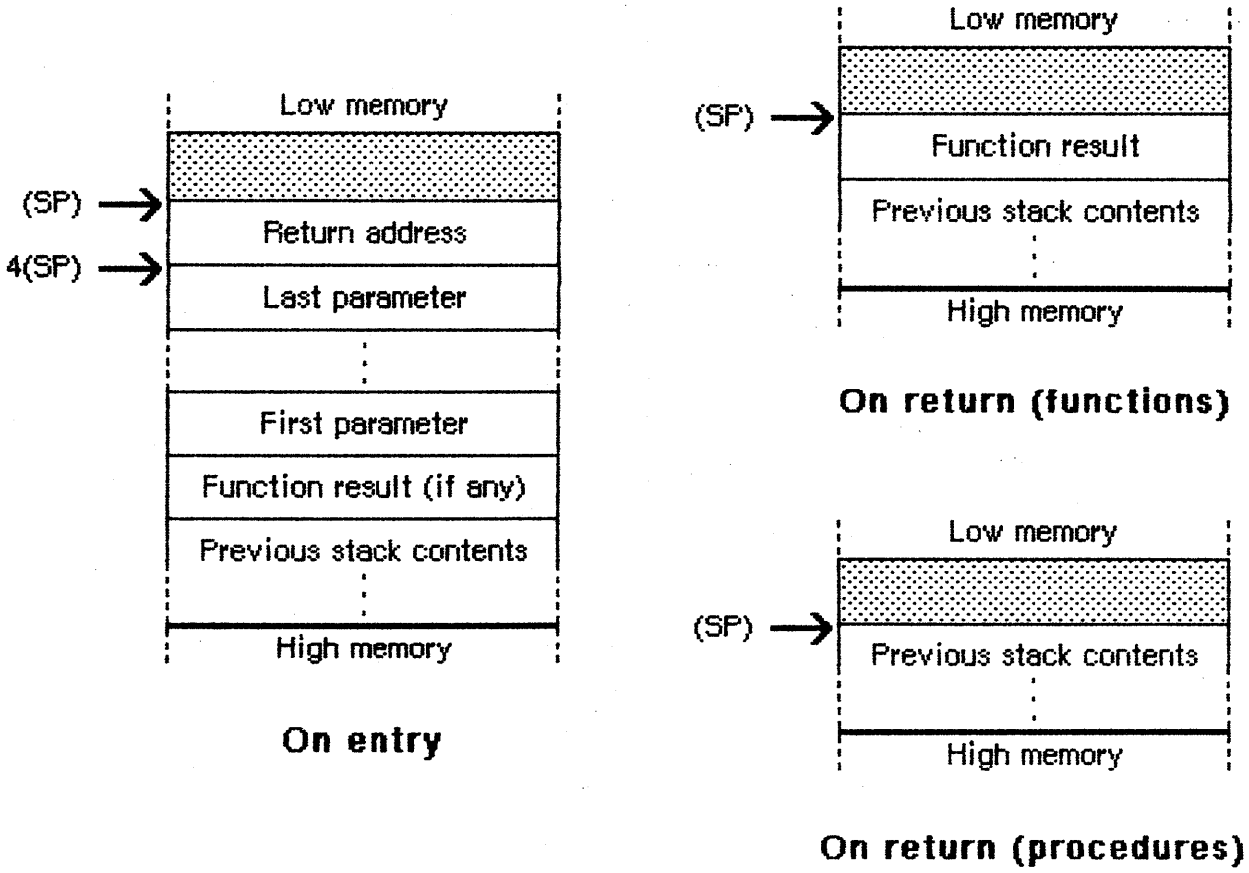


Figure 5. Stack Format for Stack-Based Calls

For example, the Window Manager function GrowWindow is defined in Pascal as follows:

```
FUNCTION GrowWindow (theWindow: WindowPtr; startPt: Point;
            sizeRect: Rect) : LongInt;
```

To call this function from assembly language, you'd write something like the following:

```
SUBQ.L  #4,SP               ;make room for LongInt result
MOVE.L  theWindow,-(SP)     ;push window pointer
MOVE.L  startPt,-(SP)       ;a Point is a 4-byte record,
                            ;  so push actual contents
PEA     sizeRect            ;a Rect is an 8-byte record,
                            ;  so push a pointer to it
_GrowWindow                 ;trap to routine
MOVE.L  (SP)+,D3            ;pop result from stack
```

- Updates the resource file by calling the UpdateResFile procedure
- For each resource in the resource file, deallocates the memory it occupies by calling the ReleaseResource procedure
- Deallocates the memory occupied by the resource map
- Closes the resource file

If there's no resource file open with the given reference number, CloseResFile will do nothing and the ResError function will return the error code resFNotFound. A refNum of Ø represents the system resource file, but if you ask to close this file, CloseResFile first closes all other open resource files.

A CloseResFile of every open resource file except the system resource file is done automatically when the application terminates. So you only need to call CloseResFile if you want to close the system resource file, or if you want to close any resource file before the application terminates.

Checking for Errors

FUNCTION ResError : INTEGER;

Called after one of the various Resource Manager routines that may result in an error condition, ResError identifies the error or returns Ø if no error occurred. If an error occurred at the Operating System level, it returns one of the Operating System error codes, such as those for file I/O errors and the Memory Manager "out of memory" error. (See the Macintosh Operating System Reference Manual for the exact codes.) If an error happened at the Resource Manager level, ResError returns one of the following predefined error codes:

```
CONST resNotFound  = -192;   {resource not found}
      resFNotFound = -193;   {resource file not found}
      addResFailed = -194;   {AddResource failed}
      addRefFailed = -195;   {AddReference failed}
      rmvResFailed = -196;   {RmveResource failed}
      rmvRefFailed = -197;   {RmveReference failed}
```

Each routine description tells which errors may occur for that routine. You can also check for an error after system startup, which calls InitResources, and application startup, which opens the application's resource file.

(hand)

Assembly-language programmers can access the current value of ResError through the global variable resErr.

Setting the Current Resource File

FUNCTION CurResFile : INTEGER

CurResFile returns the reference number of the current resource file. You can call it when the application starts up to get the reference number of its resource file.

(hand)
Assembly-language programmers can access the reference number of the current resource file through the global variable curMap.

FUNCTION HomeResFile (theResource: Handle) : INTEGER;

Given a handle to a resource, HomeResFile returns the reference number of the resource file containing that resource. If the given handle isn't a handle to a resource, HomeResFile will return -1 and the ResError function will return the error code resNotFound.

PROCEDURE UseResFile (refNum: INTEGER);

Given the reference number of a resource file, UseResFile sets the current resource file to that file. If there's no resource file open with the given reference number, UseResFile will do nothing and the ResError function will return the error code resFNotFound. A refNum of Ø represents the system resource file.

This procedure is useful for changing which resource file is searched first. For example, if you no longer want to override a system resource with one by the same name in your application's resource file, you can call UseResFile(Ø) to make the search begin (and end) in the system resource file.

Getting Resource Types

FUNCTION CountTypes : INTEGER;

CountTypes returns the number of resource types in all open resource files.

PROCEDURE GetIndType (VAR theType: ResType; index: INTEGER);

Given an index ranging from 1 to CountTypes (above), GetIndType returns a resource type in theType. Called repeatedly over the entire range

for the index, it returns all the resource types in all open resource files. If the given index isn't in the range from 1 to CountTypes, GetIndType returns four NUL characters (ASCII code Ø).

Getting and Disposing of Resources

```
PROCEDURE SetResLoad (load: BOOLEAN);
```

Normally, the routines that return handles to resources read the resource data into memory if it's not already in memory. SetResLoad(FALSE) affects all those routines so that they will not read the resource data into memory and will return an empty handle. Resources whose resPreload attribute is set will still be read in, however, when a resource file is opened. SetResLoad(TRUE) restores the normal state.

(eye)

If you call SetResLoad(FALSE), be sure to restore the normal state as soon as possible, because other Toolbox units that call the Resource Manager rely on it.

(hand)

Assembly-language programmers can access the current SetResLoad state (TRUE or FALSE) through the global variable resLoad.

```
FUNCTION CountResources (theType: ResType) : INTEGER;
```

CountResources returns the total number of resources of the given type in all open resource files.

```
FUNCTION GetIndResource (theType: ResType; index: INTEGER) : Handle;
```

Given an index ranging from 1 to CountResources(theType), GetIndResource returns a handle to a resource of the given type (see CountResources, above). Called repeatedly over the entire range for the index, it returns handles to all resources of the given type in all open resource files. GetIndResource reads the resource data into memory if it's not already in memory, unless you've called SetResLoad(FALSE).

(eye)

The handle returned will be an empty handle if you've called SetResLoad(FALSE), or will become empty if the resource data for a purgeable resource is read in but later purged. (You can test for an empty handle with, for example, myHndl^ = NIL.) To read in the data and make the handle no longer be empty, you can call LoadResource.

GetIndResource returns handles for all resources in the most recently opened resource file first, and then for those in the resource files opened before it, in the reverse of the order that they were opened. If you want to find out how many resources of a given type are in a particular resource file, you can do so as follows: Call GetIndResource repeatedly with the index ranging from 1 to the number of resources of that type. Pass each handle returned by GetIndResource to HomeResFile and count all occurrences where the reference number returned is that of the desired file. Be sure to start the index from 1, and to call SetResLoad(FALSE) so the resources won't be read in.

(hand)

The UseResFile procedure affects which file the Resource Manager searches first when looking for a particular resource but not when getting indexed resources with GetIndResource.

If the given index isn't in the range from 1 to CountResources(theType), GetIndResource returns NIL. It also returns NIL if the resource is to be read into memory but won't fit; in this case, the ResError function will return an appropriate Operating System error code.

```
FUNCTION GetResource (theType: ResType; theID: INTEGER) : Handle;
```

GetResource returns a handle to the resource having the given type and ID number, reading the resource data into memory if it's not already in memory and if you haven't called SetResLoad(FALSE) (see the first note above for GetIndResource). GetResource looks in the current resource file and all resource files opened before it, in the reverse of the order that they were opened; the system resource file is searched last. If it doesn't find the resource, GetResource returns NIL. It also returns NIL if the resource is to be read into memory but won't fit; in this case, the ResError function will return an appropriate Operating System error code.

```
FUNCTION GetNamedResource (theType: ResType; name: Str255) : Handle;
```

GetNamedResource is the same as GetResource (above) except that you pass a resource name instead of an ID number.

```
PROCEDURE LoadResource (theResource: Handle);
```

Given a handle to a resource (returned by GetIndResource, GetResource, or GetNamedResource), LoadResource reads that resource into memory. It does nothing if the resource is already in memory or if the given handle isn't a handle to a resource; in the latter case, the ResError function will return the error code resNotFound. Call this procedure if you want to access the data for a resource through its handle and either you've called SetResLoad(FALSE) or the resource is purgeable.

If you've changed the resource data for a purgeable resource and the resource is purged before being written to the resource file, the changes will be lost; LoadResource will reread the original resource from the resource file. See the descriptions of ChangedResource and SetResPurge for information about how to ensure that changes made to purgeable resources will be written to the resource file.

(hand)
Assembly-language programmers: LoadResource preserves all registers.

```
PROCEDURE ReleaseResource (theResource: Handle);
```

Given a handle to a resource, ReleaseResource deallocates the memory occupied by the resource data, if any, and replaces the handle to that resource in the resource map with NIL. (See Figure 6.) The given handle will no longer be recognized as a handle to a resource; if the Resource Manager is subsequently called to get the released resource, a new handle will be allocated. Use this procedure only after you're completely through with a resource.

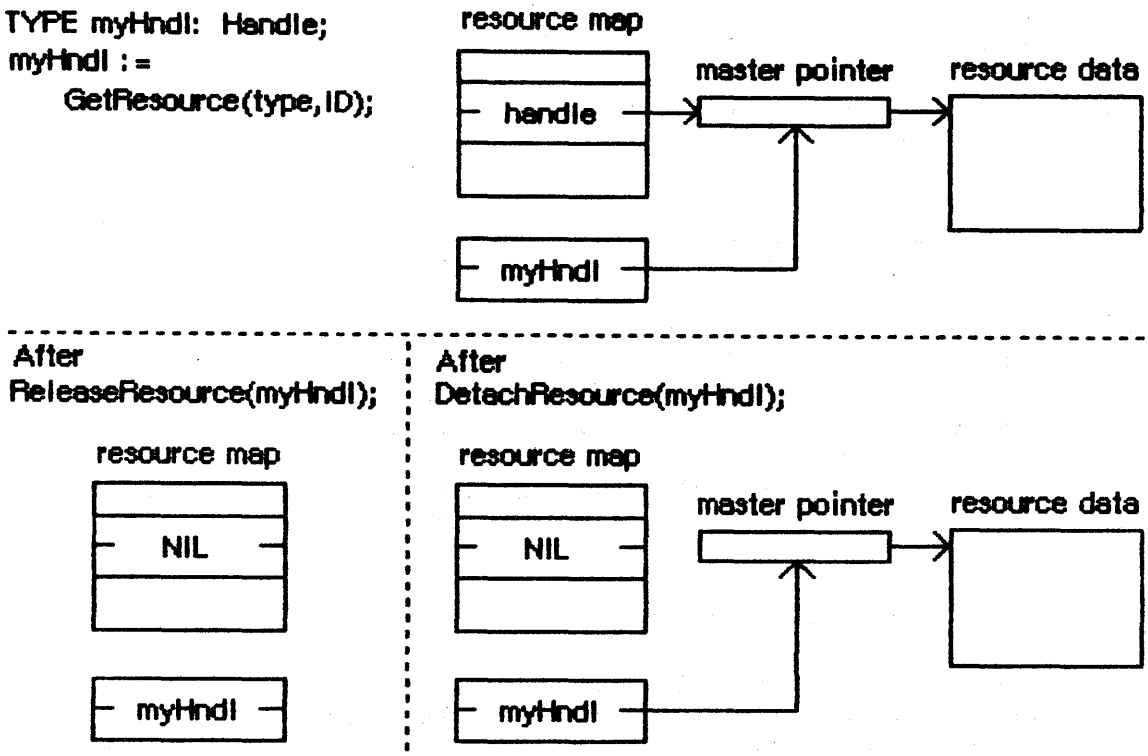


Figure 6. ReleaseResource and DetachResource

If the given handle isn't a handle to a resource, ReleaseResource will do nothing and the ResError function will return the error code resNotFound.

```
PROCEDURE DetachResource (theResource: Handle);
```

Given a handle to a resource, DetachResource replaces the handle to that resource in the resource map with NIL. (See Figure 6.) The given handle will no longer be recognized as a handle to a resource; if the Resource Manager is subsequently called to get the detached resource, a new handle will be allocated. DetachResource is useful if you want the resource data to be accessed only by yourself through the given handle and not by the Resource Manager. It's also useful in the unusual case that you don't want a resource to be deallocated when a resource file is closed.

If the given handle isn't a handle to a resource, DetachResource will do nothing and the ResError function will return the error code resNotFound.

Getting Resource Information

```
FUNCTION UniqueID (theType: ResType) : INTEGER;
```

UniqueID returns an ID number greater than Ø that isn't currently assigned to any resource of the given type in any open resource file. Using this number when you add a new resource to a resource file ensures that it won't override an existing resource.

```
PROCEDURE GetResInfo (theResource: Handle; VAR theID: INTEGER; VAR
          theType: ResType; VAR name: Str255);
```

Given a handle to a resource, GetResInfo returns the ID number, type, and name of the resource. If the current resource file contains a system reference to the resource, it returns the ID number, type, and name of the system reference, which may be different from those of the resource itself in the system resource file. If the given handle isn't a handle to a resource, GetResInfo will do nothing and the ResError function will return the error code resNotFound.

```
FUNCTION GetResAttrs (theResource: Handle) : INTEGER;
```

Given a handle to a resource, GetResAttrs returns the resource attributes for the resource. (Resource attributes are described earlier under "Resource References".) If the current resource file contains a system reference to the resource, GetResAttrs returns the attributes of the system reference, which may be different from those of the resource itself in the system resource file. If the given handle isn't a handle to a resource, GetResAttrs will do nothing and the ResError function will return the error code resNotFound.

Modifying Resources

Except for UpdateResFile and WriteResource, all the routines described below change the resource map in memory and not the resource file itself.

```
PROCEDURE SetResInfo (theResource: Handle; theID: INTEGER; name:
          Str255);
```

Given a handle to a resource, SetResInfo sets the ID number and name of the resource to the given ID number and name. If the current resource file contains a system reference to the resource, SetResInfo sets only the ID number and name of the system reference.

(hand)

Assembly-language programmers: If you pass NIL for the name parameter, the name will not be changed.

(eye)

If the resource is a system resource but the current resource file doesn't contain a reference to it, SetResInfo will set the ID number and name in the system resource file itself. This is a dangerous practice, because other applications may already access the resource and may not work properly if the ID number or name is changed.

The change will be written to the resource file when the file is updated if you follow SetResInfo with a call to ChangedResource.

(eye)

Even if you don't call ChangedResource for this resource, the change may be written to the resource file when the file is updated. If you've **ever** called ChangedResource for **any** resource in the file, or if you've added or removed a resource or a resource reference, the Resource Manager will write out the entire resource map when it updates the file, so all changes made to resource information in the map will become permanent. If you want any of the changes to be temporary, you'll have to restore the original information before the file is updated.

SetResInfo does nothing in the following cases:

- The resProtected attribute for the resource is set.
- The given handle isn't a handle to a resource. The ResError function will return the error code resNotFound.
- The resource map becomes too large to fit in memory (which can happen if a name is passed) or sufficient space for the modified

resource file can't be reserved on the disk. ResError will return an appropriate Operating System error code.

```
PROCEDURE SetResAttrs (theResource: Handle; attrs: INTEGER);
```

Given a handle to a resource, SetResAttrs sets the resource attributes for the resource to attrs. (Resource attributes are described earlier under "Resource References".) If the current resource file contains a system reference to the resource, SetResAttrs sets only the attributes of the system reference. The resProtected attribute takes effect immediately; the others take effect the next time the resource is read in.

(eye)

Do not use SetResAttrs to set the resChanged attribute; you must call ChangedResource instead. Be sure that the attrs parameter passed to SetResAttrs doesn't change the current setting of this attribute.

The attributes set with SetResAttrs will be written to the resource file when the file is updated if you follow SetResAttrs with a call to ChangedResource. However, even if you don't call ChangedResource for this resource, the change may be written to the resource file when the file is updated. See the last warning for SetResInfo (above).

If the given handle isn't a handle to a resource, SetResAttrs will do nothing and the ResError function will return the error code resNotFound.

```
PROCEDURE ChangedResource (theResource: Handle);
```

Call ChangedResource after changing either the information about a resource in the resource map (as described above under SetResInfo and SetResAttrs) or the resource data for a resource, if you want the change to be permanent. Given a handle to a resource, ChangedResource sets the resChanged attribute for the resource. This attribute tells the Resource Manager to do **both** of the following:

- Write the resource data for the resource to the resource file when the file is updated or when WriteResource is called
- Write the entire resource map to the resource file when the file is updated

(eye)

If you change information in the resource map with SetResInfo or SetResAttrs and then call ChangedResource, remember that not only the resource map but also the resource data will be written out when the resource file is updated.

To change the resource data for a purgeable resource and make the change permanent, you have to take special precautions to ensure that the resource won't be purged while you're changing it. You can make the resource temporarily unpurgeable and then write it out with WriteResource before making it purgeable again. You have to use the Memory Manager routines HNoPurge and HPurge to make the resource unpurgeable and purgeable; SetResAttrs can't be used because it won't take effect immediately. For example:

```
myHndl := GetResource(type,ID);     {or LoadResource(myHndl) if  }
                                    { you've gotten it previously}
HNoPurge(myHndl);                   {make it unpurgeable}
  . . .                             {make the changes here}
  ChangedResource(myHndl);          {mark it changed}
  WriteResource(myHndl);            {write it out}
HPurge(myHndl);                     {make it purgeable again}
```

Or, instead of calling WriteResource to write the data out immediately, you can call SetResPurge(TRUE) before making any changes to purgeable resource data.

ChangedResource does nothing in the following cases:

- The given handle isn't a handle to a resource. The ResError function will return the error code resNotFound.

- Sufficient space for the modified resource file can't be reserved on the disk. ResError will return an appropriate Operating System error code.

```
PROCEDURE AddResource (theData: Handle; theType: ResType; theID:
          INTEGER; name: Str255);
```

Given a handle to data in memory (not a handle to an existing resource), AddResource adds to the current resource file a local reference that points to the data. It sets the resChanged attribute for the resource, so the data will be written to the resource file when the file is updated or when WriteResource is called. If the given handle is empty, zero-length resource data will be written. AddResource does nothing in the following cases:

- The given handle is NIL or is already a handle to an existing resource. The ResError function will return the error code addResFailed.

- The resource map becomes too large to fit in memory or sufficient space for the modified resource file can't be reserved on the disk. ResError will return an appropriate Operating System error code.

```
PROCEDURE RmveResource (theResource: Handle);
```

Given a handle to a resource in the current resource file, RmveResource removes the local reference to the resource. The resource data will be removed from the resource file when the file is updated.

(hand)

It doesn't deallocate the memory occupied by the resource data; to do that, call the Memory Manager routine DisposeHandle after calling RmveResource.

If the resProtected attribute for the resource is set or if the given handle isn't a handle to a resource in the current resource file, RmveResource will do nothing and the ResError function will return the error code rmvResFailed.

(eye)

It's dangerous to remove a resource from the system resource file, because all system references to it will become invalid.

```
PROCEDURE AddReference (theResource: Handle; theID: INTEGER; name:
          Str255);
```

Given a handle to a system resource, AddReference adds to the current resource file a system reference to the resource, giving it the ID number and name specified by the parameters. It sets the resChanged attribute for the resource, so the reference will be written to the resource file when the file is updated. AddReference does nothing in the following cases:

- The current resource file is the system resource file or already contains a system reference to the specified resource, or the given handle isn't a handle to a system resource. The ResError function will return the error code addRefFailed.

- The resource map becomes too large to fit in memory or sufficient space for the modified resource file can't be reserved on the disk. ResError will return an appropriate Operating System error code.

```
PROCEDURE RmveReference (theResource: Handle);
```

Given a handle to a system resource, RmveReference removes the system reference to the resource from the current resource file. (The reference will be removed from the resource file when the file is updated.) In the following cases, RmveReference will do nothing and the ResError function will return the error code rmvRefFailed: the resProtected attribute for the resource is set; there's no system reference to the resource in the current resource file; or the given handle isn't a handle to a system resource.

```
PROCEDURE UpdateResFile (refNum: INTEGER);
```

Given the reference number of a resource file, UpdateResFile does the following:

- Changes, adds, or removes resource data in the file as appropriate to match the map. Remember that changed resource data is written out only if you called ChangedResource. If a resource whose data is to be written out has been purged, zero-length resource data will be written.

- Compacts the resource file if necessary, closing up any empty space created when a resource or a resource reference was removed or when a resource was made larger. (If the size of a changed resource is greater than its original size in the resource file, it's written at the end of the file rather than at its original location, leaving empty space at that location. UpdateResFile doesn't close up any empty space created when a resource is made smaller.)

- Writes out the resource map of the resource file, if you ever called ChangedResource for any resource in the file or if you added or removed a resource or a resource reference. All changes to resource information in the map will become permanent as a result of this, so if you want any such changes to be temporary, you must restore the original information before calling UpdateResFile.

If there's no open resource file with the given reference number, UpdateResFile will do nothing and the ResError function will return the error code resFNotFound. A refNum of Ø represents the system resource file.

The CloseResFile procedure calls UpdateResFile before it closes the resource file, so you only need to call UpdateResFile yourself if you want to update the file without closing it.

```
PROCEDURE WriteResource (theResource: Handle);
```

Given a handle to a resource, WriteResource checks the resChanged attribute for that resource and, if it's set (which it will be if you called ChangedResource or AddResource), writes its resource data to the resource file and clears its resChanged attribute. If the resource is purgeable and has been purged, zero-length resource data will be written. WriteResource does nothing if the resProtected attribute for the resource is set or if the given handle isn't a handle to a resource; in the latter case, the ResError function will return the error code resNotFound.

Since the resource file is updated when the application terminates or when you call UpdateResFile (or CloseResFile, which calls UpdateResFile), you only need to call WriteResource if you want to write out just one or a few resources immediately.

```
PROCEDURE SetResPurge (install: BOOLEAN);
```

SetResPurge(TRUE) sets a "hook" in the Memory Manager such that before purging data specified by a handle, the Memory Manager will first pass the handle to the Resource Manager. The Resource Manager will determine whether the handle is that of a resource in the application heap and, if so, will call WriteResource to write the resource data for that resource to the resource file if its resChanged attribute is set (see ChangedResource and WriteResource above). SetResPurge(FALSE) restores the normal state, clearing the hook so that the Memory Manager will once again purge without checking with the Resource Manager.

SetResPurge(TRUE) is useful in applications that modify purgeable resources. You still have to make the resources temporarily unpurgeable while making the changes, as shown in the description of ChangedResource, but you can set the purge hook instead of writing the data out immediately with WriteResource. Notice that you won't know exactly when the resources are being written out; most applications will want more control than this. If you wish, you can set your own such hook.

Advanced Routines

The routines described below allow advanced programmers to have even greater control over resource file operations. Just as individual resources have attributes, an entire resource file also has attributes, which these routines manipulate. Like the attributes of individual resources, resource file attributes are specified by bits in the low-order byte of a word. The Resource Manager provides a predefined constant for each attribute, in which the bit corresponding to that attribute is set.

```
CONST mapReadOnly = 128;
      mapCompact  = 64;
      mapChanged  = 32;
```

When the mapReadOnly attribute is set, the Resource Manager will neither write anything to the resource file nor check whether there's sufficient space for the file on the disk when the resource map is modified.

(eye)

> If you set mapReadOnly but then later clear it, the resource file will be written even if there's no room for it on the disk. This would destroy the file.

The mapCompact attribute causes resource file compaction to occur when the file is updated. It's set by the Resource Manager when a resource or a resource reference is removed, or when a resource is made larger and thus has to be written at the end of the resource file. You may want to set mapCompact to force compaction when you've only made resources smaller.

The mapChanged attribute causes the resource map to be written to the resource file when the file is updated. It's set by the Resource Manager when you call ChangedResource or when you add or remove a resource or a resource reference. You can set mapChanged if, for example, you've changed resource attributes only and don't want to call ChangedResource because you don't want the resource data to be written out.

```
FUNCTION GetResFileAttrs (refNum: INTEGER) : INTEGER;
```

Given the reference number of a resource file, GetResFileAttrs returns the resource file attributes for the file. If there's no resource file with the given reference number, GetResFileAttrs will do nothing and the ResError function will return the error code resFNotFound. A refNum of Ø represents the system reference file.

```
PROCEDURE SetResFileAttrs (refNum: INTEGER; attrs: INTEGER);
```

Given the reference number of a resource file, SetResFileAttrs sets the resource file attributes of the file to attrs. If there's no resource file with the given reference number, SetResFileAttrs will do nothing and the ResError function will return the error code resFNotFound. A refNum of Ø represents the system reference file, but you shouldn't change its resource file attributes.

RESOURCES WITHIN RESOURCES

Resources may point to other resources; this section discusses how this is normally done, for programmers who are interested in background information about resources or who are defining their own resource types.

In a resource file, one resource points to another with the ID number of the other resource. For example, the resource data for a menu includes the ID number of the menu's definition procedure (a separate resource that determines how the menu looks and behaves). To work with the resource data in memory, however, it's faster and more convenient to have a handle to the other resource rather than its ID number. Since a handle occupies two words, the ID number in the resource file is followed by a word containing Ø; these two words together serve as a placeholder for the handle. Once the other resource has been read into memory, these two words can be replaced by a handle to it. (See Figure 7.)

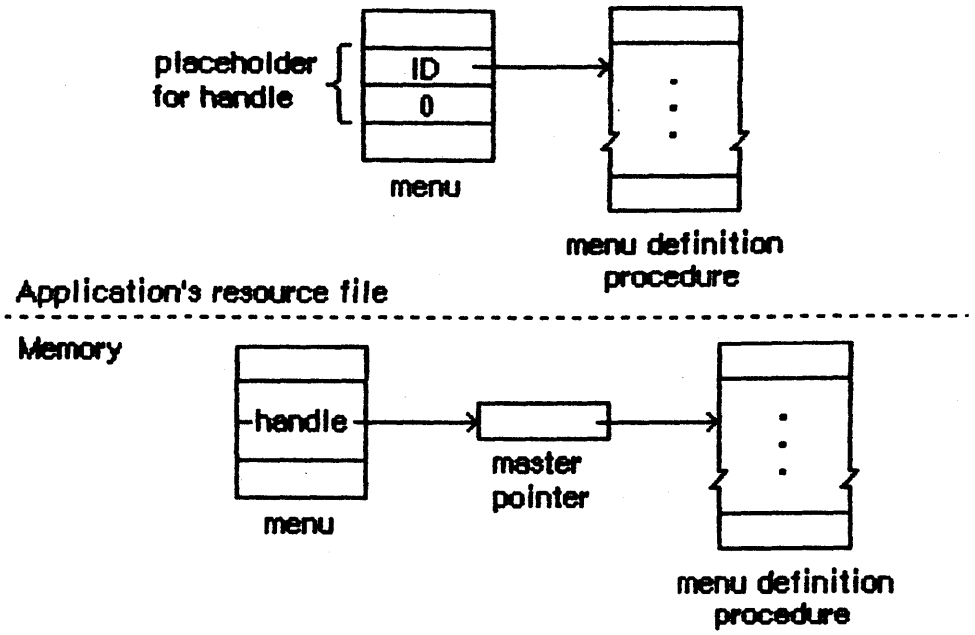


Figure 7. How Resources Point to Resources

(hand)

The practice of using the ID number followed by Ø as a placeholder is simply a convention. If you like, you can set up your own resources to have the ID number followed by a dummy word, or even a word of useful information, or you can put the ID in the second rather than the first word of the placeholder.

In the case of menus, the Menu Manager routine GetMenu calls the Resource Manager to read the menu and the menu definition procedure into memory, and then replaces the placeholder in the menu with the handle to the procedure. There may be other cases where you call the Resource Manager directly and store the handle in the placeholder yourself. It might be useful in these cases to call HomeResFile to learn which resource file the original resource is located in, and then, before getting the resource it points to, call UseResFile to set the current resource file to that file. This will ensure that the resource pointed to is read from that same file (rather than one that was opened after it).

(eye)

If you modify a resource that points to another resource and you make the change permanent by calling ChangedResource, be sure you reverse the process described here, restoring the other resource's ID number in the placeholder.

FORMAT OF A RESOURCE FILE

This section gives the exact format of a resource file, which you need to know if you're writing a program that will create or modify resource files directly. You don't have to know the exact format to be able to use the Resource Manager routines.

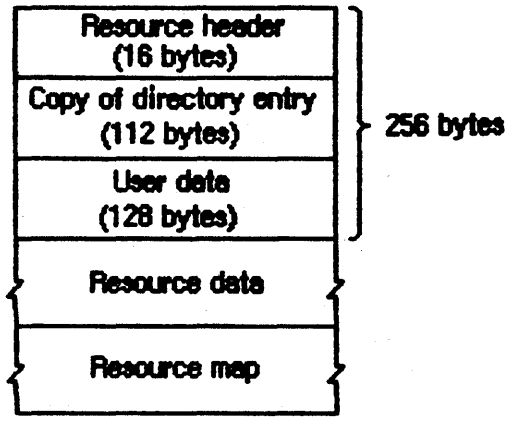


Figure 8. Format of a Resource File

As illustrated in Figure 8, every resource file begins with a resource header. The resource header gives the offsets to and lengths of the resource data and resource map parts of the file, as follows:

Number of bytes	Contents
4 bytes	Offset from beginning of resource file to resource data
4 bytes	Offset from beginning of resource file to resource map
4 bytes	Length of resource data
4 bytes	Length of resource map

(hand)
All offsets and lengths in the resource file are given in bytes.

This is what immediately follows the resource header:

Number of bytes	Contents
112 bytes	Partial copy of directory entry for this file
128 bytes	Available for user data

The directory copy is used by the Finder. The user data may be whatever the you want.

The resource data follows the user data. It consists of the following for each resource in the file:

Number of bytes	Contents
For each resource:	
4 bytes	Length of following resource data
n bytes	Resource data for this resource

To learn exactly what the resource data is for a standard type of resource, see the documentation on the Toolbox unit that deals with that resource type.

After the resource data, the resource map begins as follows:

Number of bytes	Contents
16 bytes	Ø (reserved for copy of resource header)
4 bytes	Ø (reserved for handle to next resource map to be searched)
2 bytes	Ø (reserved for file reference number)
2 bytes	Resource file attributes
2 bytes	Offset from beginning of resource map to type list (see below)
2 bytes	Offset from beginning of resource map to resource name list (see below)

After reading the resource map into memory, the Resource Manager stores the indicated information in the reserved areas at the beginning of the map.

The resource map continues with a type list, reference lists, and a resource name list. The type list contains the following:

Number of bytes	Contents
2 bytes	Number of resource types in the map minus 1
For each type:	
4 bytes	Resource type
2 bytes	Number of resources of this type in the map minus 1
2 bytes	Offset from beginning of type list to reference list for resources of this type

This is followed by the reference list for each type of resource, which contains the resource references for all resources of that type. The reference lists are contiguous and in the same order as the types in the type list. The format of a reference list is as follows:

Number of bytes	Contents
For each reference of this type:	
2 bytes	Resource ID
2 bytes	Offset from beginning of resource name list to length of resource name, or -1 if none
1 byte	Resource attributes
3 bytes	If local reference, offset from beginning of resource data to length of data for this resource If system reference, Ø (ignored)
4 bytes	If local reference, Ø (reserved for handle to resource) If system reference, resource specification for system resource: in high-order word, resource ID; in low-order word, offset from beginning of resource name list to length of resource name, or -1 if none

The resource name list follows the reference list and has this format:

Number of bytes	Contents
For each name:	
1 byte	Length of following resource name
n bytes	Characters of resource name

Figure 9 on the following page shows where the various offsets lead to in a resource file, in general and also specifically for a local reference.

NOTES FOR ASSEMBLY-LANGUAGE PROGRAMMERS

*** This will be moved to a separate chapter of the final comprehensive manual. For now, see the QuickDraw manual for complete information about how to use the User Interface Toolbox from assembly language. ***

The primary aid to assembly-language programmers is a file named ToolEqu.Text. If you use .INCLUDE to include this file when you assemble your program, all the Resource Manager constants and locations of system globals will be available in symbolic form.

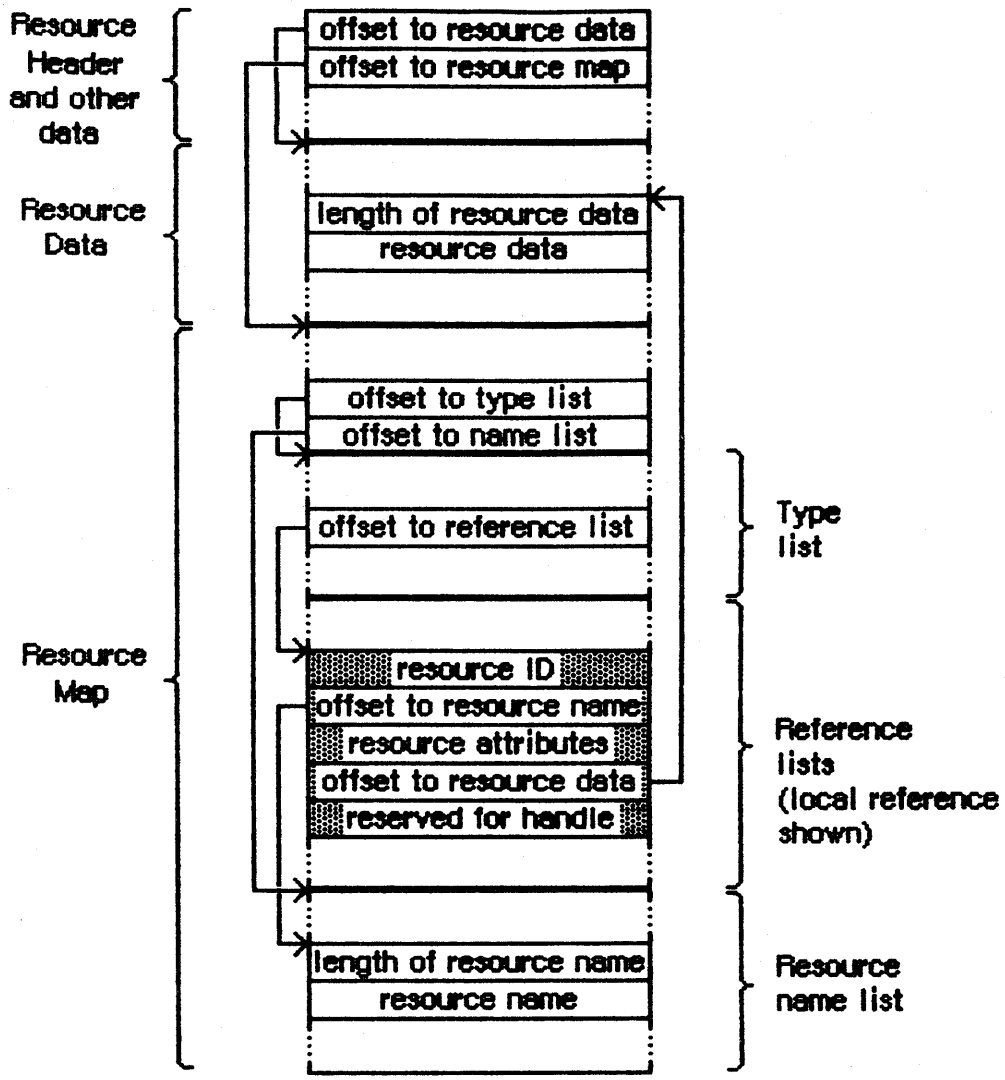


Figure 9. Local Reference in a Resource File

SUMMARY OF THE RESOURCE MANAGER

```
CONST resSysRef     = 128;  {set if system reference}
      resSysHeap    = 64;   {set if read into system heap}
      resPurgeable  = 32;   {set if purgeable}
      resLocked     = 16;   {set if locked}
      resProtected  = 8;    {set if protected}
      resPreload    = 4;    {set if to be preloaded}
      resChanged    = 2;    {set if to be written to resource file}
      resUser       = 1;    {available for use by your application}

      resNotFound   = -192;   {resource not found}
      resFNotFound  = -193;   {resource file not found}
      addResFailed  = -194;   {AddResource failed}
      addRefFailed  = -195;   {AddReference failed}
      rmvResFailed  = -196;   {RmveResource failed}
      rmvRefFailed  = -197;   {RmveReference failed}

      mapReadOnly = 128;
      mapCompact  = 64;
      mapChanged  = 32;

TYPE ResType = PACKED ARRAY [1..4] OF CHAR;
```

Initializing the Resource Manager

```
FUNCTION  InitResources : INTEGER;
PROCEDURE RsrcZoneInit;
```

Opening and Closing Resource Files

```
PROCEDURE CreateResFile (filename: Str255);
FUNCTION  OpenResFile   (fileName: Str255) : INTEGER;
PROCEDURE CloseResFile  (refNum: INTEGER);
```

Checking for Errors

```
FUNCTION ResError : INTEGER;
```

Setting the Current Resource File

```
FUNCTION  CurResFile : INTEGER;
FUNCTION  HomeResFile (theResource: Handle) : INTEGER;
PROCEDURE UseResFile  (refNum: INTEGER);
```

Getting Resource Types

```
FUNCTION  CountTypes : INTEGER;
PROCEDURE GetIndType  (VAR theType: ResType; index: INTEGER);
```

Getting and Disposing of Resources

```
PROCEDURE SetResLoad       (load: BOOLEAN);
FUNCTION  CountResources   (theType: ResType) : INTEGER;
FUNCTION  GetIndResource   (theType: ResType; index: INTEGER) : Handle;
FUNCTION  GetResource      (theType: ResType; theID: INTEGER) : Handle;
FUNCTION  GetNamedResource (theType: ResType; name: Str255) : Handle;
PROCEDURE LoadResource     (theResource: Handle);
PROCEDURE ReleaseResource  (theResource: Handle);
PROCEDURE DetachResource   (theResource: Handle);
```

Getting Resource Information

```
FUNCTION  UniqueID    (theType: ResType) : INTEGER;
PROCEDURE GetResInfo  (theResource: Handle; VAR theID: INTEGER; VAR
                       theType: ResType; VAR name: Str255);
FUNCTION  GetResAttrs (theResource: Handle) : INTEGER;
```

Modifying Resources

```
PROCEDURE SetResInfo      (theResource: Handle; theID: INTEGER; name:
                           Str255);
PROCEDURE SetResAttrs     (theResource: Handle; attrs: INTEGER);
PROCEDURE ChangedResource (theResource: Handle);
PROCEDURE AddResource     (theData: Handle; theType: ResType; theID:
                           INTEGER; name: Str255);
PROCEDURE RmveResource    (theResource: Handle);
PROCEDURE AddReference    (theResource: Handle; theID: INTEGER; name:
                           Str255);
PROCEDURE RmveReference   (theResource: Handle);
PROCEDURE UpdateResFile   (refNum: INTEGER);
PROCEDURE WriteResource   (theResource: Handle);
PROCEDURE SetResPurge     (install: BOOLEAN);
```

Advanced Routines

```
FUNCTION  GetResFileAttrs (refNum: INTEGER) : INTEGER;
PROCEDURE SetResFileAttrs (refNum: INTEGER; attrs: INTEGER);
```

SUMMARY OF THE RESOURCE FILE FORMAT

(hand)

All offsets and lengths are given in bytes.

Resource Header and other data	4 bytes	Offset to resource data
	4 bytes	Offset to resource map
	4 bytes	Length of resource data
	4 bytes	Length of resource map
	112 bytes	Partial copy of file's directory entry
	128 bytes	User data
Resource Data	For each resource:	
	4 bytes	Length of following resource data
	n bytes	Resource data for this resource
Resource Map	16 bytes	Reserved for copy of resource header
	4 bytes	Reserved for handle to next resource map to be searched
	2 bytes	Reserved for file reference number
	2 bytes	Resource file attributes
	2 bytes	Offset to type list
	2 bytes	Offset to resource name list
Type list	2 bytes	Number of resource types minus 1
	For each type:	
	4 bytes	Resource type
	2 bytes	Number of resources of this type minus 1
	2 bytes	Offset to reference list for this type
Reference lists (one per type, contiguous, same order as in type list)	For each reference of this type:	
	2 bytes	Resource ID
	2 bytes	Offset to length of resource name or -1 if none
	1 byte	Resource attributes
	3 bytes	If local reference, offset to length of resource data If system reference, Ø
	4 bytes	If local, reserved for handle to resource If system, resource specification for system resource: in high-order word, resource ID; in low-order word, offset to length of resource name or -1 if none
Resource name list	For each name:	
	1 byte	Length of following resource name
	n bytes	Characters of resource name

GLOSSARY

current resource file: The last resource file opened, unless you specify otherwise with a Resource Manager routine.

empty handle: A pointer to a NIL master pointer.

local reference: A resource reference to a resource in the same file as the reference. It points to the resource data in the file and contains a handle to the data if it's in memory.

purgeable: Able to be removed from the heap (purged) when space is required by the Memory Manager.

reference number: A number greater than Ø, returned when a file is opened, by which you can refer to that file. In Resource Manager routines that expect a reference number, Ø represents the system resource file.

resource: Data or code stored in a resource file and managed by the Resource Manager.

resource attribute: One of several characteristics, specified by bits in a resource reference, that determine how the resource should be dealt with.

resource data: In a resource file, the data that comprises a resource.

resource file: The resource fork of a file, which contains data used by the application (such as menus, fonts, and icons) and also the application code itself.

resource header: At the beginning of a resource file, data that gives the offsets to and lengths of the resource data and resource map.

resource ID: A number that, together with the resource type, identifies a resource in a resource file. Every resource has an ID number.

resource map: In a resource file, data that is read into memory when the file is opened and that, given a resource specification, leads to the corresponding resource data.

resource name: A string that, together with the resource type, identifies a resource in a resource file. A resource may or may not have a name.

resource reference: In a resource map, a local reference leading to resource data in the same file as the reference, or a system reference leading to data in the system resource file.

resource specification: A resource type and either a resource ID or a resource name.

resource type: The type of a resource in a resource file, designated by a sequence of four characters (such as 'MENU' for a menu).

system reference: In an application's resource file, a resource reference to a system resource. It provides a resource specification for the resource in the system resource file.

system resource: A resource in the system resource file.

system resource file: A resource file containing standard resources, accessed if a requested resource wasn't found in any of the other resource files that were searched.

COMMENTS?

Macintosh User Education encourages your comments on this manual.

- What do you like or dislike about it?

- Were you able to find the information you needed?

- Was it complete and accurate?

- Do you have any suggestions for improvement?

Please send your comments to the author (indicated on the cover page) at 10460 Bandley Drive M/S 3-G, Cupertino CA 95014. Mark up a copy of the manual or note your remarks separately. (We'll return your marked-up copy if you like.)

Thanks for your help!

MACINTOSH PUBLICATIONS

QuickDraw: A Programmer's Guide /QUICK/QUIKDRAW

See Also: Macintosh User Interface Guidelines
Macintosh Operating System Reference Manual
The Window Manager: A Programmer's Guide

Modification History:			
	First Draft	C. Espinosa	11/27/81
	Revised and Edited	C. Espinosa	2/15/82
	Revised and Edited	C. Rose	8/16/82
	Errata Added	C. Rose	8/19/82
	Revised	C. Rose	11/15/82
	Revised for ROM 2.1	C. Rose	3/2/83

ABSTRACT

This document describes the QuickDraw graphics package, heart of the Macintosh User Interface Toolbox routines. It describes the conceptual and physical data types used by QuickDraw and gives details of the procedures and functions available in QuickDraw.

Summary of significant changes and additions since last version:

- "Font" no longer includes type size. There is a new grafPort field (txSize) and a procedure (TextSize) for specifying the size (pages 25, 43). Some other grafPort fields were reordered and some global variables were moved to the grafPort (page 18).
- The character style data type was renamed Style and now includes two new variations, condense and extend (page 23).
- You can set up your application now to produce color output when devices supporting it are available in the future (pages 3Ø, 45).
- The Polygon data type was changed (page 33), and the PolyNext procedure was removed.
- There are two new grafPort routines, InitPort and ClosePort (pages 35, 36), and three new calculation routines, EqualRect and EmptyRect (page 48) and EqualPt (page 65).
- XferRgn and XferRect were removed; use CopyBits, PaintRgn, FillRgn, PaintRect, or FillRect. CursorVis was also removed; use HideCursor or ShowCursor.
- A section on customizing QuickDraw operations was added (page 7Ø).

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes QuickDraw, a set of graphics procedures, functions, and data types that allow a Pascal or assembly-language programmer of Macintosh to perform highly complex graphic operations very easily and very quickly. It covers the graphic concepts behind QuickDraw, as well as the technical details of the data types, procedures, and functions you will use in your programs.

(hand)
This manual describes version 2.1 of the ROM. In earlier versions, QuickDraw may not work as discussed here.

We assume that you are familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's memory management. This graphics package is for programmers, not end users. Although QuickDraw may be used from either Pascal or assembly language, this manual gives all examples in their Pascal form, to be clear, concise, and more intuitive; a section near the end describes the details of the assembly-language interface to QuickDraw.

The manual begins with an introduction to QuickDraw and what you can do with it. It then steps back a little and looks at the mathematical concepts that form the foundation for QuickDraw: coordinate planes, points, and rectangles. Once you understand these concepts, read on about the graphic entities based on those concepts -- how the mathematical world of planes and rectangles is translated into the physical phenomena of light and shadow.

Then comes some discussion of how to use several graphics ports, a summary of the basic drawing process, and a discussion of two more parts of QuickDraw, pictures and polygons.

Next, there's the detailed description of all QuickDraw procedures and functions, their parameters, calling protocol, effects, side effects, and so on -- all the technical information you'll need each time you write a program for Macintosh.

Following these descriptions are sections that will not be of interest to all readers. Special information is given for programmers who want to customize QuickDraw operations by overriding the standard drawing procedures, and for those who will be using QuickDraw from assembly language.

Finally, there's a summary of the QuickDraw data structures and routine calls, for quick reference, and a glossary that explains terms that may be unfamiliar to you.

ABOUT QUICKDRAW

QuickDraw allows you to divide the Macintosh screen into a number of individual areas. Within each area you can draw many things, as illustrated in Figure 1.

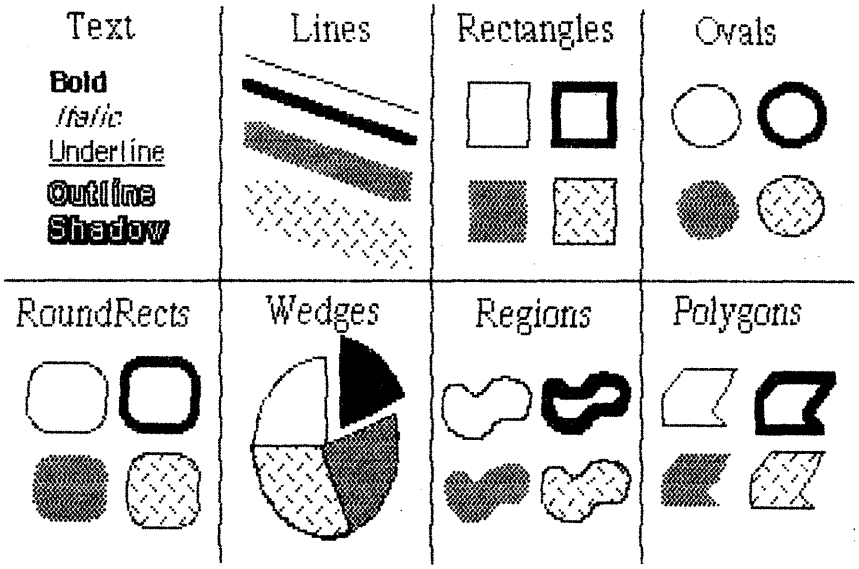


Figure 1. Samples of QuickDraw's Abilities

You can draw:

- Text characters in a number of proportionally-spaced fonts, with variations that include boldfacing, italicizing, underlining, and outlining.

- Straight lines of any length and width.

- A variety of shapes, either solid or hollow, including: rectangles, with or without rounded corners; full circles and ovals or wedge-shaped sections; and polygons.

- Any other arbitrary shape or collection of shapes, again either solid or hollow.

- A picture consisting of any combination of the above items, with just a single procedure call.

In addition, QuickDraw has some other abilities that you won't find in many other graphics packages. These abilities take care of most of the "housekeeping" -- the trivial but time-consuming and bothersome overhead that's necessary to keep things in order.

- The ability to define many distinct "ports" on the screen, each with its own complete drawing environment -- its own coordinate system, drawing location, character set, location on the screen, and so on. You can easily switch from one such port to another.

- Full and complete "clipping" to arbitrary areas, so that drawing will occur only where you want. It's like a super-duper coloring book that won't let you color outside the lines. You don't have to worry about accidentally drawing over something else on the screen, or drawing off the screen and destroying memory.

- Off-screen drawing. Anything you can draw on the screen, you can draw into an off-screen buffer, so you can prepare an image for an output device without disturbing the screen, or you can prepare a picture and move it onto the screen very quickly.

And QuickDraw lives up to its name! It's very fast. The speed and responsiveness of the Macintosh user interface is due primarily to the speed of the QuickDraw package. You can do good-quality animation, fast interactive graphics, and complex yet speedy text displays using the full features of QuickDraw. This means you don't have to bypass the general-purpose QuickDraw routines by writing a lot of special routines to improve speed.

How To Use QuickDraw

QuickDraw can be used from either Pascal or MC68ØØØ machine language. It has no user interface of its own; you must write and compile (or assemble) a Pascal (or assembly-language) program that includes the proper QuickDraw calls, link the resulting object code with the QuickDraw code, and execute the linked object file.

Some programming models are available through your Macintosh software coordinator; they show the structure of a properly organized QuickDraw program. What's best for beginners is to obtain a machine-readable version of the text of one of these programs, read through the text, and, using the superstructure of the program as a "shell", modify it to suit your own purposes. Once you get the hang of writing programs inside the presupplied shell, you can work on changing the shell itself.

QuickDraw is stored permanently in the ROM memory. All access is made through an indirection table in low RAM. When you write a program that uses QuickDraw, you link it with this indirection table. Each time you call a QuickDraw procedure or function, or load a predefined constant, the request goes through the table into QuickDraw. You'll never access any QuickDraw address directly, nor will you have to code constant addresses into your program. The linker will make sure all address references get straightened out.

QuickDraw is an independent unit; it doesn't use any other units, not even HeapZone (the Pascal interface to the Operating System's memory management routines). This means it cannot use the data types Ptr and Handle, because they are defined in HeapZone. Instead, QuickDraw defines two data types that are equivalent to Ptr and Handle, QDPtr and QDHandle.

```
TYPE QDByte    = -128..127;
     QDPtr     = ^QDByte;
     QDHandle  = ^QDPtr;
```

QuickDraw includes only the graphics and utility procedures and functions you'll need to create graphics on the screen. Keyboard input, mouse input, and larger user-interface constructs such as windows and menus are implemented in separate packages that use QuickDraw but are linked in as separate units. You don't need these units in order to use QuickDraw; however, you'll probably want to read the documentation for windows and menus and learn how to use them with your Macintosh programs.

THE MATHEMATICAL FOUNDATION OF QUICKDRAW

To create graphics that are both precise and pretty requires not supercharged features but a firm mathematical foundation for the features you have. If the mathematics that underlie a graphics package are imprecise or fuzzy, the graphics will be, too. QuickDraw defines some clear mathematical constructs that are widely used in its procedures, functions, and data types: the coordinate plane, the point, the rectangle, and the region.

The Coordinate Plane

All information about location, placement, or movement that you give to QuickDraw is in terms of coordinates on a plane. The coordinate plane is a two-dimensional grid, as illustrated in Figure 2.

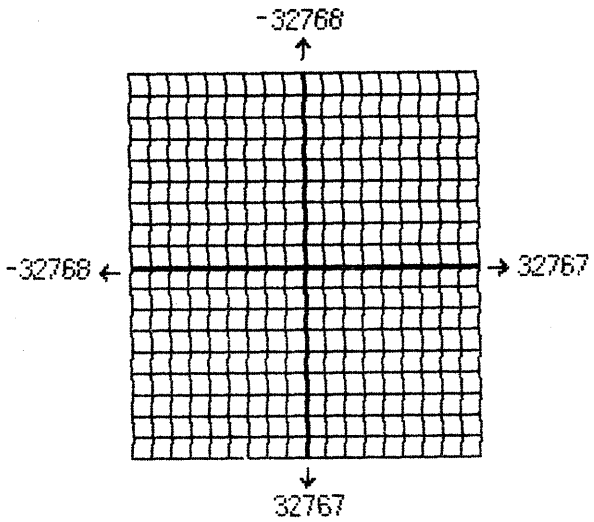


Figure 2. The Coordinate Plane

There are two distinctive features of the QuickDraw coordinate plane:

- All grid coordinates are integers.

- All grid lines are infinitely thin.

These concepts are important! First, they mean that the QuickDraw plane is finite, not infinite (although it's very large). Horizontal coordinates range from -32768 to +32767, and vertical coordinates have the same range. (An auxiliary package is available that maps real Cartesian space, with X, Y, and Z coordinates, onto QuickDraw's two-dimensional integer coordinate system.)

Second, they mean that all elements represented on the coordinate plane are mathematically pure. Mathematical calculations using integer arithmetic will produce intuitively correct results. If you keep in mind that grid lines are infinitely thin, you'll never have "endpoint paranoia" -- the confusion that results from not knowing whether that last dot is included in the line.

Points

On the coordinate plane are 4,294,967,296 unique points. Each point is at the intersection of a horizontal grid line and a vertical grid line. As the grid lines are infinitely thin, a point is infinitely small. Of course there are more points on this grid than there are dots on the Macintosh screen: when using QuickDraw you associate small parts of the grid with areas on the screen, so that you aren't bound into an arbitrary, limited coordinate system.

The coordinate origin (Ø,Ø) is in the middle of the grid. Horizontal coordinates increase as you move from left to right, and vertical coordinates increase as you move from top to bottom. This is the way both a TV screen and a page of English text are scanned: from the top left to the bottom right.

You can store the coordinates of a point into a Pascal variable whose type is defined by QuickDraw. The type Point is a record of two integers, and has this structure:

```
TYPE VHSelect = (V,H);
     Point    = RECORD CASE INTEGER OF

                  Ø:  (v:  INTEGER;
                       h:  INTEGER);

                  1:  (vh:  ARRAY [VHSelect] OF INTEGER)

                  END;
```

The variant part allows you to access the vertical and horizontal components of a point either individually or as an array. For example, if the variable goodPt were declared to be of type Point, the following would all refer to the coordinate parts of the point:

goodPt.v goodPt.h
goodPt.vh[V] goodPt.vh[H]

Rectangles

Any two points can define the top left and bottom right corners of a rectangle. As these points are infinitely small, the borders of the rectangle are infinitely thin (see Figure 3).

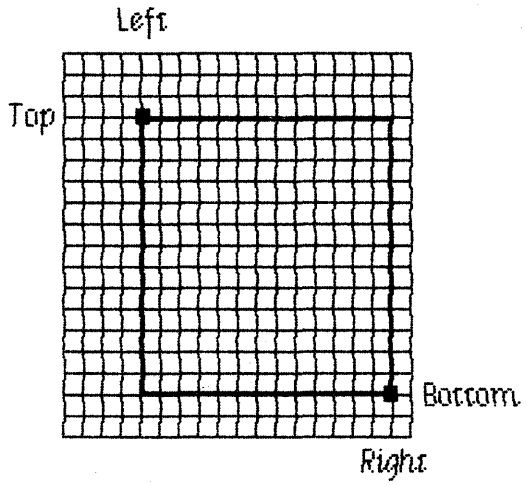


Figure 3. A Rectangle

Rectangles are used to define active areas on the screen, to assign coordinate systems to graphic entities, and to specify the locations and sizes for various drawing commands. QuickDraw also allows you to perform many mathematical calculations on rectangles -- changing their sizes, shifting them around, and so on.

(hand)
Remember that rectangles, like points, are mathematical concepts that have no direct representation on the screen. The association between these conceptual elements and their physical representations is made by a bitMap, described below.

The data type for rectangles is called Rect, and consists of four integers or two points:

```
TYPE Rect = RECORD CASE INTEGER OF

            Ø:  (top:      INTEGER;
                 left:     INTEGER;
                 bottom:   INTEGER;
                 right:    INTEGER);

            1:  (topLeft:   Point;
                 botRight:  Point)

            END;
```

Again, the record variant allows you to access a variable of type Rect either as four boundary coordinates or as two diagonally opposing corner points. Combined with the record variant for points, all of the following references to the rectangle named bRect are legal:

```
bRect                                                   {type Rect}

bRect.topLeft               bRect.botRight              {type Point}

bRect.top                   bRect.left                  {type INTEGER}
bRect.topLeft.v             bRect.topLeft.h             {type INTEGER}
bRect.topLeft.vh[V]         bRect.topLeft.vh[H]         {type INTEGER}

bRect.bottom                bRect.right                 {type INTEGER}
bRect.botRight.v            bRect.botRight.h            {type INTEGER}
bRect.botRight.vh[V]        bRect.botRight.vh[H]        {type INTEGER}
```

(eye)
If the bottom coordinate of a rectangle is equal to or less than the top, or the right coordinate is equal to or less than the left, the rectangle is an empty rectangle (i.e., one that contains no bits).

Regions

Unlike most graphics packages that can manipulate only simple, geometric structures (usually rectilinear, at that), QuickDraw has the unique and amazing ability to gather an arbitrary set of spatially coherent points into a structure called a region, and perform complex yet rapid manipulations and calculations on such structures. This remarkable feature not only will make your standard programs simpler and faster, but will let you perform operations that would otherwise be nearly impossible; it is fundamental to the Macintosh user interface.

You define a region by drawing lines, shapes such as rectangles and ovals, or even other regions. The outline of a region should be one or more closed loops. A region can be concave or convex, can consist of one area or many disjoint areas, and can even have "holes" in the middle. In Figure 4, the region on the left has a hole in the middle, and the region on the right consists of two disjoint areas.

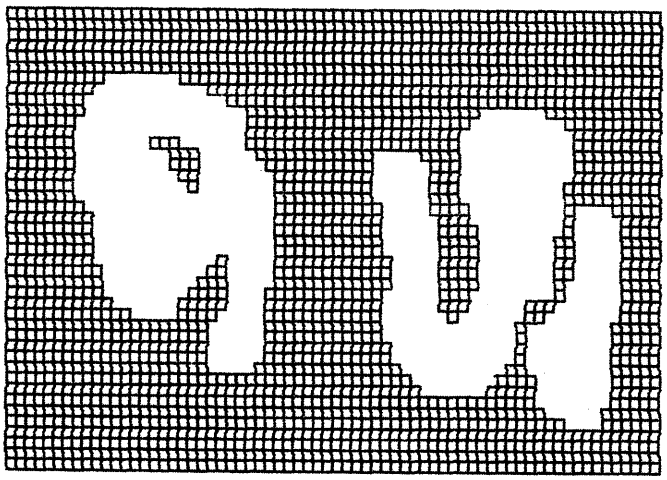


Figure 4. Regions

Because a region can be any arbitrary area or set of areas on the coordinate plane, it takes a variable amount of information to store the outline of a region. The data structure for a region, therefore, is a variable-length entity with two fixed fields at the beginning, followed by a variable-length data field:

```
TYPE Region = RECORD
                rgnSize:  INTEGER;
                rgnBBox:  Rect;
                {optional region definition data}
              END;
```

The rgnSize field contains the size, in bytes, of the region variable. The rgnBBox field is a rectangle which completely encloses the region.

The simplest region is a rectangle. In this case, the rgnBBox field defines the entire region, and there is no optional region data. For rectangular regions (or empty regions), the rgnSize field contains 10.

The region definition data for nonrectangular regions is stored in a compact way which allows for highly efficient access by QuickDraw procedures.

As regions are of variable size, they are stored dynamically on the heap, and the Operating System's memory management moves them around as their sizes change. Being dynamic, a region can be accessed only through a pointer; but when a region is moved, all pointers referring to it must be updated. For this reason, all regions are accessed through handles, which point to one master pointer which in turn points to the region.

```
TYPE RgnPtr    = ^Region;
     RgnHandle = ^RgnPtr;
```

When the memory management relocates a region's data in memory, it updates only the RgnPtr master pointer to that region. The references through the master pointer can find the region's new home, but any references pointing directly to the region's previous position in memory would now point at dead bits. To access individual fields of a region, use the region handle and double indirection:

```
myRgn^^.rgnSize          {size of region whose handle is myRgn}
myRgn^^.rgnBBox          {rectangle enclosing the same region}
myRgn^^.rgnBBox.top      {minimum vertical coordinate of all
                          points in the region}

myRgn^.rgnBBox           {syntactically incorrect; will not compile
                         if myRgn is a rgnHandle}
```

Regions are created by a QuickDraw function which allocates space for the region, creates a master pointer, and returns a rgnHandle. When you're done with a region, you dispose of it with another QuickDraw routine which frees up the space used by the region. Only these calls allocate or deallocate regions; do NOT use the Pascal procedure NEW to create a new region!

You specify the outline of a region with procedures that draw lines and shapes, as described in the section "QuickDraw Routines". An example is given in the discussion of CloseRgn under "Calculations with Regions" in that section.

Many calculations can be performed on regions. A region can be "expanded" or "shrunk" and, given any two regions, QuickDraw can find their union, intersection, difference, and exclusive-OR; it can also determine whether a given point or rectangle intersects a given region, and so on. There is of course a set of graphic operations on regions to draw them on the screen.

GRAPHIC ENTITIES

Coordinate planes, points, rectangles, and regions are all good mathematical models, but they aren't really graphic elements -- they don't have a direct physical appearance. Some graphic entities that do have a direct graphic interpretation are the bit image, bitMap, pattern, and cursor. This section describes the data structure of these graphic entities and how they relate to the mathematical constructs described above.

The Bit Image

A bit image is a collection of bits in memory which have a rectilinear representation. Take a collection of words in memory and lay them end to end so that bit 15 of the lowest-numbered word is on the left and bit Ø of the highest-numbered word is on the far right. Then take this array of bits and divide it, on word boundaries, into a number of equal-size rows. Stack these rows vertically so that the first row is on the top and the last row is on the bottom. The result is a matrix like the one shown in Figure 5 -- rows and columns of bits, with each row containing the same number of bytes. The number of bytes in each row of the bit image is called the row width of that image.

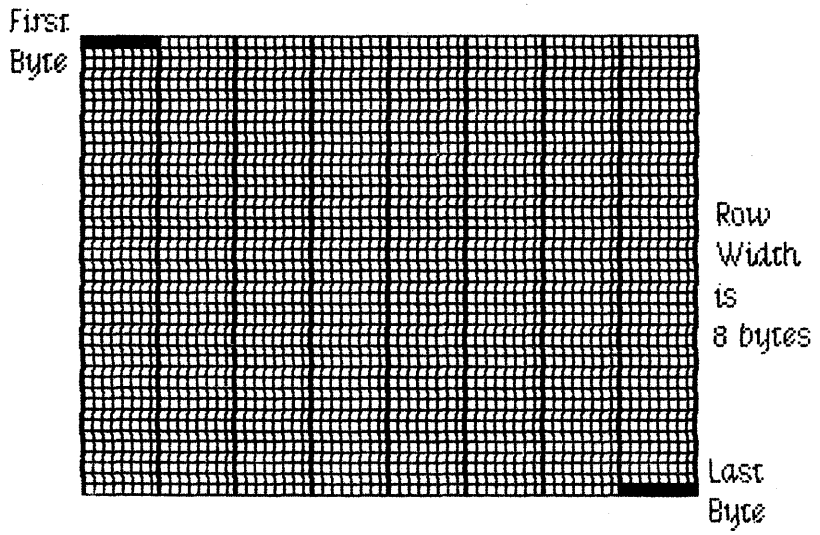


Figure 5. A Bit Image

A bit image can be stored in any static or dynamic variable, and can be of any length that is a multiple of the row width.

The Macintosh screen itself is one large visible bit image. The upper 21,888 bytes of memory are displayed as a matrix of 175,1Ø4 pixels on the screen, each bit corresponding to one pixel. If a bit's value is Ø, its pixel is white; if the bit's value is 1, the pixel is black.

The screen is 342 pixels tall and 512 pixels wide, and the row width of its bit image is 64 bytes. Each pixel on the screen is square; there are 72 pixels per inch in each direction.

(hand)

> Since each pixel on the screen represents one bit in a bit image, wherever this document says "bit", you can substitute "pixel" if the bit image is the Macintosh screen. Likewise, this document often refers to pixels on the screen where the discussion applies equally to bits in an off-screen bit image.

The BitMap

When you combine the physical entity of a bit image with the conceptual entities of the coordinate plane and rectangle, you get a bitMap. A bitMap has three parts: a pointer to a bit image, the row width (in bytes) of that image, and a boundary rectangle which gives the bitMap both its dimensions and a coordinate system. Notice that a bitMap does not actually include the bits themselves: it points to them.

There can be several bitMaps pointing to the same bit image, each imposing a different coordinate system on it. This important feature is explained more fully in "Coordinates in GrafPorts", below.

As shown in Figure 6, the data structure of a bitMap is as follows:

```
TYPE BitMap = RECORD
                baseAddr:  QDPtr;
                rowBytes:  INTEGER;
                bounds:    Rect
              END;
```

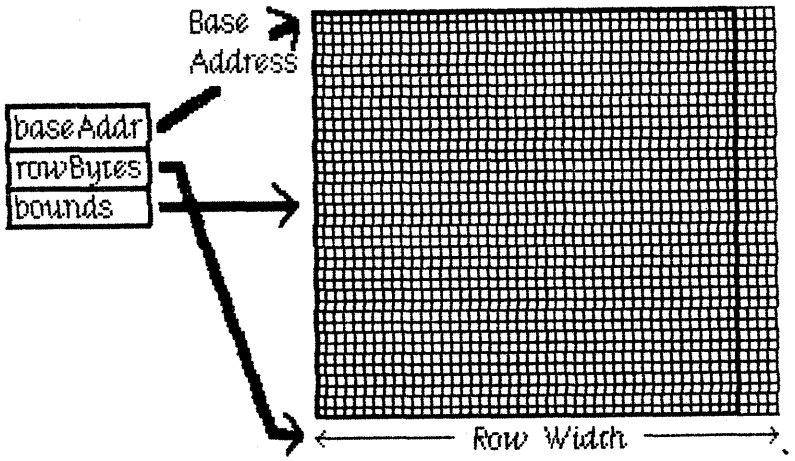


Figure 6. A BitMap

The baseAddr field is a pointer to the beginning of the bit image in memory, and the rowBytes field is the number of bytes in each row of the image. Both of these should always be even: a bitMap should always begin on a word boundary and contain an integral number of words in each row.

The bounds field is a boundary rectangle that both encloses the active area of the bit image and imposes a coordinate system on it. The relationship between the boundary rectangle and the bit image in a bitMap is simple yet very important. First, a few general rules:

- Bits in a bit image fall between points on the coordinate plane.
- A rectangle divides a bit image into two sets of bits: those bits inside the rectangle and those outside the rectangle.
- A rectangle that is H points wide and V points tall encloses exactly (H-1)*(V-1) bits.

The top left corner of the boundary rectangle is aligned around the first bit in the bit image. The width of the rectangle determines how many bits of one row are logically owned by the bitMap; the relationship

```
8*map.rowBytes >= map.bounds.right-map.bounds.left
```

must always be true. The height of the rectangle determines how many rows of the image are logically owned by the bitMap; the relationship

```
SIZEOF(map.baseAddr^) >= (map.bounds.bottom-map.bounds.top)
                                    * map.rowBytes
```

must always be true to ensure that the number of bits in the logical bitMap area is not larger than the number of bits in the bit image.

Normally, the boundary rectangle completely encloses the bit image: the width of the boundary rectangle is equal to the number of bits in one row of the image, and the height of the rectangle is equal to the number of rows in the image. If the rectangle is smaller than the dimensions of the image, the least significant bits in each row, as well as the last rows in the image, are not affected by any operations on the bitMap.

The bitMap also imposes a coordinate system on the image. Because bits fall between coordinate points, the coordinate system assigns integer values to the lines that border and separate bits, not to the bit positions themselves. For example, if a bitMap is assigned the boundary rectangle with corners (10,-8) and (34,8), the bottom right bit in the image will be between horizontal coordinates 33 and 34, and between vertical coordinates 7 and 8 (see Figure 7).

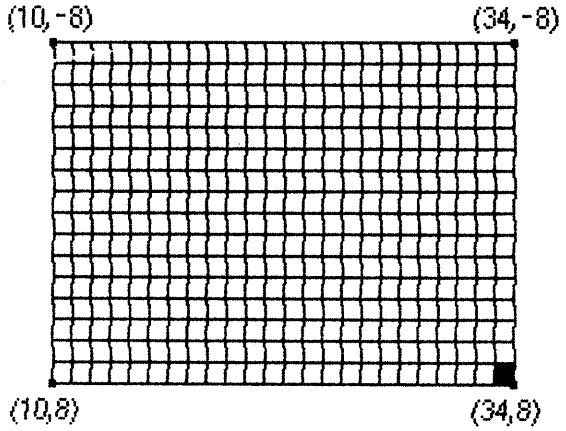


Figure 7. Coordinates and BitMaps

Patterns

A pattern is a 64-bit image, organized as an 8-by-8-bit square, which is used to define a repeating design (such as stripes) or tone (such as gray). Patterns can be used to draw lines and shapes or to fill areas on the screen.

When a pattern is drawn, it is aligned such that adjacent areas of the same pattern in the same graphics port will blend with it into a continuous, coordinated pattern. QuickDraw provides the predefined patterns white, black, gray, ltGray, and dkGray. Any other 64-bit variable or constant can be used as a pattern, too. The data type definition for a pattern is as follows:

```
TYPE Pattern = PACKED ARRAY [Ø..7] OF Ø..255;
```

The row width of a pattern is 1 byte.

Cursors

A cursor is a small image that appears on the screen and is controlled by the mouse. (It appears only on the screen, and never in an off-screen bit image.)

(hand)

Other Macintosh documentation calls this image a "pointer", since it points to a location on the screen. To avoid confusion with other meanings of "pointer" in this manual and other Toolbox documentation, we use the alternate term "cursor".

A cursor is defined as a 256-bit image, a 16-by-16-bit square. The row width of a cursor is 2 bytes. Figure 8 illustrates four cursors.

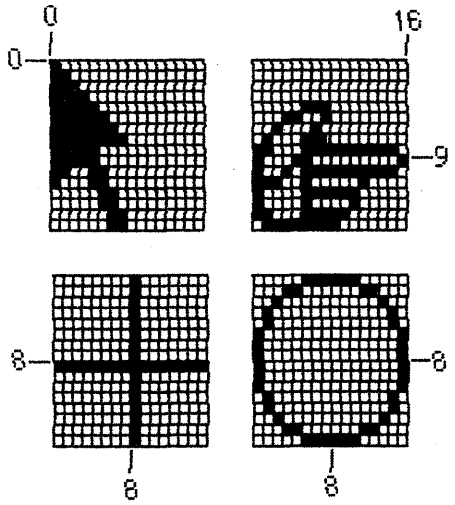


Figure 8. Cursors

A cursor has three fields: a 16-word data field that contains the image itself, a 16-word mask field that contains information about the screen appearance of each bit of the cursor, and a hotSpot point that aligns the cursor with the position of the mouse.

```
TYPE Cursor = RECORD
                data:     ARRAY [Ø..15] OF INTEGER;
                mask:     ARRAY [Ø..15] OF INTEGER;
                hotSpot:  Point
              END;
```

The data for the cursor must begin on a word boundary.

The cursor appears on the screen as a 16-by-16-bit square. The appearance of each bit of the square is determined by the corresponding bits in the data and mask and, if the mask bit is Ø, by the pixel "under" the cursor (the one already on the screen in the same position as this bit of the cursor):

Data	Mask	Resulting pixel on screen
Ø	1	White
1	1	Black
Ø	Ø	Same as pixel under cursor
1	Ø	Inverse of pixel under cursor

Notice that if all mask bits are Ø, the cursor is completely transparent, in that the image under the cursor can still be viewed: pixels under the white part of the cursor appear unchanged, while under the black part of the cursor, black pixels show through as white.

The hotSpot aligns a point in the image (not a bit, a point!) with the mouse position. Imagine the rectangle with corners (Ø,Ø) and (16,16) framing the image, as in each of the examples in Figure 8; the hotSpot is defined in this coordinate system. A hotSpot of (Ø,Ø) is at the top left of the image. For the arrow in Figure 8 to point to the mouse position, (Ø,Ø) would be its hotSpot. A hotSpot of (8,8) is in the exact center of the image; the center of the plus sign or circle in Figure 8 would coincide with the mouse position if (8,8) were the hotSpot for that cursor. Similarly, the hotSpot for the pointing hand would be (16,9).

Whenever you move the mouse, the low-level interrupt-driven mouse routines move the cursor's hotSpot to be aligned with the new mouse position.

(hand)

> The mouse position is always linked to the cursor position. You can't reposition the cursor through software; the only control you have is whether it's visible or not, and what shape it will assume. Think of it as being hard-wired: if the cursor is visible, it always follows the mouse over the full size of the screen.

QuickDraw supplies a predefined arrow cursor, an arrow pointing north-northwest.

THE DRAWING ENVIRONMENT: GRAFPORT

A grafPort is a complete drawing environment that defines how and where graphic operations will have their effect. It contains all the information about one instance of graphic output that is kept separate from all other instances. You can have many grafPorts open at once, and each one will have its own coordinate system, drawing pattern, background pattern, pen size and location, character font and style, and bitMap in which drawing takes place. You can instantly switch from one port to another. GrafPorts are the structures on which a program builds windows, which are fundamental to the Macintosh "overlapping windows" user interface.

A grafPort is a dynamic data structure, defined as follows:

```
TYPE GrafPtr  = ^GrafPort;
     GrafPort = RECORD
                  device:      INTEGER;
                  portBits:    BitMap;
                  portRect:    Rect;
                  visRgn:      RgnHandle;
                  clipRgn:     RgnHandle;
                  bkPat:       Pattern;
                  fillPat:     Pattern;
                  pnLoc:       Point;
                  pnSize:      Point;
                  pnMode:      INTEGER;
                  pnPat:       Pattern;
                  pnVis:       INTEGER;
                  txFont:      INTEGER;
                  txFace:      Style;
                  txMode:      INTEGER;
                  txSize:      INTEGER;
                  spExtra:     INTEGER;
                  fgColor:     LongInt;
                  bkColor:     LongInt;
                  colrBit:     INTEGER;
                  patStretch:  INTEGER;
                  picSave:     QDHandle;
                  rgnSave:     QDHandle;
                  polySave:    QDHandle;
                  grafProcs:   QDProcsPtr
                END;
```

All QuickDraw operations refer to grafPorts via grafPtrs. You create a grafPort with the Pascal procedure NEW and use the resulting pointer in calls to QuickDraw. You could, of course, declare a static VAR of type grafPort, and obtain a pointer to that static structure (with the @ operator), but as most grafPorts will be used dynamically, their data structures should be dynamic also.

(hand)

> You can access all fields and subfields of a grafPort normally, but you should not store new values directly into them. QuickDraw has procedures for altering all fields of a grafPort, and using these procedures ensures that changing a grafPort produces no unusual side effects.

The device field of a grafPort is the number of the logical output device that the grafPort will be using. The Font Manager uses this information, since there are physical differences in the same logical font for different output devices. The default device number is Ø, for the Macintosh screen. For more information about device numbers, see the *** not yet existing *** Font Manager documentation.

The portBits field is the bitMap that points to the bit image to be used by the grafPort. All drawing that is done in this grafPort will take place in this bit image. The default bitMap uses the entire Macintosh screen as its bit image, with rowBytes of 64 and a boundary rectangle of (Ø,Ø,512,342). The bitMap may be changed to indicate a different structure in memory: all graphics procedures work in exactly the same way regardless of whether their effects are visible on the screen. A program can, for example, prepare an image to be printed on a printer without ever displaying the image on the screen, or develop a picture in an off-screen bitMap before transferring it to the screen. By altering the coordinates of the portBits.bounds rectangle, you can change the coordinate system of the grafPort; with a QuickDraw procedure call, you can set an arbitrary coordinate system for each grafPort, even if the different grafPorts all use the same bit image (e.g., the full screen).

The portRect field is a rectangle that defines a subset of the bitMap for use by the grafPort. Its coordinates are in the system defined by the portBits.bounds rectangle. All drawing done by the application occurs inside this rectangle. The portRect usually defines the "writable" interior area of a window, document, or other object on the screen.

The visRgn field is manipulated by the Window Manager; users and programmers will normally never change a grafPort's visRgn. It indicates that region (remember, an arbitrary area or set of areas) which is actually visible on the screen. For example, if you move one window in front of another, the Window Manager logically removes the area of overlap from the visRgn of the window in the back. When you draw into the back window, whatever's being drawn is clipped to the visRgn so that it doesn't run over onto the front window. The default visRgn is set to the portRect. The visRgn has no effect on images that are not displayed on the screen.

The clipRgn is an arbitrary region that the application can use to limit drawing to any region within the portRect. If, for example, you want to draw a half circle on the screen, you can set the clipRgn to half the square that would enclose the whole circle, and go ahead and draw the whole circle. Only the half within the clipRgn will actually be drawn in the grafPort. The default clipRgn is set arbitrarily large, and you have full control over its setting. Notice that unlike the visRgn, the clipRgn affects the image even if it is not displayed on the screen.

Figure 9 illustrates a typical bitMap (as defined by portBits), portRect, visRgn, and clipRgn.

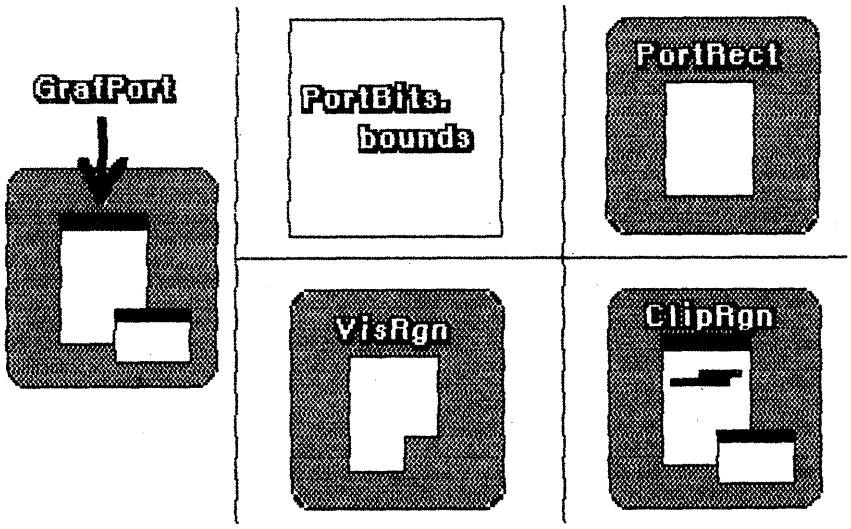


Figure 9. GrafPort Regions

The bkPat and fillPat fields of a grafPort contain patterns used by certain QuickDraw routines. BkPat is the "background" pattern that is used when an area is erased or when bits are scrolled out of it. When asked to fill an area with a specified pattern, QuickDraw stores the given pattern in the fillPat field and then calls a low-level drawing routine which gets the pattern from that field. The various graphic operations are discussed in detail later in the descriptions of individual QuickDraw routines.

Of the next ten fields, the first five determine characteristics of the graphics pen and the last five determine characteristics of any text that may be drawn; these are described in subsections below.

The fgColor, bkColor, and colrBit fields contain values related to drawing in color, a capability that will be available in the future when Apple supports color output devices for the Macintosh. FgColor is the grafPort's foreground color and bkColor is its background color. ColrBit tells the color imaging software which plane of the color picture to draw into. For more information, see "Drawing in Color" in the general discussion of drawing.

The patStretch field is used during output to a printer to expand patterns if necessary. The application should not change its value.

The picSave, rgnSave, and polySave fields reflect the state of picture, region, and polygon defintion, respectively. To define a region, for example, you "open" it, call routines that draw it, and then "close" it. If no region is open, rgnSave contains NIL; otherwise, it contains a handle to information related to the region definition. The application should not be concerned about exactly what information the handle leads to; you may, however, save the current value of rgnSave, set the field to NIL to disable the region definition, and later restore it to the saved value to resume the region definition. The

picSave and polySave fields work similarly for pictures and polygons.

Finally, the grafProcs field may point to a special data structure that the application stores into if it wants to customize QuickDraw drawing procedures or use QuickDraw in other advanced, highly specialized ways. (For more information, see "Customizing QuickDraw Operations".) If grafProcs is NIL, QuickDraw responds in the standard ways described in this manual.

Pen Characteristics

The pnLoc, pnSize, pnMode, pnPat, and pnVis fields of a grafPort deal with the graphics pen. Each grafPort has one and only one graphics pen, which is used for drawing lines, shapes, and text. As illustrated in Figure 1Ø, the pen has four characteristics: a location, a size, a drawing mode, and a drawing pattern.

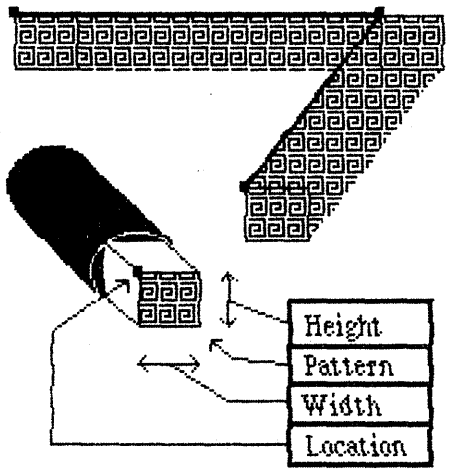


Figure 1Ø. A Graphics Pen

The pen location is a point in the coordinate system of the grafPort, and is where QuickDraw will begin drawing the next line, shape, or character. It can be anywhere on the coordinate plane: there are no restrictions on the movement or placement of the pen. Remember that the pen location is a point on the coordinate plane, not a pixel in a bit image!

The pen is rectangular in shape, and has a user-definable width and height. The default size is a 1-by-1-bit square; the width and height can range from (Ø,Ø) to (32767,32767). If either the pen width or the pen height is less than 1, the pen will not draw on the screen.

- The pen appears as a rectangle with its top left corner at the pen location; it hangs below and to the right of the pen location.

The pnMode and pnPat fields of a grafPort determine how the bits under the pen are affected when lines or shapes are drawn. The pnPat is a pattern that is used like the "ink" in the pen. This pattern, like all other patterns drawn in the grafPort, is always aligned with the port's coordinate system: the top left corner of the pattern is aligned with the top left corner of the portRect, so that adjacent areas of the same pattern will blend into a continuous, coordinated pattern. Five patterns are predefined (white, black, and three shades of gray); you can also create your own pattern and use it as the pnPat. (A utility procedure, called StuffHex, allows you to fill patterns easily.)

The pnMode field determines how the pen pattern is to affect what's already on the bitMap when lines or shapes are drawn. When the pen draws, QuickDraw first determines what bits of the bitMap will be affected and finds their corresponding bits in the pattern. It then does a bit-by-bit evaluation based on the pen mode, which specifies one of eight boolean operations to perform. The resulting bit is placed into its proper place in the bitMap. The pen modes are described under "Transfer Modes" in the general discussion of drawing below.

The pnVis field determines the pen's visibility, that is, whether it draws on the screen. For more information, see the descriptions of HidePen and ShowPen under "Pen and Line-Drawing Routines" in the "QuickDraw Routines" section.

Text Characteristics

The txFont, txFace, txMode, txSize, and spExtra fields of a grafPort determine how text will be drawn -- the font, style, and size of characters and how they will be placed on the bitMap.

(hand)

> In the Macintosh User Interface Toolbox, character style means stylistic variations such as bold, italic, and underline; font means the complete set of characters of one typeface, such as Helvetica, and does not include the character style or size.

QuickDraw can draw characters as quickly and easily as it draws lines and shapes, and in many prepared fonts. Figure 11 shows two QuickDraw characters and some terms you should become familiar with.

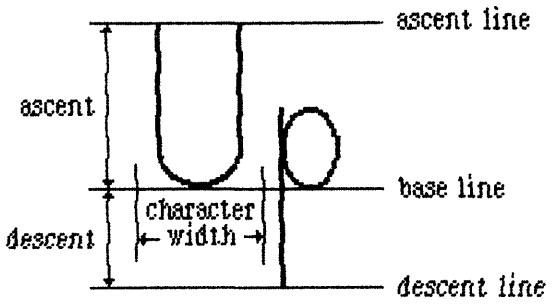


Figure 11. QuickDraw Characters

QuckDraw can display characters in any size, as well as boldfaced, italicized, outlined, or shadowed, all without changing fonts. It can also underline the characters, or draw them closer together or farther apart.

The txFont field is a font number that identifies the character font to be used in the grafPort. The font number Ø represents the system font. For more information about the system font, the other font numbers recognized by the Font Manager, and the construction, layout, and loading of fonts, see the *** not yet existing *** Font Manager documentation.

A character font is defined as a collection of bit images: these images make up the individual characters of the font. The characters can be of unequal widths, and they're not restricted to their "cells": the lower curl of a lowercase j, for example, can stretch back under the previous character (typographers call this kerning). A font can consist of up to 256 distinct characters, yet not all characters need be defined in a single font. Each font contains a missing symbol to be drawn in case of a request to draw a character that is missing from the font.

The txFace field controls the appearance of the font with values from the set defined by the Style data type:

```
TYPE StyleItem = (bold, italic, underline, outline, shadow,
                  condense, extend);
     Style     = SET OF StyleItem;
```

You can apply these either alone or in combination (see Figure 12). Most combinations usually look good only for large fonts.

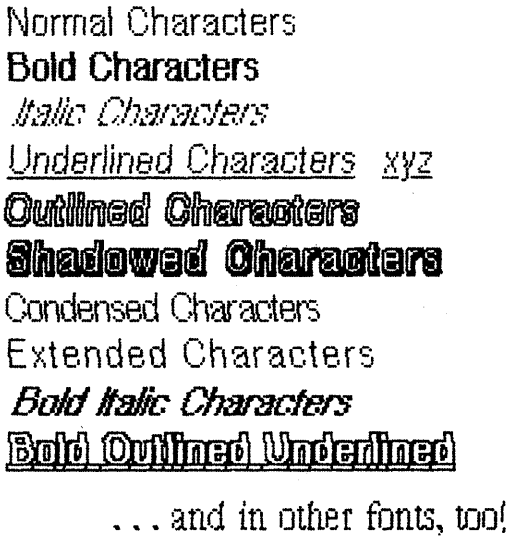


Figure 12. Character Styles

If you specify bold, each character is repeatedly drawn one bit to the right an appropriate number of times for extra thickness.

Italic adds an italic slant to the characters. Character bits above the base line are skewed right; bits below the base line are skewed left.

Underline draws a line below the base line of the characters. If part of a character descends below the base line (as "y" in Figure 12), the underline is not drawn through the pixel on either side of the descending part.

You may specify either outline or shadow. Outline makes a hollow, outlined character rather than a solid one. With shadow, not only is the character hollow and outlined, but the outline is thickened below and to the right of the character to achieve the effect of a shadow. If you specify bold along with outline or shadow, the hollow part of the character is widened.

Condense and extend affect the horizontal distance between all characters, including spaces. Condense decreases the distance between characters and extend increases it, by an amount which the Font Manager determines is appropriate.

The txMode field controls the way characters are placed on a bit image. It functions much like a pnMode: when a character is drawn, QuickDraw determines which bits of the bit image will be affected, does a bit-by-bit comparison based on the mode, and stores the resulting bits into the bit image. These modes are described under "Transfer Modes" in the general discussion of drawing below. Only three of them -- srcOr, srcXor, and srcBic -- should be used for drawing text.

The txSize field specifies the type size for the font, in points (where "point" here is a printing term meaning 1/72 inch). Any size may be specified. If the Font Manager does not have the font in a specified size, it will scale a size it does have as necessary to produce the size desired. A value of Ø in this field directs the Font Manager to choose the size from among those it has for the font; it will choose whichever size is closest to the system font size.

Finally, the spExtra field is useful when a line of characters is to be drawn justified such that it is aligned with both a left and a right margin (sometimes called "full justification"). SpExtra is the number of pixels by which each space character should be widened to fill out the line.

COORDINATES IN GRAFPORTS

Each grafPort has its own local coordinate system. All fields in the grafPort are expressed in these coordinates, and all calculations and actions performed in QuickDraw use the local coordinate system of the currently selected port.

Two things are important to remember:

- Each grafPort maps a portion of the coordinate plane into a similarly-sized portion of a bit image.

- The portBits.bounds rectangle defines the local coordinates for a grafPort.

The top left corner of portBits.bounds is always aligned around the first bit in the bit image; the coordinates of that corner "anchor" a point on the grid to that bit in the bit image. This forms a common reference point for multiple grafPorts using the same bit image (such as the screen). Given a portBits.bounds rectangle for each port, you know that their top left corners coincide.

The interrelationship between the portBits.bounds and portRect rectangles is very important. As the portBits.bounds rectangle establishes a coordinate system for the port, the portRect rectangle indicates the section of the coordinate plane (and thus the bit image) that will be used for drawing. The portRect usually falls inside the portBits.bounds rectangle, but it's not required to do so.

When a new grafPort is created, its bitMap is set to point to the entire Macintosh screen, and both the portBits.bounds and the portRect rectangles are set to 512-by-342-bit rectangles, with the point (Ø,Ø) at the top left corner of the screen.

You can redefine the local coordinates of the top left corner of the grafPort's portRect, using the SetOrigin procedure. This changes the local coordinate system of the grafPort, recalculating the coordinates of all points in the grafPort to be relative to the new corner

coordinates. For example, consider these procedure calls:

```
SetPort(gamePort);
SetOrigin(4Ø,8Ø);
```

The call to SetPort sets the current grafPort to gamePort; the call to SetOrigin changes the local coordinates of the top left corner of that port's portRect to (4Ø,8Ø) (see Figure 13).

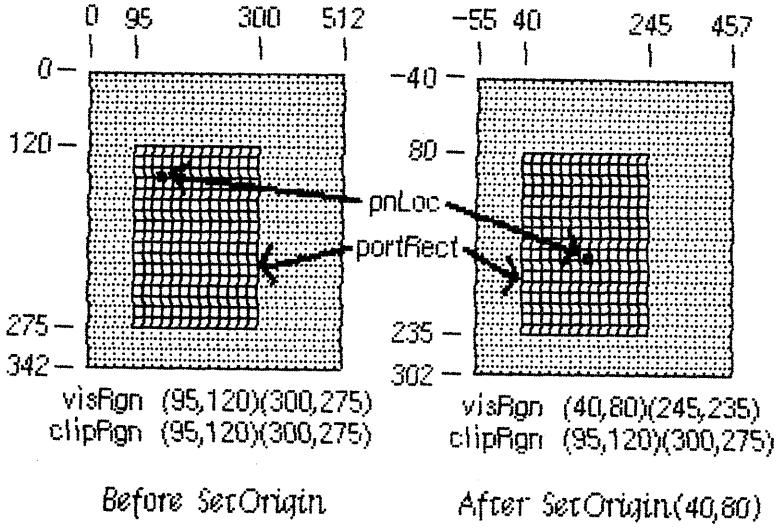


Figure 13. Changing Local Coordinates

This recalculates the coordinate components of the following elements:

```
gamePort^.portBits.bounds        gamePort^.portRect

gamePort^.visRgn
```

These elements are always kept "in sync", so that all calculations, comparisons, or operations that seem right, work right.

Notice that when the local coordinates of a grafPort are offset, the visRgn of that port is offset also, but the clipRgn is not. A good way to think of it is that if a document is being shown inside a grafPort, the document "sticks" to the coordinate system, and the port's structure "sticks" to the screen. Suppose, for example, that the visRgn and clipRgn in Figure 13 before SetOrigin are the same as the portRect, and a document is being shown. After the SetOrigin call, the top left corner of the clipRgn is still (95,12Ø), but this location has moved down and to the right, and the location of the pen within the document has similarly moved. The locations of portBits.bounds, portRect, and visRgn did not change; their coordinates were offset. As always, the top left corner of portBits.bounds remains aligned around the first bit in the bit image (the first pixel on the screen).

If you are moving, comparing, or otherwise dealing with mathematical items in different grafPorts (for example, finding the intersection of

two regions in two different grafPorts), you must adjust to a common coordinate system before you perform the operation. A QuickDraw procedure, LocalToGlobal, lets you convert a point's local coordinates to a global system where the top left corner of the bit image is (Ø,Ø); by converting the various local coordinates to global coordinates, you can compare and mix them with confidence. For more information, see the description of this procedure under "Calculations with Points" in the section "QuickDraw Routines".

GENERAL DISCUSSION OF DRAWING

Drawing occurs:

- Always inside a grafPort, in the bit image and coordinate system defined by the grafPort's bitMap.
- Always within the intersection of the grafPort's portBits.bounds and portRect, and clipped to its visRgn and clipRgn.
- Always at the grafPort's pen location.
- Usually with the grafPort's pen size, pattern, and mode.

With QuickDraw procedures, you can draw lines, shapes, and text. Shapes include rectangles, ovals, rounded-corner rectangles, wedge-shaped sections of ovals, regions, and polygons.

Lines are defined by two points: the current pen location and a destination location. When drawing a line, QuickDraw moves the top left corner of the pen along the mathematical trajectory from the current location to the destination. The pen hangs below and to the right of the trajectory (see Figure 14).

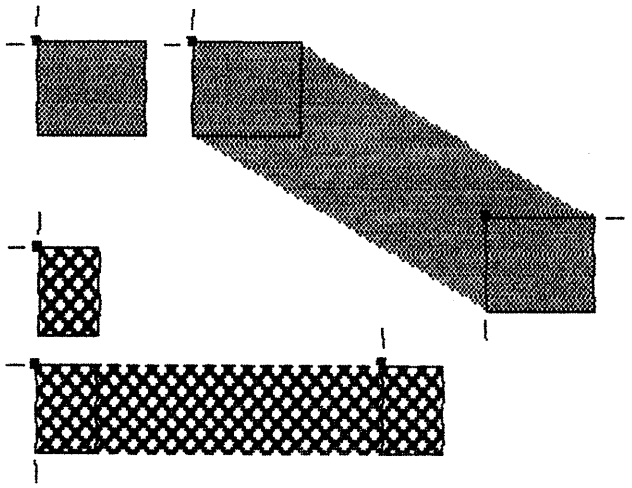


Figure 14. Drawing Lines

(hand)

No mathematical element (such as the pen location) is ever affected by clipping; clipping only determines what appears where in the bit image. If you draw a line to a location outside your grafPort, the pen location will move there, but only the portion of the line that is inside the port will actually be drawn. This is true for all drawing procedures.

Rectangles, ovals, and rounded-corner rectangles are defined by two corner points. The shapes always appear inside the mathematical rectangle defined by the two points. A region is defined in a more complex manner, but also appears only within the rectangle enclosing it. Remember, these enclosing rectangles have infinitely thin borders and are not visible on the screen.

As illustrated in Figure 15, shapes may be drawn either solid (filled in with a pattern) or framed (outlined and hollow).

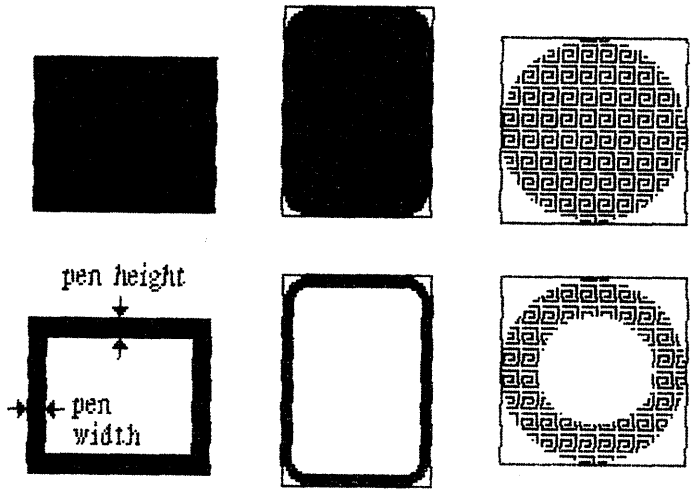


Figure 15. Solid Shapes and Framed Shapes

In the case of framed shapes, the outline appears completely within the enclosing rectangle -- with one exception -- and the vertical and horizontal thickness of the outline is determined by the pen size. The exception is polygons, as discussed in "Pictures and Polygons" below.

The pen pattern is used to fill in the bits that are affected by the drawing operation. The pen mode defines how those bits are to be affected by directing QuickDraw to apply one of eight boolean operations to the bits in the shape and the corresponding pixels on the screen.

Text drawing does not use the pnSize, pnPat, or pnMode, but it does use the pnLoc. Each character is placed to the right of the current pen location, with the left end of its base line at the pen's location. The pen is moved to the right to the location where it will draw the

next character. No wrap or carriage return is performed automatically.

The method QuickDraw uses in placing text is controlled by a mode similar to the pen mode. This is explained in "Transfer Modes", below. Clipping of text is performed in exactly the same manner as all other clipping in QuickDraw.

Transfer Modes

When lines or shapes are drawn, the pnMode field of the grafPort determines how the drawing is to appear in the port's bit image; similarly, the txMode field determines how text is to appear. There is also a QuickDraw procedure that transfers a bit image from one bitMap to another, and this procedure has a mode parameter that determines the appearance of the result. In all these cases, the mode, called a transfer mode, specifies one of eight boolean operations: for each bit in the item to be drawn, QuickDraw finds the corresponding bit in the destination bit image, performs the boolean operation on the pair of bits, and stores the resulting bit into the bit image.

There are two types of transfer mode:

- Pattern transfer modes, for drawing lines or shapes with a pattern.

- Source transfer modes, for drawing text or transferring any bit image between two bitMaps.

For each type of mode, there are four basic operations -- Copy, Or, Xor, and Bic. The Copy operation simply replaces the pixels in the destination with the pixels in the pattern or source, "painting" over the destination without regard for what is already there. The Or, Xor, and Bic operations leave the destination pixels under the white part of the pattern or source unchanged, and differ in how they affect the pixels under the black part: Or replaces those pixels with black pixels, thus "overlaying" the destination with the black part of the pattern or source; Xor inverts the pixels under the black part; and Bic erases them to white.

Each of the basic operations has a variant in which every pixel in the pattern or source is inverted before the operation is performed, giving eight operations in all. Each mode is defined by name as a constant in QuickDraw (see Figure 16).

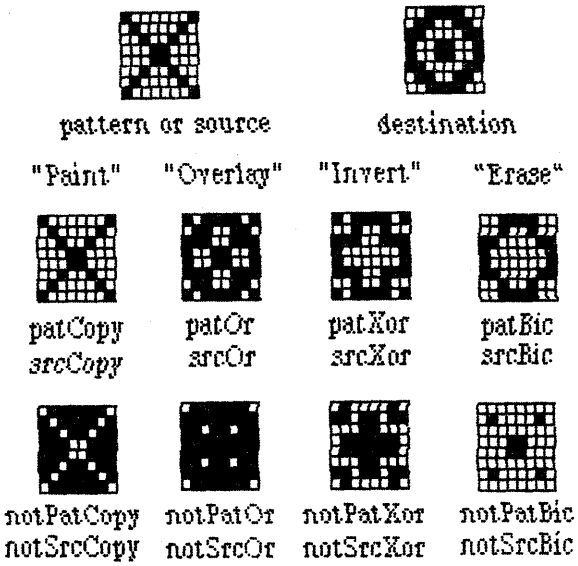


Figure 16. Transfer Modes

Pattern transfer mode	Source transfer mode	Action on each pixel in destination: If black pixel in pattern or source	If white pixel in pattern or source
patCopy	srcCopy	Force black	Force white
patOr	srcOr	Force black	Leave alone
patXor	srcXor	Invert	Leave alone
patBic	srcBic	Force white	Leave alone
notPatCopy	notSrcCopy	Force white	Force black
notPatOr	notSrcOr	Leave alone	Force black
notPatXor	notSrcXor	Leave alone	Invert
notPatBic	notSrcBic	Leave alone	Force white

Drawing in Color

Currently you can only look at QuickDraw output on a black-and-white screen or printer. Eventually, however, Apple will support color output devices. If you want to set up your application now to produce color output in the future, you can do so by using QuickDraw procedures to set the foreground color and the background color. Eight standard colors may be specified with the following predefined constants: blackColor, whiteColor, redColor, greenColor, blueColor, cyanColor, magentaColor, and yellowColor. Initially, the foreground color is blackColor and the background color is whiteColor. If you specify a color other than whiteColor, it will appear as black on a black-and-white output device.

To apply the table in the "Transfer Modes" section above to drawing in color, make the following translation: where the table shows "Force black", read "Force foreground color", and where it shows "Force white", read "Force background color". When you eventually receive the

color output device, you'll find out the effect of inverting a color on it.

(hand)
QuickDraw can support output devices that have up to 32 bits of color information per pixel. A color picture may be thought of, then, as having up to 32 planes. At any one time, QuickDraw draws into only one of these planes. A QuickDraw routine called by the color-imaging software specifies which plane.

PICTURES AND POLYGONS

QuickDraw lets you save a sequence of drawing commands and "play them back" later with a single procedure call. There are two such mechanisms: one for drawing any picture to scale in a destination rectangle that you specify, and another for drawing polygons in all the ways you can draw other shapes in QuickDraw.

Pictures

A picture in QuickDraw is a transcript of calls to routines which draw something -- anything -- on a bitMap. Pictures make it easy for one program to draw something defined in another program, with great flexibility and without knowing the details about what's being drawn.

For each picture you define, you specify a rectangle that surrounds the picture; this rectangle is called the picture frame. When you later call the procedure that draws the saved picture, you supply a destination rectangle, and QuickDraw scales the picture so that its frame is completely aligned with the destination rectangle. Thus, the picture may be expanded or shrunk to fit its destination rectangle. For example, if the picture is a circle inside a square picture frame, and the destination rectangle is not square, the picture is drawn as an oval.

Since a picture may include any sequence of drawing commands, its data structure is a variable-length entity. It consists of two fixed fields followed by a variable-length data field:

```
TYPE Picture = RECORD
                 picSize:   INTEGER;
                 picFrame:  Rect;
                 {picture definition data}
               END;
```

The picSize field contains the size, in bytes, of the picture variable. The picFrame field is the picture frame which surrounds the picture and gives a frame of reference for scaling when the picture is drawn. The rest of the structure contains a compact representation of the drawing

commands that define the picture.

All pictures are accessed through handles, which point to one master pointer which in turn points to the picture.

```
TYPE PicPtr    = ^Picture;
     PicHandle = ^PicPtr;
```

To define a picture, you call a QuickDraw function that returns a picHandle and then call the routines that draw the picture. There is a procedure to call when you've finished defining the picture, and another for when you're done with the picture altogether.

QuickDraw also allows you to intersperse picture comments in with the definition of a picture. These comments, which do not affect the picture's appearance, may be used to provide additional information about the picture when it's played back. This is especially valuable when pictures are transmitted from one application to another. There are two standard types of comment which, like parentheses, serve to group drawing commands together (such as all the commands that draw a particular part of a picture):

```
CONST picLParen = 0;
      picRParen = 1;
```

The application defining the picture can use these standard comments as well as comments of its own design.

To include a comment in the definition of a picture, the application calls a QuickDraw procedure that specifies the comment with three parameters: the comment kind, which identifies the type of comment; a handle to additional data if desired; and the size of the additional data, if any. When playing back a picture, QuickDraw passes any comments in the picture's definition to a low-level procedure accessed indirectly through the grafProcs field of the grafPort (see "Customizing QuickDraw Operations" for more information). To process comments, the application must include a procedure to do the processing and store a pointer to it in the data structure pointed to by the grafProcs field.

(hand)
The standard low-level procedure for processing picture comments simply ignores all comments.

Polygons

Polygons are similar to pictures in that you define them by a sequence of calls to QuickDraw routines. They are also similar to other shapes that QuickDraw knows about, since there is a set of procedures for performing graphic operations and calculations on them.

A polygon is simply any sequence of connected lines (see Figure 17). You define a polygon by moving to the starting point of the polygon and

drawing lines from there to the next point, from that point to the next, and so on.

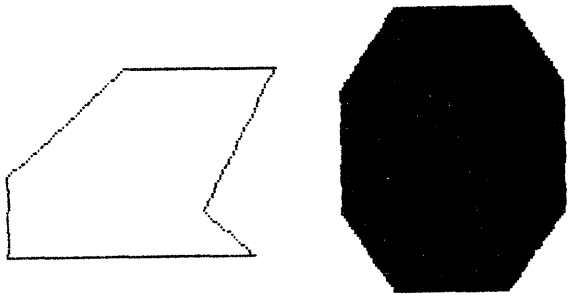


Figure 17. Polygons

The data structure for a polygon is a variable-length entity. It consists of two fixed fields followed by a variable-length array:

```
TYPE Polygon = RECORD
                 polySize:    INTEGER;
                 polyBBox:    Rect;
                 polyPoints:  ARRAY [Ø..Ø] OF Point
               END;
```

The polySize field contains the size, in bytes, of the polygon variable. The polyBBox field is a rectangle which just encloses the entire polygon. The polyPoints array expands as necessary to contain the points of the polygon -- the starting point followed by each succesive point to which a line is drawn.

Like pictures and regions, polygons are accessed through handles.

```
TYPE PolyPtr    = ^Polygon;
     PolyHandle = ^PolyPtr;
```

To define a polygon, you call a QuickDraw function that returns a polyHandle and then form the polygon by calling procedures that draw lines. You call a procedure when you've finished defining the polygon, and another when you're done with the polygon altogether.

Just as for other shapes that QuickDraw knows about, there is a set of graphic operations on polygons to draw them on the screen. QuickDraw draws a polygon by moving to the starting point and then drawing lines to the remaining points in succession, just as when the routines were called to define the polygon. In this sense it "plays back" those routine calls. As a result, polygons are not treated exactly the same

as other QuickDraw shapes. For example, the procedure that frames a polygon draws outside the actual boundary of the polygon, because QuickDraw line-drawing routines draw below and to the right of the pen location. The procedures that fill a polygon with a pattern, however, stay within the boundary of the polygon; they also add an additional line between the ending point and the starting point if those points are not the same, to complete the shape.

There is also a difference in the way QuickDraw scales a polygon and a similarly-shaped region if it's being drawn as part of a picture: when stretched, a slanted line is drawn more smoothly if it's part of a polygon rather than a region. You may find it helpful to keep in mind the conceptual difference between polygons and regions: a polygon is treated more as a continuous shape, a region more as a set of bits.

QUICKDRAW ROUTINES

This section describes all the procedures and functions in QuickDraw, their parameters, and their operation. They are presented in their Pascal form; for information on using them from assembly language, see "Using QuickDraw from Assembly Language".

GrafPort Routines

```
PROCEDURE InitGraf (globalPtr: QDPtr);
```

Call InitGraf once and only once at the beginning of your program to initialize QuickDraw. It initializes the QuickDraw global variables listed below.

Variable	Type	Initial setting
thePort	GrafPtr	NIL
white	Pattern	all-white pattern
black	Pattern	all-black pattern
gray	Pattern	50% gray pattern
ltGray	Pattern	25% gray pattern
dkGray	Pattern	75% gray pattern
arrow	Cursor	pointing arrow cursor
screenBits	BitMap	Macintosh screen, (0,0,512,342)
randSeed	LongInt	1

The globalPtr parameter tells QuickDraw where to store its global variables, beginning with thePort. From Pascal programs, this parameter should always be set to @thePort; assembly-language programmers may choose any location, as long as it can accommodate the number of bytes specified by GRAFSIZE in GRAFTYPES.TEXT (see "Using QuickDraw from Assembly Language").

(hand)
To initialize the cursor, call InitCursor (described under "Cursor-Handling Routines" below).

PROCEDURE OpenPort (gp: GrafPtr);

OpenPort allocates space for the given grafPort's visRgn and clipRgn, initializes the fields of the grafPort as indicated below, and makes the grafPort the current port (see SetPort). You must call OpenPort before using any grafPort; first perform a NEW to create a grafPtr and then use that grafPtr in the OpenPort call.

Field	Type	Initial setting
device	INTEGER	Ø (Macintosh screen)
portBits	BitMap	screenBits (see InitGraf)
portRect	Rect	screenBits.bounds (Ø,Ø,512,342)
visRgn	RgnHandle	handle to the rectangular region (Ø,Ø,512,342)
clipRgn	RgnHandle	handle to the rectangular region (-3ØØØØ,-3ØØØØ,3ØØØØ,3ØØØØ)
bkPat	Pattern	white
fillPat	Pattern	black
pnLoc	Point	(Ø,Ø)
pnSize	Point	(1,1)
pnMode	INTEGER	patCopy
pnPat	Pattern	black
pnVis	INTEGER	Ø (visible)
txFont	INTEGER	Ø (system font)
txFace	Style	normal
txMode	INTEGER	srcOr
txSize	INTEGER	Ø (Font Manager decides)
spExtra	INTEGER	Ø
fgColor	LongInt	blackColor
bkColor	LongInt	whiteColor
colrBit	INTEGER	Ø
patStretch	INTEGER	Ø
picSave	QDHandle	NIL
rgnSave	QDHandle	NIL
polySave	QDHandle	NIL
grafProcs	QDProcsPtr	NIL

PROCEDURE InitPort (gp: GrafPtr);

Given a pointer to a grafPort that has been opened with OpenPort, InitPort reinitializes the fields of the grafPort and makes it the current port (if it's not already).

(hand)
InitPort does everything OpenPort does except allocate space for the visRgn and clipRgn.

PROCEDURE ClosePort (gp: GrafPtr);

ClosePort deallocates the space occupied by the given grafPort's visRgn and clipRgn. When you are completely through with a grafPort, call this procedure and then dispose of the grafPort (with a DISPOSE of the grafPtr).

(eye)
If you do not call ClosePort before disposing of the grafPort, the memory used by the visRgn and clipRgn will be unrecoverable.

(eye)
After calling ClosePort, be sure not to use any copies of the visRgn or clipRgn handles that you may have made.

PROCEDURE SetPort (gp: GrafPtr);

SetPort sets the grafPort indicated by gp to be the current port. The global pointer thePort always points to the current port. All QuickDraw drawing routines affect the bitMap thePort^.portBits and use the local coordinate system of thePort^. Note that OpenPort and InitPort do a SetPort to the given port.

(eye)
Never do a SetPort to a port that has not been opened with OpenPort.

Each port possesses its own pen and text characteristics which remain unchanged when the port is not selected as the current port.

PROCEDURE GetPort (VAR gp: GrafPtr);

GetPort returns a pointer to the current grafPort. If you have a program that draws into more than one grafPort, it's extremely useful to have each procedure save the current grafPort (with GetPort), set its own grafPort, do drawing or calculations, and then restore the previous grafPort (with SetPort). The pointer to the current grafPort is also available through the global pointer thePort, but you may prefer to use GetPort for better readability of your program text. For example, a procedure could do a GetPort(savePort) before setting its own grafPort and a SetPort(savePort) afterwards to restore the previous port.

PROCEDURE GrafDevice (device: INTEGER);

GrafDevice sets thePort^.device to the given number, which identifies the logical output device for this grafPort. The Font Manager uses this information. The initial device number is Ø, which represents the Macintosh screen.

```
PROCEDURE SetPortBits (bm: BitMap);
```

SetPortBits sets thePort^.portBits to any previously defined bitMap. This allows you to perform all normal drawing and calculations on a buffer other than the Macintosh screen -- for example, a 64Ø-by-7 output buffer for a C. Itoh printer, or a small off-screen image for later "stamping" onto the screen.

Remember to prepare all fields of the bitMap before you call SetPortBits.

```
PROCEDURE PortSize (width,height: INTEGER);
```

PortSize changes the size of the current grafPort's portRect. THIS DOES NOT AFFECT THE SCREEN; it merely changes the size of the "active area" of the grafPort.

(hand)
This procedure is normally called only by the Window Manager.

The top left corner of the portRect remains at its same location; the width and height of the portRect are set to the given width and height. In other words, PortSize moves the bottom right corner of the portRect to a position relative to the top left corner.

PortSize does not change the clipRgn or the visRgn, nor does it affect the local coordinate system of the grafPort: it changes only the portRect's width and height. Remember that all drawing occurs only in the intersection of the portBits.bounds and the portRect, clipped to the visRgn and the clipRgn.

```
PROCEDURE MovePortTo (leftGlobal,topGlobal: INTEGER);
```

MovePortTo changes the position of the current grafPort's portRect. THIS DOES NOT AFFECT THE SCREEN; it merely changes the location at which subsequent drawing inside the port will appear.

(hand)
This procedure is normally called only by the Window Manager.

The leftGlobal and topGlobal parameters set the distance between the top left corner of portBits.bounds and the top left corner of the new portRect. For example,

```
MovePortTo(256,171);
```

will move the top left corner of the portRect to the center of the screen (if portBits is the Macintosh screen) regardless of the local coordinate system.

Like PortSize, MovePortTo does not change the clipRgn or the visRgn, nor does it affect the local coordinate system of the grafPort.

PROCEDURE SetOrigin (h,v: INTEGER);

SetOrigin changes the local coordinate system of the current grafPort. THIS DOES NOT AFFECT THE SCREEN; it does, however, affect where subsequent drawing and calculation will appear in the grafPort. SetOrigin updates the coordinates of the portBits.bounds, the portRect, and the visRgn. All subsequent drawing and calculation routines will use the new coordinate system.

The h and v parameters set the coordinates of the top left corner of the portRect. All other coordinates are calculated from this point. All relative distances among any elements in the port will remain the same; only their absolute local coordinates will change.

(hand)
SetOrigin does not update the coordinates of the clipRgn or the pen; these items stick to the coordinate system (unlike the port's structure, which sticks to the screen).

SetOrigin is useful for adjusting the coordinate system after a scrolling operation. (See ScrollRect under "Bit Transfer Operations" below.)

PROCEDURE SetClip (rgn: RgnHandle);

SetClip changes the clipping region of the current grafPort to a region equivalent to the given region. Note that this does not change the region handle, but affects the clipping region itself. Since SetClip makes a copy of the given region, any subsequent changes you make to that region will not affect the clipping region of the port.

You can set the clipping region to any arbitrary region, to aid you in drawing inside the grafPort. The initial clipRgn is an arbitrarily large rectangle.

PROCEDURE GetClip (rgn: RgnHandle);

GetClip changes the given region to a region equivalent to the clipping region of the current grafPort. This is the reverse of what SetClip does. Like SetClip, it does not change the region handle.

PROCEDURE ClipRect (r: Rect);

ClipRect changes the clipping region of the current grafPort to a rectangle equivalent to given rectangle. Note that this does not change the region handle, but affects the region itself.

```
PROCEDURE BackPat (pat: Pattern);
```

BackPat sets the background pattern of the current grafPort to the given pattern. The background pattern is used in ScrollRect and in all QuickDraw routines that perform an "erase" operation.

Cursor-Handling Routines

```
PROCEDURE InitCursor;
```

InitCursor sets the current cursor to the predefined arrow cursor, an arrow pointing north-northwest, and sets the cursor level to Ø, making the cursor visible. The cursor level, which is initialized to Ø when the system is booted, keeps track of the number of times the cursor has been hidden to compensate for nested calls to HideCursor and ShowCursor (below).

Before you call InitCursor, the cursor is undefined (or, if set by a previous process, it's whatever that process set it to).

```
PROCEDURE SetCursor (crsr: Cursor);
```

SetCursor sets the current cursor to the 16-by-16-bit image in crsr. If the cursor is hidden, it remains hidden and will attain the new appearance when it's uncovered; if the cursor is already visible, it changes to the new appearance immediately.

The cursor image is initialized by InitCursor to a north-northwest arrow, visible on the screen. There is no way to retrieve the current cursor image.

```
PROCEDURE HideCursor;
```

HideCursor removes the cursor from the screen, restoring the bits under it, and decrements the cursor level (which InitCursor initialized to Ø). Every call to HideCursor should be balanced by a subsequent call to ShowCursor.

```
PROCEDURE ShowCursor;
```

ShowCursor increments the cursor level, which may have been decremented by HideCursor, and displays the cursor on the screen if the level becomes Ø. A call to ShowCursor should balance each previous call to HideCursor. The level is not incremented beyond Ø, so extra calls to ShowCursor don't hurt.

QuickDraw low-level interrupt-driven routines link the cursor with the mouse position, so that if the cursor level is Ø (visible), the cursor

automatically follows the mouse. You don't need to do anything but a ShowCursor to have a cursor track the mouse. There is no way to "disconnect" the cursor from the mouse; you can't force the cursor to a certain position, nor can you easily prevent the cursor from entering a certain area of the screen.

If the cursor has been changed (with SetCursor) while hidden, ShowCursor presents the new cursor.

The cursor is initialized by InitCursor to a north-northwest arrow, not hidden.

```
PROCEDURE ObscureCursor;
```

ObscureCursor hides the cursor until the next time the mouse is moved. Unlike HideCursor, it has no effect on the cursor level and must not be balanced by a call to ShowCursor.

Pen and Line-Drawing Routines

The pen and line-drawing routines all depend on the coordinate system of the current grafPort. Remember that each grafPort has its own pen; if you draw in one grafPort, change to another, and return to the first, the pen will have remained in the same location.

```
PROCEDURE HidePen;
```

HidePen decrements the current grafPort's pnVis field, which is initialized to Ø by OpenPort; whenever pnVis is negative, the pen does not draw on the screen. PnVis keeps track of the number of times the pen has been hidden to compensate for nested calls to HidePen and ShowPen (below). HidePen is called by OpenRgn, OpenPicture, and OpenPoly so that you can define regions, pictures, and polygons without drawing on the screen.

```
PROCEDURE ShowPen;
```

ShowPen increments the current grafPort's pnVis field, which may have been decremented by HidePen; if pnVis becomes Ø, QuickDraw resumes drawing on the screen. Extra calls to ShowPen will increment pnVis beyond Ø, so every call to ShowPen should be balanced by a subsequent call to HidePen. ShowPen is called by CloseRgn, ClosePicture, and ClosePoly.

```
PROCEDURE GetPen (VAR pt: Point);
```

GetPen returns the current pen location, in the local coordinates of the current grafPort.

```
PROCEDURE GetPenState (VAR pnState: PenState);
```

GetPenState saves the pen location, size, pattern, and mode into a storage variable, to be restored later with SetPenState (below). This is useful when calling short subroutines that operate in the current port but must change the graphics pen: each such procedure can save the pen's state when it's called, do whatever it needs to do, and restore the previous pen state immediately before returning.

The PenState data type is not useful for anything except saving the pen's state.

```
PROCEDURE SetPenState (pnState: PenState);
```

SetPenState sets the pen location, size, pattern, and mode in the current grafPort to the values stored in pnState. This is usually called at the end of a procedure that has altered the pen parameters and wants to restore them to their state at the beginning of the procedure. (See GetPenState, above.)

```
PROCEDURE PenSize (width,height: INTEGER);
```

PenSize sets the dimensions of the graphics pen in the current grafPort. All subsequent calls to Line, LineTo, and the procedures that draw framed shapes in the current grafPort will use the new pen dimensions.

The pen dimensions can be accessed in the variable thePort^.pnSize, which is of type Point. If either of the pen dimensions is set to a negative value, the pen assumes the dimensions (Ø,Ø) and no drawing is performed. For a discussion of how the pen draws, see the "General Discussion of Drawing" earlier in this manual.

```
PROCEDURE PenMode (mode: INTEGER);
```

PenMode sets the transfer mode through which the pnPat is transferred onto the bitMap when lines or shapes are drawn. The mode may be any one of the pattern transfer modes:

patCopy	patXor	notPatCopy	notPatXor
patOr	patBic	notPatOr	notPatBic

If the mode is one of the source transfer modes (or negative), no drawing is performed. The current pen mode can be obtained in the variable thePort^.pnMode. The initial pen mode is patCopy, in which the pen pattern is copied directly to the bitMap.

```
PROCEDURE PenPat (pat: Pattern);
```

PenPat sets the pattern that is used by the pen in the current grafPort. The standard patterns white, black, gray, ltGray, and dkGray are predefined; the initial pnPat is black. The current pen pattern can be obtained in the variable thePort^.pnPat, and this value can be assigned (but not compared!) to any other variable of type Pattern.

```
PROCEDURE PenNormal;
```

PenNormal resets the initial state of the pen in the current grafPort, as follows:

Field	Setting
pnSize	(1,1)
pnMode	patCopy
pnPat	black

The pen location is not changed.

```
PROCEDURE MoveTo (h,v: INTEGER);
```

MoveTo moves the pen to location (h,v) in the local coordinates of the current grafPort. No drawing is performed.

```
PROCEDURE Move (dh,dv: INTEGER);
```

This procedure moves the pen a distance of dh horizontally and dv vertically from its current location; it calls MoveTo(h+dh,v+dv), where (h,v) is the current location. The positive directions are to the right and down. No drawing is performed.

```
PROCEDURE LineTo (h,v: INTEGER);
```

LineTo draws a line from the current pen location to the location specified (in local coordinates) by h and v. The new pen location is (h,v) after the line is drawn. See the general discussion of drawing.

If a region or polygon is open and being formed, its outline is infinitely thin and is not affected by the pnSize, pnMode, or pnPat. (See OpenRgn and OpenPoly.)

```
PROCEDURE Line (dh,dv: INTEGER);
```

This procedure draws a line to the location that is a distance of dh horizontally and dv vertically from the current pen location; it calls LineTo(h+dh,v+dv), where (h,v) is the current location. The positive directions are to the right and down. The pen location becomes the coordinates of the end of the line after the line is drawn. See the

general discussion of drawing.

If a region or polygon is open and being formed, its outline is infinitely thin and is not affected by the pnSize, pnMode, or pnPat. (See OpenRgn and OpenPoly.)

Text-Drawing Routines

Each grafPort has its own text characteristics, and all these procedures deal with those of the current port.

```
PROCEDURE TextFont (font: INTEGER);
```

TextFont sets the current grafPort's font (thePort^.txFont) to the given font number. The initial font number is Ø, which represents the system font.

```
PROCEDURE TextFace (face: Style);
```

TextFace sets the current grafPort's character style (thePort^.txFace). The Style data type allows you to specify a set of one or more of the following predefined constants: bold, italic, underline, outline, shadow, condense, and extend. For example:

```
TextFace([bold]);                       {bold}
TextFace([bold,italic]);                {bold and italic}
TextFace(thePort^.txFace+[bold]);       {whatever it was plus bold}
TextFace(thePort^.txFace-[bold]);       {whatever it was but not bold}
TextFace([]);                           {normal}
```

```
PROCEDURE TextMode (mode: INTEGER);
```

TextMode sets the current grafPort's transfer mode for drawing text (thePort^.txMode). The mode should be srcOr, srcXor, or srcBic. The initial transfer mode for drawing text is srcOr.

```
PROCEDURE TextSize (size: INTEGER);
```

TextSize sets the current grafPort's type size (thePort^.txSize) to the given number of points. Any size may be specified, but the result will look best if the Font Manager has the font in that size (otherwise it will scale a size it does have). The next best result will occur if the given size is an even multiple of a size available for the font. If Ø is specified, the Font Manager will choose one of the available sizes -- whichever is closest to the system font size. The initial txSize setting is Ø.

```
PROCEDURE SpaceExtra (extra: INTEGER);
```

SpaceExtra sets the current grafPort's spExtra field, which specifies the number of pixels by which to widen each space in a line of text. This is useful when text is being fully justified (that is, aligned with both a left and a right margin). Consider, for example, a line that contains three spaces; if there would normally be six pixels between the end of the line and the right margin, you would call SpaceExtra(2) to print the line with full justification. The initial spExtra setting is Ø.

(hand)

> SpaceExtra will also take a negative argument, but be careful not to narrow spaces so much that the text is unreadable.

```
PROCEDURE DrawChar (ch: CHAR);
```

DrawChar places the given character to the right of the pen location, with the left end of its base line at the pen's location, and advances the pen accordingly. If the character is not in the font, the font's missing symbol is drawn.

```
PROCEDURE DrawString (s: Str255);
```

DrawString performs consecutive calls to DrawChar for each character in the supplied string; the string is placed beginning at the current pen location and extending right. No formatting (carriage returns, line feeds, etc.) is performed by QuickDraw. The pen location ends up to the right of the last character in the string.

```
PROCEDURE DrawText (textBuf: QDPtr; firstByte,byteCount: INTEGER);
```

DrawText draws text from an arbitrary structure in memory specified by textBuf, starting firstByte bytes into the structure and continuing for byteCount bytes. The string of text is placed beginning at the current pen location and extending right. No formatting (carriage returns, line feeds, etc.) is performed by QuickDraw. The pen location ends up to the right of the last character in the string.

```
FUNCTION CharWidth (ch: CHAR) : INTEGER;
```

CharWidth returns the value that will be added to the pen horizontal coordinate if the specified character is drawn. CharWidth includes the effects of the stylistic variations set with TextFace; if you change these after determining the character width but before actually drawing the character, the predetermined width may not be correct. If the character is a space, CharWidth also includes the effect of SpaceExtra.

```
FUNCTION StringWidth (s: Str255) : INTEGER;
```

StringWidth returns the width of the given text string, which it calculates by adding the CharWidths of all the characters in the string (see above). This value will be added to the pen horizontal coordinate if the specified string is drawn.

```
FUNCTION TextWidth (textBuf: QDPtr; firstByte,byteCount: INTEGER) :
          INTEGER;
```

TextWidth returns the width of the text stored in the arbitrary structure in memory specified by textBuf, starting firstByte bytes into the structure and continuing for byteCount bytes. It calculates the width by adding the CharWidths of all the characters in the text. (See CharWidth, above.)

```
PROCEDURE GetFontInfo (VAR info: FontInfo);
```

GetFontInfo returns the following information about the current grafPort's character font, taking into consideration the style and size in which the characters will be drawn: the ascent, descent, maximum character width (the greatest distance the pen will move when a character is drawn), and leading (the vertical distance between the descent line and the ascent line below it), all in pixels. The FontInfo data structure is defined as:

```
TYPE FontInfo = RECORD
                  ascent:  INTEGER;
                  descent: INTEGER;
                  widMax:  INTEGER;
                  leading: INTEGER
                END;
```

Drawing in Color

These routines will enable applications to do color drawing in the future when Apple supports color output devices for the Macintosh. All nonwhite colors will appear as black on black-and-white output devices.

```
PROCEDURE ForeColor (color: LongInt);
```

ForeColor sets the foreground color for all drawing in the current grafPort (^thePort.fgColor) to the given color. The following standard colors are predefined: blackColor, whiteColor, redColor, greenColor, blueColor, cyanColor, magentaColor, and yellowColor. The initial foreground color is blackColor.

```
PROCEDURE BackColor (color: LongInt);
```

BackColor sets the background color for all drawing in the current grafPort (^thePort.bkColor) to the given color. Eight standard colors are predefined (see ForeColor above). The initial background color is whiteColor.

```
PROCEDURE ColorBit (whichBit: INTEGER);
```

ColorBit is called by printing software for a color printer, or other color-imaging software, to set the current grafPort's colrBit field to whichBit; this tells QuickDraw which plane of the color picture to draw into. QuickDraw will draw into the plane corresponding to bit number whichBit. Since QuickDraw can support output devices that have up to 32 bits of color information per pixel, the possible range of values for whichBit is Ø through 31. The initial value of the colrBit field is Ø.

Calculations with Rectangles

Calculation routines are independent of the current coordinate system; a calculation will operate the same regardless of which grafPort is active.

(hand)

> Remember that if the parameters to one of the calculation routines were defined in different grafPorts, you must first adjust them to be in the same coordinate system. If you do not adjust them, the result returned by the routine may be different from what you see on the screen. To adjust to a common coordinate system, see LocalToGlobal and GlobalToLocal under "Calculations with Points" below.

```
PROCEDURE SetRect (VAR r: Rect; left,top,right,bottom: INTEGER);
```

SetRect assigns the four boundary coordinates to the rectangle. The result is a rectangle with coordinates (left,top,right,bottom).

This procedure is supplied as a utility to help you shorten your program text. If you want a more readable text at the expense of length, you can assign integers (or points) directly into the rectangle's fields. There is no significant code size or execution speed advantage to either method; one's just easier to write, and the other's easier to read.

```
PROCEDURE OffsetRect (VAR r: Rect; dh,dv: INTEGER);
```

OffsetRect moves the rectangle by adding dh to each horizontal coordinate and dv to each vertical coordinate. If dh and dv are

positive, the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The rectangle retains its shape and size; it's merely moved on the coordinate plane. This does not affect the screen unless you subsequently call a routine to draw within the rectangle.

```
PROCEDURE InsetRect (VAR r: Rect; dh,dv: INTEGER);
```

InsetRect shrinks or expands the rectangle. The left and right sides are moved in by the amount specified by dh; the top and bottom are moved towards the center by the amount specified by dv. If dh or dv is negative, the appropriate pair of sides is moved outwards instead of inwards. The effect is to alter the size by 2*dh horizontally and 2*dv vertically, with the rectangle remaining centered in the same place on the coordinate plane.

If the resulting width or height becomes less than 1, the rectangle is set to the empty rectangle (Ø,Ø,Ø,Ø).

```
FUNCTION SectRect (srcRectA,srcRectB: Rect; VAR dstRect: Rect) :
          BOOLEAN;
```

SectRect calculates the rectangle that is the intersection of the two input rectangles, and returns TRUE if they indeed intersect or FALSE if they do not. Rectangles that "touch" at a line or a point are not considered intersecting, because their intersection rectangle (really, in this case, an intersection line or point) does not enclose any bits on the bitMap.

If the rectangles do not intersect, the destination rectangle is set to (Ø,Ø,Ø,Ø). SectRect works correctly even if one of the source rectangles is also the destination.

```
PROCEDURE UnionRect (srcRectA,srcRectB: Rect; VAR dstRect: Rect);
```

UnionRect calculates the smallest rectangle which encloses both input rectangles. It works correctly even if one of the source rectangles is also the destination.

```
FUNCTION PtInRect (pt: Point; r: Rect) : BOOLEAN;
```

PtInRect determines whether the pixel below and to the right of the given coordinate point is enclosed in the specified rectangle, and returns TRUE if so or FALSE if not.

```
PROCEDURE Pt2Rect (ptA,ptB: Point; VAR: dstRect: Rect);
```

Pt2Rect returns the smallest rectangle which encloses the two input points.

```
PROCEDURE PtToAngle (r: Rect; pt: Point; VAR angle: INTEGER);
```

PtToAngle calculates an integer angle between a line from the center of the rectangle to the given point and a line from the center of the rectangle pointing straight up (12 o'clock high). The angle is in degrees from Ø to 359, measured clockwise from 12 o'clock, with 9Ø degrees at 3 o'clock, 18Ø at 6 o'clock, and 27Ø at 9 o'clock. Other angles are measured relative to the rectangle: If the line to the given point goes through the top right corner of the rectangle, the angle returned is 45 degrees, even if the rectangle is not square; if it goes through the bottom right corner, the angle is 135 degrees, and so on (see Figure 18).

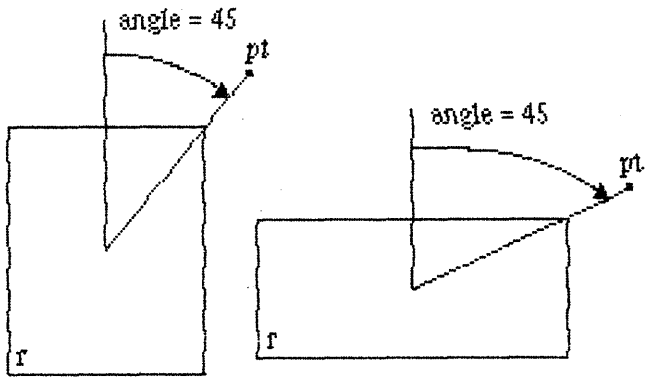


Figure 18. PtToAngle

The angle returned might be used as input to one of the procedures that manipulate arcs and wedges, as described below under "Graphic Operations on Arcs and Wedges".

```
FUNCTION EqualRect (rectA,rectB: Rect) : BOOLEAN;
```

EqualRect compares the two rectangles and returns TRUE if they are equal or FALSE if not. The two rectangles must have identical boundary coordinates to be considered equal.

```
FUNCTION EmptyRect (r: Rect) : BOOLEAN;
```

EmptyRect returns TRUE if the given rectangle is an empty rectangle or FALSE if not. A rectangle is considered empty if the bottom coordinate is equal to or less than the top or the right coordinate is equal to or less than the left.

Graphic Operations on Rectangles

These procedures perform graphic operations on rectangles. See also ScrollRect under "Bit Transfer Operations".

```
PROCEDURE FrameRect (r: Rect);
```

FrameRect draws a hollow outline just inside the specified rectangle, using the current grafPort's pen pattern, mode, and size. The outline is as wide as the pen width and as tall as the pen height. It is drawn with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

If a region is open and being formed, the outside outline of the new rectangle is mathematically added to the region's boundary.

```
PROCEDURE PaintRect (r: Rect);
```

PaintRect paints the specified rectangle with the current grafPort's pen pattern and mode. The rectangle on the bitMap is filled with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

```
PROCEDURE EraseRect (r: Rect);
```

EraseRect paints the specified rectangle with the current grafPort's background pattern bkPat (in patCopy mode). The grafPort's pnPat and pnMode are ignored; the pen location is not changed.

```
PROCEDURE InvertRect (r: Rect);
```

InvertRect inverts the pixels enclosed by the specified rectangle: every white pixel becomes black and every black pixel becomes white. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

```
PROCEDURE FillRect (r: Rect; pat: Pattern);
```

FillRect fills the specified rectangle with the given pattern (in patCopy mode). The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

Graphic Operations on Ovals

Ovals are drawn inside rectangles that you specify. If the rectangle you specify is square, QuickDraw draws a circle.

```
PROCEDURE FrameOval (r: Rect);
```

FrameOval draws a hollow outline just inside the oval that fits inside the specified rectangle, using the current grafPort's pen pattern, mode, and size. The outline is as wide as the pen width and as tall as the pen height. It is drawn with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

If a region is open and being formed, the outside outline of the new oval is mathematically added to the region's boundary.

```
PROCEDURE PaintOval (r: Rect);
```

PaintOval paints an oval just inside the specified rectangle with the current grafPort's pen pattern and mode. The oval on the bitMap is filled with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

```
PROCEDURE EraseOval (r: Rect);
```

EraseOval paints an oval just inside the specified rectangle with the current grafPort's background pattern bkPat (in patCopy mode). The grafPort's pnPat and pnMode are ignored; the pen location is not changed.

```
PROCEDURE InvertOval (r: Rect);
```

InvertOval inverts the pixels enclosed by an oval just inside the specified rectangle: every white pixel becomes black and every black pixel becomes white. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

```
PROCEDURE FillOval (r: Rect; pat: Pattern);
```

FillOval fills an oval just inside the specified rectangle with the given pattern (in patCopy mode). The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

Graphic Operations on Rounded-Corner Rectangles

PROCEDURE FrameRoundRect (r: Rect; ovalWidth,ovalHeight: INTEGER);

FrameRoundRect draws a hollow outline just inside the specified rounded-corner rectangle, using the current grafPort's pen pattern, mode, and size. OvalWidth and ovalHeight specify the diameters of curvature for the corners (see Figure 19). The outline is as wide as the pen width and as tall as the pen height. It is drawn with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

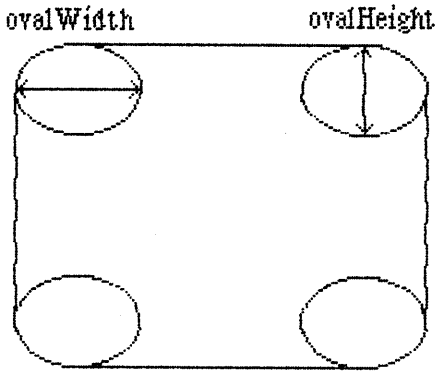


Figure 19. Rounded-Corner Rectangle

If a region is open and being formed, the outside outline of the new rounded-corner rectangle is mathematically added to the region's boundary.

PROCEDURE PaintRoundRect (r: Rect; ovalWidth,ovalHeight: INTEGER);

PaintRoundRect paints the specified rounded-corner rectangle with the current grafPort's pen pattern and mode. OvalWidth and ovalHeight specify the diameters of curvature for the corners. The rounded-corner rectangle on the bitMap is filled with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

PROCEDURE EraseRoundRect (r: Rect; ovalWidth,ovalHeight: INTEGER);

EraseRoundRect paints the specified rounded-corner rectangle with the current grafPort's background pattern bkPat (in patCopy mode).

OvalWidth and ovalHeight specify the diameters of curvature for the corners. The grafPort's pnPat and pnMode are ignored; the pen location is not changed.

```
PROCEDURE InvertRoundRect (r: Rect; ovalWidth,ovalHeight: INTEGER);
```

InvertRoundRect inverts the pixels enclosed by the specified rounded-corner rectangle: every white pixel becomes black and every black pixel becomes white. OvalWidth and ovalHeight specify the diameters of curvature for the corners. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

```
PROCEDURE FillRoundRect (r: Rect; ovalWidth,ovalHeight: INTEGER; pat:
          Pattern);
```

FillRoundRect fills the specified rounded-corner rectangle with the given pattern (in patCopy mode). OvalWidth and ovalHeight specify the diameters of curvature for the corners. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

Graphic Operations on Arcs and Wedges

These procedures perform graphic operations on arcs and wedge-shaped sections of ovals. See also PtToAngle under "Calculations with Rectangles".

```
PROCEDURE FrameArc (r: Rect; startAngle,arcAngle: INTEGER);
```

FrameArc draws an arc of the oval that fits inside the specified rectangle, using the current grafPort's pen pattern, mode, and size. StartAngle indicates where the arc begins and is treated mod 360. ArcAngle defines the extent of the arc. The angles are given in positive or negative degrees; a positive angle goes clockwise, while a negative angle goes counterclockwise. Zero degrees is at 12 o'clock high, 90 (or -270) is at 3 o'clock, 180 (or -180) is at 6 o'çlock, and 270 (or -90) is at 9 o'clock. Other angles are measured relative to the enclosing rectangle: a line from the center of the rectangle through its top right corner is at 45 degrees, even if the rectangle is not square; a line through the bottom right corner is at 135 degrees, and so on (see Figure 20).

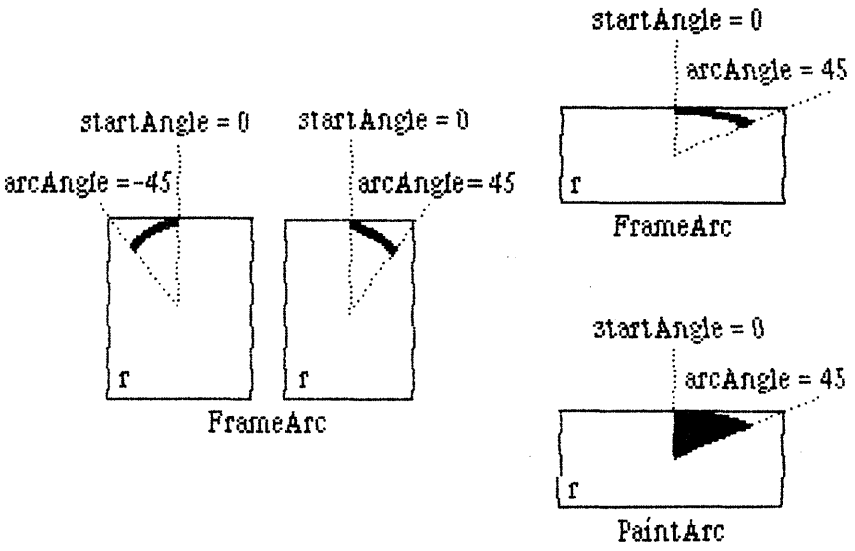


Figure 2Ø. Operations on Arcs and Wedges

The arc is as wide as the pen width and as tall as the pen height. It is drawn with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

(eye)

FrameArc differs from other QuickDraw procedures that frame shapes in that the arc is not mathematically added to the boundary of a region that is open and being formed.

```
PROCEDURE PaintArc (r: Rect; startAngle,arcAngle: INTEGER);
```

PaintArc paints a wedge of the oval just inside the specified rectangle with the current grafPort's pen pattern and mode. StartAngle and arcAngle define the arc of the wedge as in FrameArc. The wedge on the bitMap is filled with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

```
PROCEDURE EraseArc (r: Rect; startAngle,arcAngle: INTEGER);
```

EraseArc paints a wedge of the oval just inside the specified rectangle with the current grafPort's background pattern bkPat (in patCopy mode). StartAngle and arcAngle define the arc of the wedge as in FrameArc. The grafPort's pnPat and pnMode are ignored; the pen location is not changed.

```
PROCEDURE InvertArc (r: Rect; startAngle,arcAngle: INTEGER);
```

InvertArc inverts the pixels enclosed by a wedge of the oval just inside the specified rectangle: every white pixel becomes black and every black pixel becomes white. StartAngle and arcAngle define the arc of the wedge as in FrameArc. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

```
PROCEDURE FillArc (r: Rect; startAngle,arcAngle: INTEGER; pat:
          Pattern);
```

FillArc fills a wedge of the oval just inside the specified rectangle with the given pattern (in patCopy mode). StartAngle and arcAngle define the arc of the wedge as in FrameArc. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

Calculations with Regions

(hand)
Remember that if the parameters to one of the calculation routines were defined in different grafPorts, you must first adjust them to be in the same coordinate system. If you do not adjust them, the result returned by the routine may be different from what you see on the screen. To adjust to a common coordinate system, see LocaltoGlobal and GlobalToLocal under "Calculations with Points" below.

```
FUNCTION NewRgn : RgnHandle;
```

NewRgn allocates space for a new, dynamic, variable-size region, initializes it to the empty region (Ø,Ø,Ø,Ø), and returns a handle to the new region. Only this function creates new regions; all other procedures just alter the size and shape of regions you create. OpenPort calls NewRgn to allocate space for the port's visRgn and clipRgn.

(eye)
Except when using visRgn or clipRgn, you MUST call NewRgn before specifying a region's handle in any drawing or calculation procedure.

(eye)
Never refer to a region without using its handle.

```
PROCEDURE DisposeRgn (rgn: RgnHandle);
```

DisposeRgn deallocates space for the region whose handle is supplied, and returns the memory used by the region to the free memory pool. Use

this only after you are completely through with a temporary region.

(eye)
Never use a region once you have deallocated it, or you will risk being hung by dangling pointers!

PROCEDURE CopyRgn (srcRgn,dstRgn: RgnHandle);

CopyRgn copies the mathematical structure of srcRgn into dstRgn; that is, it makes a duplicate copy of srcRgn. Once this is done, srcRgn may be altered (or even disposed of) without affecting dstRgn. COPYRGN DOES NOT CREATE THE DESTINATION REGION: you must use NewRgn to create the dstRgn before you call CopyRgn.

PROCEDURE SetEmptyRgn (rgn: RgnHandle);

SetEmptyRgn destroys the previous structure of the given region, then sets the new structure to the empty region (Ø,Ø,Ø,Ø).

PROCEDURE SetRectRgn (rgn: RgnHandle; left,top,right,bottom: INTEGER);

SetRectRgn destroys the previous structure of the given region, then sets the new structure to the rectangle specified by left, top, right, and bottom.

If the specified rectangle is empty (i.e., left>=right or top>=bottom), the region is set to the empty region (Ø,Ø,Ø,Ø).

PROCEDURE RectRgn (rgn: RgnHandle; r: Rect);

RectRgn destroys the previous structure of the given region, then sets the new structure to the rectangle specified by r. This is operationally synonymous with SetRectRgn, except the input rectangle is defined by a rectangle rather than by four boundary coordinates.

PROCEDURE OpenRgn;

OpenRgn tells QuickDraw to allocate temporary space and start saving lines and framed shapes for later processing as a region definition. While a region is open, all calls to Line, LineTo, and the procedures that draw framed shapes (except arcs) affect the outline of the region. Only the line endpoints and shape boundaries affect the region definition; the pen mode, pattern, and size do not affect it. In fact, OpenRgn calls HidePen, so no drawing occurs on the screen while the region is open (unless you called ShowPen just after OpenRgn, or you called ShowPen previously without balancing it by a call to HidePen). Since the pen hangs below and to the right of the pen location, drawing lines with even the smallest pen will change bits that lie outside the region you define.

The outline of a region is mathematically defined and infinitely thin, and separates the bitMap into two groups of bits: those within the region and those outside it. A region should consist of one or more closed loops. Each framed shape itself constitutes a loop. Any lines drawn with Line or LineTo should connect with each other or with a framed shape. Even though the on-screen presentation of a region is clipped, the definition of a region is not; you can define a region anywhere on the coordinate plane with complete disregard for the location of various grafPort entities on that plane.

When a region is open, the current grafPort's rgnSave field contains a handle to information related to the region definition. If you want to temporarily disable the collection of lines and shapes, you can save the current value of this field, set the field to NIL, and later restore the saved value to resume the region definition.

(eye)
Do not call OpenRgn while another region is already open. All open regions but the most recent will behave strangely.

```
PROCEDURE CloseRgn (dstRgn: RgnHandle);
```

CloseRgn stops the collection of lines and framed shapes, organizes them into a region definition, and saves the resulting region into the region indicated by dstRgn. You should perform one and only one CloseRgn for every OpenRgn. CloseRgn calls ShowPen, balancing the HidePen call made by OpenRgn.

Here's an example of how to create and open a region, define a barbell shape, close the region, and draw it:

```
barbell := NewRgn;                      {make a new region}
OpenRgn;                                {begin collecting stuff}
   SetRect(tempRect,2Ø,2Ø,3Ø,5Ø);       {form the left weight}
   FrameOval(tempRect);
   SetRect(tempRect,3Ø,3Ø,8Ø,4Ø);       {form the bar}
   FrameRect(tempRect);
   SetRect(tempRect,8Ø,2Ø,9Ø,5Ø);       {form the right weight}
   FrameOval(tempRect);
CloseRgn(barbell);                      {we're done; save in barbell}
FillRgn(barbell,black);                 {draw it on the screen}
DisposeRgn(barbell);                    {we don't need you anymore...}
```

```
PROCEDURE OffsetRgn (rgn: RgnHandle; dh,dv: INTEGER);
```

OffsetRgn moves the region on the coordinate plane, a distance of dh horizontally and dv vertically. This does not affect the screen unless you subsequently call a routine to draw the region. If dh and dv are positive, the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The region retains its size and shape.

(hand)
OffsetRgn is an especially efficient operation, because most of the data defining a region is stored relative to rgnBBox and so isn't actually changed by OffsetRgn.

```
PROCEDURE InsetRgn (rgn: RgnHandle; dh,dv: INTEGER);
```

InsetRgn shrinks or expands the region. All points on the region boundary are moved inwards a distance of dv vertically and dh horizontally; if dh or dv is negative, the points are moved outwards in that direction. InsetRgn leaves the region "centered" at the same position, but moves the outline in (for positive values of dh and dv) or out (for negative values of dh and dv). InsetRgn of a rectangular region works just like InsetRect.

```
PROCEDURE SectRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);
```

SectRgn calculates the intersection of two regions and places the intersection in a third region. THIS DOES NOT CREATE THE DESTINATION REGION: you must use NewRgn to create the dstRgn before you call SectRgn. The dstRgn can be one of the source regions, if desired.

If the regions do not intersect, or one of the regions is empty, the destination is set to the empty region (Ø,Ø,Ø,Ø).

```
PROCEDURE UnionRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);
```

UnionRgn calculates the union of two regions and places the union in a third region. THIS DOES NOT CREATE THE DESTINATION REGION: you must use NewRgn to create the dstRgn before you call UnionRgn. The dstRgn can be one of the source regions, if desired.

If both regions are empty, the destination is set to the empty region (Ø,Ø,Ø,Ø).

```
PROCEDURE DiffRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);
```

DiffRgn subtracts srcRgnB from srcRgnA and places the difference in a third region. THIS DOES NOT CREATE THE DESTINATION REGION: you must use NewRgn to create the dstRgn before you call DiffRgn. The dstRgn can be one of the source regions, if desired.

If the first source region is empty, the destination is set to the empty region (Ø,Ø,Ø,Ø).

```
PROCEDURE XorRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);
```

XorRgn calculates the difference between the union and the intersection of two regions and places the result in a third region. THIS DOES NOT

CREATE THE DESTINATION REGION: you must use NewRgn to create the dstRgn before you call XorRgn. The dstRgn can be one of the source regions, if desired.

If the regions are coincident, the destination is set to the empty region (Ø,Ø,Ø,Ø).

```
FUNCTION PtInRgn (pt: Point; rgn: RgnHandle) : BOOLEAN;
```

PtInRgn checks whether the pixel below and to the right of the given coordinate point is within the specified region, and returns TRUE if so or FALSE if not.

```
FUNCTION RectInRgn (r: Rect; rgn: RgnHandle) : BOOLEAN;
```

RectInRgn checks whether the given rectangle intersects the specified region, and returns TRUE if the intersection encloses at least one bit or FALSE if not.

```
FUNCTION EqualRgn (rgnA,rgnB: RgnHandle) : BOOLEAN;
```

EqualRgn compares the two regions and returns TRUE if they are equal or FALSE if not. The two regions must have identical sizes, shapes, and locations to be considered equal. Any two empty regions are always equal.

```
FUNCTION EmptyRgn (rgn: RgnHandle) : BOOLEAN;
```

EmptyRgn returns TRUE if the region is an empty region or FALSE if not. Some of the circumstances in which an empty region can be created are: a NewRgn call; a CopyRgn of an empty region; a SetRectRgn or RectRgn with an empty rectangle as an argument; CloseRgn without a previous OpenRgn or with no drawing after an OpenRgn; OffsetRgn of an empty region; InsetRgn with an empty region or too large an inset; SectRgn of nonintersecting regions; UnionRgn of two empty regions; and DiffRgn or XorRgn of two identical or nonintersecting regions.

Graphic Operations on Regions

These routines all depend on the coordinate system of the current grafPort. If a region is drawn in a different grafPort than the one in which it was defined, it may not appear in the proper position inside the port.

```
PROCEDURE FrameRgn (rgn: RgnHandle);
```

FrameRgn draws a hollow outline just inside the specified region, using the current grafPort's pen pattern, mode, and size. The outline is as

wide as the pen width and as tall as the pen height; under no circumstances will the frame go outside the region boundary. The pen location is not changed by this procedure.

If a region is open and being formed, the outside outline of the region being framed is mathematically added to that region's boundary.

PROCEDURE PaintRgn (rgn: RgnHandle);

PaintRgn paints the specified region with the current grafPort's pen pattern and pen mode. The region on the bitMap is filled with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

PROCEDURE EraseRgn (rgn: RgnHandle);

EraseRgn paints the specified region with the current grafPort's background pattern bkPat (in patCopy mode). The grafPort's pnPat and pnMode are ignored; the pen location is not changed.

PROCEDURE InvertRgn (rgn: RgnHandle);

InvertRgn inverts the pixels enclosed by the specified region: every white pixel becomes black and every black pixel becomes white. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

PROCEDURE FillRgn (rgn: RgnHandle; pat: Pattern);

FillRgn fills the specified region with the given pattern (in patCopy mode). The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

Bit Transfer Operations

PROCEDURE ScrollRect (r: Rect; dh,dv: INTEGER; updateRgn: RgnHandle);

ScrollRect shifts ("scrolls") those bits inside the intersection of the specified rectangle, visRgn, clipRgn, portRect, and portBits.bounds. The bits are shifted a distance of dh horizontally and dv vertically. The positive directions are to the right and down. No other bits are affected. Bits that are shifted out of the scroll area are lost; they are neither placed outside the area nor saved. The grafPort's background pattern bkPat fills the space created by the scroll. In addition, updateRgn is changed to the area filled with bkPat (see Figure 21).

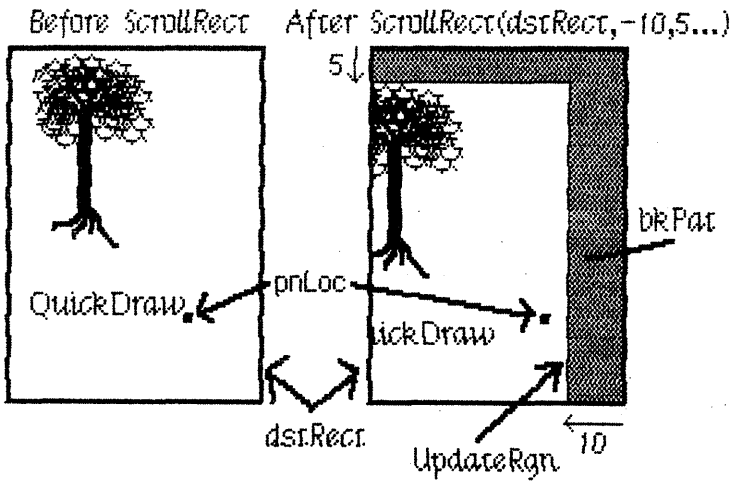


Figure 21. Scrolling

Figure 21 shows that the pen location after a ScrollRect is in a different position relative to what was scrolled in the rectangle. The entire scrolled item has been moved to different coordinates. To restore it to its coordinates before the ScrollRect, you can use the SetOrigin procedure. For example, suppose the dstRect here is the portRect of the grafPort and its top left corner is at (95,12Ø). SetOrigin(1Ø5,115) will offset the coordinate system to compensate for the scroll. Since the clipRgn and pen location are not offset, they move down and to the left.

```
PROCEDURE CopyBits (srcBits,dstBits: BitMap; srcRect,dstRect: Rect;
          mode: INTEGER; maskRgn: RgnHandle);
```

CopyBits transfers a bit image between any two bitMaps and clips the result to the area specified by the maskRgn parameter. The transfer may be performed in any of the eight source transfer modes. The result is always clipped to the maskRgn and the boundary rectangle of the destination bitMap; if the destination bitMap is the current grafPort's portBits, it is also clipped to the intersection of the grafPort's clipRgn and visRgn. If you do not want to clip to a maskRgn, just pass NIL for the maskRgn parameter.

The dstRect and maskRgn coordinates are in terms of the dstBits.bounds coordinate system, and the srcRect coordinates are in terms of the srcBits.bounds coordinates.

The bits enclosed by the source rectangle are transferred into the destination rectangle according to the rules of the chosen mode. The source transfer modes are as follows:

srcCopy	srcXor	notSrcCopy	notSrcXor
srcOr	srcBic	notSrcOr	notSrcBic

The source rectangle is completely aligned with the destination rectangle; if the rectangles are of different sizes, the bit image is expanded or shrunk as necessary to fit the destination rectangle. For example, if the bit image is a circle in a square source rectangle, and the destination rectangle is not square, the bit image appears as an oval in the destination (see Figure 22).

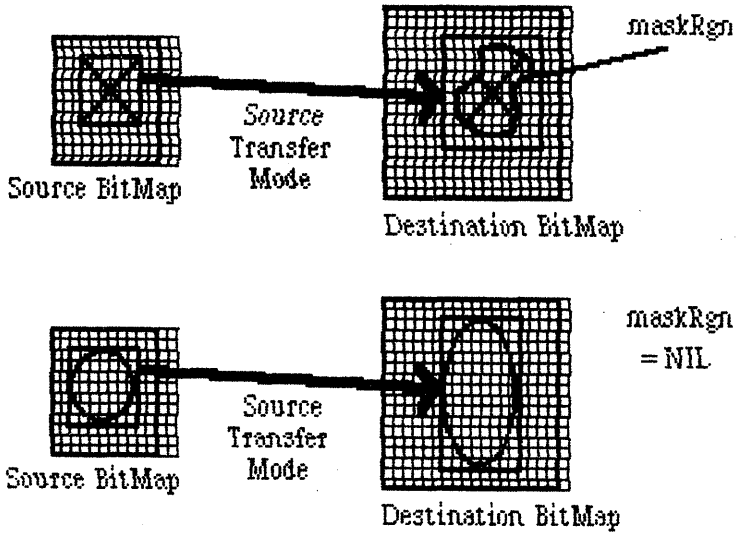


Figure 22. Operation of CopyBits

Pictures

FUNCTION OpenPicture (picFrame: Rect) : PicHandle;

OpenPicture returns a handle to a new picture which has the given rectangle as its picture frame, and tells QuickDraw to start saving as the picture definition all calls to drawing routines and all picture comments (if any).

OpenPicture calls HidePen, so no drawing occurs on the screen while the picture is open (unless you call ShowPen just after OpenPicture, or you called ShowPen previously without balancing it by a call to HidePen).

When a picture is open, the current grafPort's picSave field contains a handle to information related to the picture definition. If you want to temporarily disable the collection of routine calls and picture comments, you can save the current value of this field, set the field to NIL, and later restore the saved value to resume the picture definition.

(eye)

Do not call OpenPicture while another picture is already open.

PROCEDURE ClosePicture;

ClosePicture tells QuickDraw to stop saving routine calls and picture comments as the definition of the currently open picture. You should perform one and only one ClosePicture for every OpenPicture. ClosePicture calls ShowPen, balancing the HidePen call made by OpenPicture.

PROCEDURE PicComment (kind,dataSize: INTEGER; dataHandle: QDHandle);

PicComment inserts the specified comment into the definition of the currently open picture. Kind identifies the type of comment. DataHandle is a handle to additional data if desired, and dataSize is the size of that data in bytes. If there is no additional data for the comment, dataHandle should be NIL and dataSize should be Ø. The application that processes the comment must include a procedure to do the processing and store a pointer to the procedure in the data structure pointed to by the grafProcs field of the grafPort (see "Customizing QuickDraw Operations").

PROCEDURE DrawPicture (myPicture: PicHandle; dstRect: Rect);

DrawPicture draws the given picture to scale in dstRect, expanding or shrinking it as necessary to align the borders of the picture frame with dstRect. DrawPicture passes any picture comments to the procedure accessed indirectly through the grafProcs field of the grafPort (see PicComment above).

PROCEDURE KillPicture (myPicture: PicHandle);

KillPicture deallocates space for the picture whose handle is supplied, and returns the memory used by the picture to the free memory pool. Use this only when you are completely through with a picture.

Calculations with Polygons

FUNCTION OpenPoly : PolyHandle;

OpenPoly returns a handle to a new polygon and tells QuickDraw to start saving the polygon definition as specified by calls to line-drawing routines. While a polygon is open, all calls to Line and LineTo affect the outline of the polygon. Only the line endpoints affect the polygon definition; the pen mode, pattern, and size do not affect it. In fact, OpenPoly calls HidePen, so no drawing occurs on the screen while the polygon is open (unless you call ShowPen just after OpenPoly, or you called ShowPen previously without balancing it by a call to HidePen).

A polygon should consist of a sequence of connected lines. Even though the on-screen presentation of a polygon is clipped, the definition of a polygon is not; you can define a polygon anywhere on the coordinate plane with complete disregard for the location of various grafPort entities on that plane.

When a polygon is open, the current grafPort's polySave field contains a handle to information related to the polygon definition. If you want to temporarily disable the polygon definition, you can save the current value of this field, set the field to NIL, and later restore the saved value to resume the polygon definition.

(eye)
Do not call OpenPoly while another polygon is already open.

PROCEDURE ClosePoly;

ClosePoly tells QuickDraw to stop saving the definition of the currently open polygon and computes the polyBBox rectangle. You should perform one and only one ClosePoly for every OpenPoly. ClosePoly calls ShowPen, balancing the HidePen call made by OpenPoly.

Here's an example of how to open a polygon, define it as a triangle, close it, and draw it:

```
triPoly := OpenPoly;        {save handle and begin collecting stuff}
  MoveTo(3ØØ,1ØØ);          { move to first point and }
  LineTo(4ØØ,2ØØ);          {          form          }
  LineTo(2ØØ,2ØØ);          {           the          }
  LineTo(3ØØ,1ØØ);          {       triangle         }
ClosePoly;                  {stop collecting stuff}
FillPoly(triPoly,gray);     {draw it on the screen}
KillPoly(triPoly);          {we're all done}
```

PROCEDURE KillPoly (poly: PolyHandle);

KillPoly deallocates space for the polygon whose handle is supplied, and returns the memory used by the polygon to the free memory pool. Use this only after you are completely through with a polygon.

PROCEDURE OffsetPoly (poly: PolyHandle; dh,dv: INTEGER);

OffsetPoly moves the polygon on the coordinate plane, a distance of dh horizontally and dv vertically. This does not affect the screen unless you subsequently call a routine to draw the polygon. If dh and dv are positive, the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The polygon retains its shape and size.

(hand)
OffsetPoly is an especially efficient operation, because the data defining a polygon is stored relative to polyStart and so isn't actually changed by OffsetPoly.

Graphic Operations on Polygons

PROCEDURE FramePoly (poly: PolyHandle);

FramePoly plays back the line-drawing routine calls that define the given polygon, using the current grafPort's pen pattern, mode, and size. The pen will hang below and to the right of each point on the boundary of the polygon; thus, the polygon drawn will extend beyond the right and bottom edges of poly^^.polyBBox by the pen width and pen height, respectively. All other graphic operations occur strictly within the boundary of the polygon, as for other shapes. You can see this difference in Figure 23, where each of the polygons is shown with its polyBBox.

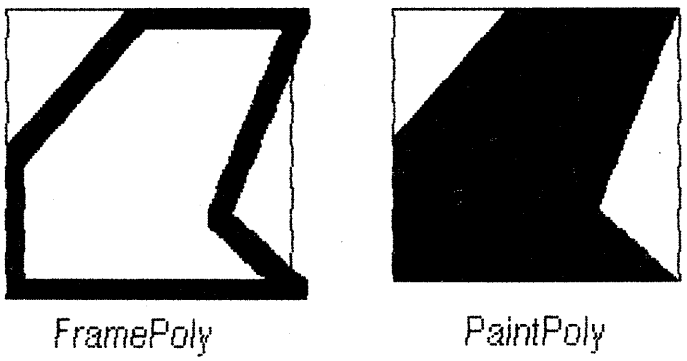


Figure 23. Drawing Polygons

If a polygon is open and being formed, FramePoly affects the outline of the polygon just as if the line-drawing routines themselves had been called. If a region is open and being formed, the outside outline of the polygon being framed is mathematically added to the region's boundary.

PROCEDURE PaintPoly (poly: PolyHandle);

PaintPoly paints the specified polygon with the current grafPort's pen pattern and pen mode. The polygon on the bitMap is filled with the pnPat, according to the pattern transfer mode specified by pnMode. The

pen location is not changed by this procedure.

```
PROCEDURE ErasePoly (poly: PolyHandle);
```

ErasePoly paints the specified polygon with the current grafPort's background pattern bkPat (in patCopy mode). The pnPat and pnMode are ignored; the pen location is not changed.

```
PROCEDURE InvertPoly (poly: PolyHandle);
```

InvertPoly inverts the pixels enclosed by the specified polygon: every white pixel becomes black and every black pixel becomes white. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

```
PROCEDURE FillPoly (poly: PolyHandle; pat: Pattern);
```

FillPoly fills the specified polygon with the given pattern (in patCopy mode). The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

Calculations with Points

```
PROCEDURE AddPt (srcPt: Point; VAR dstPt: Point);
```

AddPt adds the coordinates of srcPt to the coordinates of dstPt, and returns the result in dstPt.

```
PROCEDURE SubPt (srcPt: Point; VAR dstPt: Point);
```

SubPt subtracts the coordinates of srcPt from the coordinates of dstPt, and returns the result in dstPt.

```
PROCEDURE SetPt (VAR pt: Point; h,v: INTEGER);
```

SetPt assigns two integer coordinates to a variable of type Point.

```
FUNCTION EqualPt (ptA,ptB: Point) : BOOLEAN;
```

EqualPt compares the two points and returns true if they are equal or FALSE if not.

```
PROCEDURE LocalToGlobal (VAR pt: Point);
```

LocalToGlobal converts the given point from the current grafPort's local coordinate system into a global coordinate system with the origin (Ø,Ø) at the top left corner of the port's bit image (such as the screen). This global point can then be compared to other global points, or be changed into the local coordinates of another grafPort.

Since a rectangle is defined by two points, you can convert a rectangle into global coordinates by performing two LocalToGlobal calls. You can also convert a rectangle, region, or polygon into global coordinates by calling OffsetRect, OffsetRgn, or OffsetPoly. For examples, see GlobalToLocal below.

```
PROCEDURE GlobalToLocal (VAR pt: Point);
```

GlobalToLocal takes a point expressed in global coordinates (with the top left corner of the bitMap as coordinate (Ø,Ø)) and converts it into the local coordinates of the current grafPort. The global point can be obtained with the LocalToGlobal call (see above). For example, suppose a game draws a "ball" within a rectangle named ballRect, defined in the grafPort named gamePort (as illustrated below in Figure 24). If you want to draw that ball in the grafPort named selectPort, you can calculate the ball's selectPort coordinates like this:

```
SetPort(gamePort);                  {start in origin port}
selectBall := ballRect;             {make a copy to be moved}
LocalToGlobal(selectBall.topLeft);  {put both corners into }
LocalToGlobal(selectBall.botRight); {  global coordinates  }

SetPort(selectPort);                {switch to destination port}
GlobalToLocal(selectBall.topLeft);  {put both corners into     }
GlobalToLocal(selectBall.botRight); {  these local coordinates }
FillOval(selectBall,ballColor);     {now you have the ball!}
```

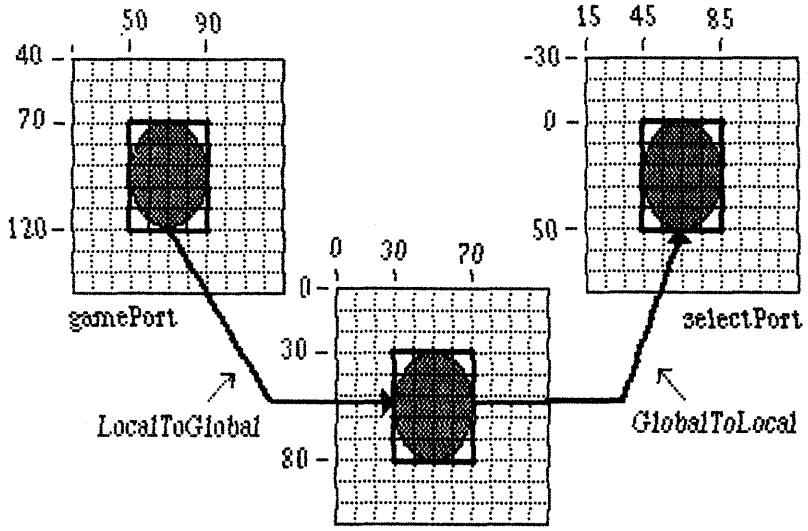


Figure 24. Converting between Coordinate Systems

You can see from Figure 24 that LocalToGlobal and GlobalToLocal simply offset the coordinates of the rectangle by the coordinates of the top left corner of the local grafPort's boundary rectangle. You could also do this with OffsetRect. In fact, the way to convert regions and polygons from one coordinate system to another is with OffsetRgn or OffsetPoly rather than LocalToGlobal and GlobalToLocal. For example, if myRgn were a region enclosed by a rectangle having the same coordinates as ballRect in gamePort, you could convert the region to global coordinates with

```
OffsetRgn(myRgn, -2Ø, -4Ø);
```

and then convert it to the coordinates of the selectPort grafPort with

```
OffsetRgn(myRgn, 15, -3Ø);
```

Miscellaneous Utilities

```
FUNCTION Random : INTEGER;
```

This function returns an integer, uniformly distributed pseudo-random, in the range from -32768 through 32767. The value returned depends on the global variable randSeed, which InitGraf initializes to 1; you can start the sequence over again from where it began by resetting randSeed to 1.

```
FUNCTION GetPixel (h,v: INTEGER) : BOOLEAN;
```

GetPixel looks at the pixel associated with the given coordinate point and returns TRUE if it is black or FALSE if it is white. The selected pixel is immediately below and to the right of the point whose coordinates are given in h and v, in the local coordinates of the current grafPort. There is no guarantee that the specified pixel actually belongs to the port, however; it may have been drawn by a port overlapping the current one. To see if the point indeed belongs to the current port, perform a PtInRgn(pt,thePort^.visRgn).

```
PROCEDURE StuffHex (thingPtr: QDPtr; s: Str255);
```

StuffHex pokes bits (expressed as a string of hexadecimal digits) into any data structure. This is a good way to create cursors, patterns, or bit images to be "stamped" onto the screen with CopyBits. For example,

```
StuffHex(@stripes,'Ø1Ø2Ø4Ø81Ø2Ø4Ø8Ø')
```

places a striped pattern into the pattern variable stripes.

(eye)
There is no range checking on the size of the destination variable. It's easy to overrun the variable and destroy something if you don't know what you're doing.

```
PROCEDURE ScalePt (VAR pt: Point; srcRect,dstRect: Rect);
```

A width and height are passed in pt; the horizontal component of pt is the width, and the vertical component of pt is the height. ScalePt scales these measurements as follows and returns the result in pt: it multiplies the given width by the ratio of dstRect's width to srcRect's width, and multiplies the given height by the ratio of dstRect's height to srcRect's height. In Figure 25, where dstRect's width is twice srcRect's width and its height is three times srcRect's height, the pen width is scaled from 3 to 6 and the pen height is scaled from 2 to 6.

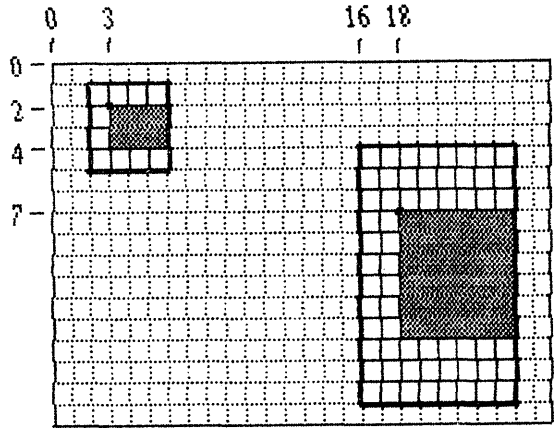


Figure 25. ScalePt and MapPt

PROCEDURE MapPt (VAR pt: Point; srcRect,dstRect: Rect);

Given a point within srcRect, MapPt maps it to a similarly located point within dstRect (that is, to where it would fall if it were part of a drawing being expanded or shrunk to fit dstRect). The result is returned in pt. A corner point of srcRect would be mapped to the corresponding corner point of dstRect, and the center of srcRect to the center of dstRect. In Figure 25 above, the point (3,2) in srcRect is mapped to (18,7) in dstRect. FromRect and dstRect may overlap, and pt need not actually be within srcRect.

(eye)
Remember, if you are going to draw inside the rectangle in dstRect, you will probably also want to scale the pen size accordingly with ScalePt.

PROCEDURE MapRect (VAR r: Rect; srcRect,dstRect: Rect);

Given a rectangle within srcRect, MapRect maps it to a similarly located rectangle within dstRect by calling MapPt to map the top left and bottom right corners of the rectangle. The result is returned in r.

PROCEDURE MapRgn (rgn: RgnHandle; srcRect,dstRect: Rect);

Given a region within srcRect, MapRgn maps it to a similarly located region within dstRect by calling MapPt to map all the points in the region.

```
PROCEDURE MapPoly (poly: PolyHandle; srcRect,dstRect: Rect);
```

Given a polygon within srcRect, MapPoly maps it to a similarly located polygon within dstRect by calling MapPt to map all the points that define the polygon.

CUSTOMIZING QUICKDRAW OPERATIONS

For each shape that QuickDraw knows how to draw, there are procedures that perform these basic graphic operations on the shape: frame, paint, erase, invert, and fill. Those procedures in turn call a low-level drawing routine for the shape. For example, the FrameOval, PaintOval, EraseOval, InvertOval, and FillOval procedures all call a low-level routine that draws the oval. For each type of object QuickDraw can draw, including text and lines, there is a pointer to such a routine. By changing these pointers, you can install your own routines, and either completely override the standard ones or call them after your routines have modified parameters as necessary.

Other low-level routines that you can install in this way are:

- The procedure that does bit transfer and is called by CopyBits.
- The function that measures the width of text and is called by CharWidth, StringWidth, and TextWidth.
- The procedure that processes picture comments and is called by DrawPicture. The standard such procedure ignores picture comments.
- The procedure that saves drawing commands as the definition of a picture, and the one that retrieves them. This enables the application to draw on remote devices, print to the disk, get picture input from the disk, and support large pictures.

The grafProcs field of a grafPort determines which low-level routines are called; if it contains NIL, the standard routines are called, so that all operations in that grafPort are done in the standard ways described in this manual. You can set the grafProcs field to point to a record of pointers to routines. The data type of grafProcs is QDProcsPtr:

```
TYPE QDProcsPtr = ^QDProcs;
     QDProcs    = RECORD
                    textProc:    QDPtr;  {text drawing}
                    lineProc:    QDPtr;  {line drawing}
                    rectProc:    QDPtr;  {rectangle drawing}
                    rRectProc:   QDPtr;  {roundRect drawing}
                    ovalProc:    QDPtr;  {oval drawing}
                    arcProc:     QDPtr;  {arc/wedge drawing}
                    polyProc:    QDPtr;  {polygon drawing}
                    rgnProc:     QDPtr;  {region drawing}
                    bitsProc:    QDPtr;  {bit transfer}
                    commentProc: QDPtr;  {picture comment processing}
                    txMeasProc:  QDPtr;  {text width measurement}
                    getPicProc:  QDPtr;  {picture retrieval}
                    putPicProc:  QDPtr   {picture saving}
                  END;
```

To assist you in setting up a QDProcs record, QuickDraw provides the following procedure:

```
PROCEDURE SetStdProcs (VAR procs: QDProcs);
```

This procedure sets all the fields of the given QDProcs record to point to the standard low-level routines. You can then change the ones you wish to point to your own routines. For example, if your procedure that processes picture comments is named MyComments, you will store @MyComments in the commentProc field of the QDProcs record.

The routines you install must of course have the same calling sequences as the standard routines, which are described below. The standard drawing routines tell which graphic operation to perform from a parameter of type GrafVerb.

```
TYPE GrafVerb = (frame, paint, erase, invert, fill);
```

When the grafVerb is fill, the pattern to use when filling is passed in the fillPat field of the grafPort.

```
PROCEDURE StdText (byteCount: INTEGER; textBuf: QDPtr; numer,denom:
          INTEGER);
```

StdText is the standard low-level routine for drawing text. It draws text from the arbitrary structure in memory specified by textBuf, starting from the first byte and continuing for byteCount bytes. Numer and denom specify the scaling, if any: numer.v over denom.v gives the vertical scaling, and numer.h over denom.h gives the horizontal scaling.

```
PROCEDURE StdLine (newPt: Point);
```

StdLine is the standard low-level routine for drawing a line. It draws a line from the current pen location to the location specified (in

local coordinates) by newPt.

```
PROCEDURE StdRect (verb: GrafVerb; r: Rect);
```

StdRect is the standard low-level routine for drawing a rectangle. It draws the given rectangle according to the specified grafVerb.

```
PROCEDURE StdRRect (verb: GrafVerb; r: Rect; ovalwidth,ovalHeight:
          INTEGER);
```

StdRRect is the standard low-level routine for drawing a rounded-corner rectangle. It draws the given rounded-corner rectangle according to the specified grafVerb. OvalWidth and ovalHeight specify the diameters of curvature for the corners.

```
PROCEDURE StdOval (verb: GrafVerb; r: Rect);
```

StdOval is the standard low-level routine for drawing an oval. It draws an oval inside the given rectangle according to the specified grafVerb.

```
PROCEDURE StdArc (verb: GrafVerb; r: Rect; startAngle,arcAngle:
          INTEGER);
```

StdArc is the standard low-level routine for drawing an arc or a wedge. It draws an arc or wedge of the oval that fits inside the given rectangle. The grafVerb specifies the graphic operation; if it's the frame operation, an arc is drawn; otherwise, a wedge is drawn.

```
PROCEDURE StdPoly (verb: GrafVerb; poly: PolyHandle);
```

StdPoly is the standard low-level routine for drawing a polygon. It draws the given polygon according to the specified grafVerb.

```
PROCEDURE StdRgn (verb: GrafVerb; rgn: RgnHandle);
```

StdRgn is the standard low-level routine for drawing a region. It draws the given region according to the specified grafVerb.

```
PROCEDURE StdBits (VAR srcBits: BitMap; VAR srcRect,dstRect: Rect;
          mode: INTEGER; maskRgn: RgnHandle);
```

StdBits is the standard low-level routine for doing bit transfer. It transfers a bit image between the given bitMap and thePort^.portBits, just as if CopyBits were called with the same parameters and with a destination bitMap equal to thePort^.portBits.

```
PROCEDURE StdComment (kind,dataSize: INTEGER; dataHandle: QDHandle);
```

StdComment is the standard low-level routine for processing a picture comment. Kind identifies the type of comment. DataHandle is a handle to additional data, and dataSize is the size of that data in bytes. If there is no additional data for the command, dataHandle will be NIL and dataSize will be Ø. StdComment simply ignores the comment.

```
FUNCTION StdTxMeas (byteCount: INTEGER; textBuf: QDPtr; VAR
            numer,denom: Point; VAR info: FontInfo) : INTEGER;
```

StdTxMeas is the standard low-level routine for measuring text width. It returns the width of the text stored in the arbitrary structure in memory specified by textBuf, starting with the first byte and continuing for byteCount bytes. Numer and denom specify the scaling as in the StdText procedure; note that StdTxMeas may change them.

```
PROCEDURE StdGetPic (dataPtr: QDPtr; byteCount: INTEGER);
```

StdGetPic is the standard low-level routine for retrieving information from the definition of a picture. It retrieves the next byteCount bytes from the definition of the currently open picture and stores them in the data structure pointed to by dataPtr.

```
PROCEDURE StdPutPic (dataPtr: QDPtr; byteCount: INTEGER);
```

StdPutPic is the standard low-level routine for saving information as the definition of a picture. It saves as the definition of the currently open picture the drawing commands stored in the data structure pointed to by dataPtr, starting with the first byte and continuing for the next byteCount bytes.

USING QUICKDRAW FROM ASSEMBLY LANGUAGE

All Macintosh User Interface Toolbox routines can be called from assembly-language programs as well as from Pascal. When you write an assembly-language program to use these routines, though, you must emulate Pascal's parameter passing and variable transfer protocols.

This section discusses how to use the QuickDraw constants, global variables, data types, procedures, and functions from assembly language.

The primary aid to assembly-language programmers is a file named GRAFTYPES.TEXT. If you use .INCLUDE to include this file when you assemble your program, all the QuickDraw constants, offsets to locations of global variables, and offsets into the fields of structured types will be available in symbolic form.

Constants

QuickDraw constants are stored in the GRAFTYPES.TEXT file, and you can use the constant values symbolically. For example, if you've loaded the effective address of the thePort^.txMode field into address register A2, you can set that field to the srcXor mode with this statement:

```
MOVE.W  #SRCXOR,(A2)
```

To refer to the number of bytes occupied by the QuickDraw global variables, you can use the constant GRAFSIZE. When you call the InitGraf procedure, you must pass a pointer to an area at least that large.

Data Types

Pascal's strong typing ability lets you write Pascal programs without really considering the size of a variable. But in assembly language, you must keep track of the size of every variable. The sizes of the standard Pascal data types are as follows:

Type	Size
INTEGER	Word (2 bytes)
LongInt	Long (4 bytes)
BOOLEAN	Word (2 bytes)
CHAR	Word (2 bytes)
REAL	Long (4 bytes)

INTEGERs and LongInts are in two's complement form; BOOLEANs have their boolean value in bit 8 of the word (the low-order bit of the byte at the same location); CHARs are stored in the high-order byte of the word; and REALs are in the KCS standard format.

The QuickDraw simple data types listed below are constructed out of these fundamental types.

Type	Size
QDPtr	Long (4 bytes)
QDHandle	Long (4 bytes)
Word	Long (4 bytes)
Str255	Page (256 bytes)
Pattern	8 bytes
Bits16	32 bytes

Other data types are constructed as records of variables of the above types. The size of such a type is the sum of the sizes of all the fields in the record; the fields appear in the variable with the first field in the lowest address. For example, consider the data type BitMap, which is defined like this:

```
TYPE BitMap = RECORD
                baseAddr: QDPtr;
                rowBytes: INTEGER;
                bounds:   Rect
              END;
```

This data type would be arranged in memory as seven words: a long for the baseAddr, a word for the rowBytes, and four words for the top, left, right, and bottom parts of the bounds rectangle. To assist you in referring to the fields inside a variable that has a structure like this, the GRAFTYPES.TEXT file defines constants that you can use as offsets into the fields of a structured variable. For example, to move a bitMap's rowBytes value into D3, you would execute the following instruction:

```
MOVE.W  MYBITMAP+ROWBYTES,D3
```

Displacements are given in the GRAFTYPES.TEXT file for all fields of all data types defined by QuickDraw.

To do double indirection, you perform an LEA indirectly to obtain the effective address from the handle. For example, to get at the top coordinate of a region's enclosing rectangle:

```
MOVE.L  MYHANDLE,A1             ; Load handle into A1
MOVE.L  (A1),A1                 ; Use handle to get pointer
MOVE.W  RGNBBOX+TOP(A1),D3      ; Load value using pointer
```

(eye)

For regions (and all other variable-length structures with handles), you must not move the pointer into a register once and just continue to use that pointer; you must do the double indirection each time. Every QuickDraw, Toolbox, or memory management call you make can possibly trigger a heap compaction that renders all pointers to movable heap items (like regions) invalid. The handles will remain valid, but pointers you've obtained through handles can be rendered invalid at any subroutine call or trap in your program.

Global Variables

Global variables are stored in a special section of Macintosh low memory; register A5 always points to this section of memory. The GRAFTYPES.TEXT file defines a constant GRAFGLOB that points to the beginning of the QuickDraw variables in this space, and other constants that point to the individual variables. To access one of the variables, put GRAFGLOB in an address register, sum the constants, and index off of that register. For example, if you want to know the horizontal coordinate of the pen location for the current grafPort, which the global variable thePort points to, you can give the following instructions:

```
MOVE.L  GRAFGLOB(A5),AØ      ; Point to QuickDraw globals
MOVE.L  THEPORT(AØ),A1       ; Get current grafPort
MOVE.W  PNLOC+H(A1),DØ       ; Get thePort^.pnLoc.h
```

Procedures and Functions

To call a QuickDraw procedure or function, you must push all parameters to it on the stack, then JSR to the function or procedure. When you link your program with QuickDraw, these JSRs are adjusted to refer to the jump table in low RAM, so that a JSR into the table redirects you to the actual location of the procedure or function.

The only difficult part about calling QuickDraw procedures and functions is stacking the parameters. You must follow some strict rules:

- Save all registers you wish to preserve BEFORE you begin pushing parameters. Any QuickDraw procedure or function can destroy the contents of the registers AØ, A1, DØ, D1, and D2, but the others are never altered.

- Push the parameters in the order that they appear in the Pascal procedural interface.

- For booleans, push a byte; for integers and characters, push a word; for pointers, handles, long integers, and reals, push a long.

- For any structured variable longer than four (4) bytes, push a pointer to the variable.

- For all VAR parameters, regardless of size, push a pointer to the variable.

- When calling a function, FIRST push a null entry equal to the size of the function result, THEN push all other parameters. The result will be left on the stack after the function returns to you.

This makes for a lengthy interface, but it also guarantees that you can mock up a Pascal version of your program, and later translate it into assembly code that works the same. For example, the Pascal statement

```
blackness := GetPixel(5Ø,mousePos.v);
```

would be written in assembly language like this:

```
CLR.W   -(SP)                ; Save space for boolean result
MOVE.W  #5Ø,-(SP)            ; Push constant 5Ø (decimal)
MOVE.W  MOUSEPOS+V,-(SP)     ; Push the value of mousePos.v
JSR     GETPIXEL             ; Call routine
MOVE.W  (SP)+,BLACKNESS      ; Fetch result from stack
```

This is a simple example, pushing and pulling word-long constants. Normally, you'll be pushing more pointers, using the PEA (Push Effective Address) instruction:

```
FillRoundRect(myRect,1,thePort^.pnSize.v,white);

PEA     MYRECT              ; Push pointer to myRect
MOVE.W  #1,-(SP)            ; Push constant 1
MOVE.L  GRAFGLOB(A5),AØ     ; Point to QuickDraw globals
MOVE.L  THEPORT(AØ),A1      ; Get current grafPort
MOVE.W  PNSIZE+V(A1),-(SP)  ; Push value of thePort^.pnSize.v
PEA     WHITE(AØ)           ; Push pointer to global variable white
JSR     FILLROUNDRECT       ; Call the subroutine
```

To call the TextFace procedure, push a word in which each of seven bits represents a stylistic variation: set bit Ø for bold, bit 1 for italic, bit 2 for underline, bit 3 for outline, bit 4 for shadow, bit 5 for condense, and bit 6 for extend.

SUMMARY OF QUICKDRAW

```
CONST srcCopy    = Ø;
      srcOr      = 1;
      srcXor     = 2;
      srcBic     = 3;
      notSrcCopy = 4;
      notSrcOr   = 5;
      notSrcXor  = 6;
      notSrcBic  = 7;
      patCopy    = 8;
      patOr      = 9;
      patXor     = 1Ø;
      patBic     = 11;
      notPatCopy = 12;
      notPatOr   = 13;
      notPatXor  = 14;
      notPatBic  = 15;

      blackColor   = 33;
      whiteColor   = 3Ø;
      redColor     = 2Ø5;
      greenColor   = 341;
      blueColor    = 4Ø9;
      cyanColor    = 273;
      magentaColor = 137;
      yellowColor  = 69;

      picLParen = Ø;
      picRParen = 1;

TYPE QDByte   = -128..127;
     QDPtr    = ^QDByte;
     QDHandle = ^QDPtr;
     Str255   = STRING[255];
     Pattern  = PACKED ARRAY [Ø..7] OF Ø..255;
     Bits16   = ARRAY [Ø..15] OF INTEGER;
     GrafVerb = (frame, paint, erase, invert, fill);

     StyleItem = (bold, italic, underline, outline, shadow, condense,
                  extend);
     Style     = SET OF StyleItem;

     FontInfo = RECORD
                  ascent:  INTEGER;
                  descent: INTEGER;
                  widMax:  INTEGER;
                  leading: INTEGER
                END;
```

```
VHSelect = (v,h);
Point    = RECORD CASE INTEGER OF

             Ø: (v: INTEGER;
                 h: INTEGER);

             1: (vh: ARRAY[VHSelect] OF INTEGER)

           END;

Rect = RECORD CASE INTEGER OF

         Ø: (top:      INTEGER;
             left:     INTEGER;
             bottom:   INTEGER;
             right:    INTEGER);

         1: (topLeft:  Point;
             botRight: Point)

       END;

BitMap = RECORD
           baseAddr: QDPtr;
           rowBytes: INTEGER;
           bounds:   Rect
         END;

Cursor = RECORD
           data:    Bits16;
           mask:    Bits16;
           hotSpot: Point
         END;

PenState = RECORD
             pnLoc:    Point;
             pnSize:   Point;
             pnMode:   INTEGER;
             pnPat:    Pattern
           END;

RgnHandle = ^RgnPtr;
RgnPtr    = ^Region;
Region    =  RECORD
               rgnSize:   INTEGER;
               rgnBBox:   Rect;
               {more data if not rectangular}
             END;
```

```
    PicHandle = ^PicPtr;
    PicPtr    = ^Picture;
    Picture   = RECORD
                  picSize:  INTEGER;
                  picFrame: Rect;
                  {picture definition data}
                END;

    PolyHandle = ^PolyPtr;
    PolyPtr    = ^Polygon;
    Polygon    = RECORD
                   polySize:   INTEGER;
                   polyBBox:   Rect;
                   polyPoints: ARRAY [Ø..Ø] OF Point
                 END;

QDProcsPtr = ^QDProcs;
QDProcs    = RECORD
               textProc:    QDPtr;
               lineProc:    QDPtr;
               rectProc:    QDPtr;
               rRectProc:   QDPtr;
               ovalProc:    QDPtr;
               arcProc:     QDPtr;
               polyProc:    QDPtr;
               rgnProc:     QDPtr;
               bitsProc:    QDPtr;
               commentProc: QDPtr;
               txMeasProc:  QDPtr;
               getPicProc:  QDPtr;
               putPicProc:  QDPtr
             END;
```

```
    GrafPtr  = ^GrafPort;
    GrafPort = RECORD
                 device:     INTEGER;
                 portBits:   BitMap;
                 portRect:   Rect;
                 visRgn:     RgnHandle;
                 clipRgn:    RgnHandle;
                 bkPat:      Pattern;
                 fillPat:    Pattern;
                 pnLoc:      Point;
                 pnSize:     Point;
                 pnMode:     INTEGER;
                 pnPat:      Pattern;
                 pnVis:      INTEGER;
                 txFont:     INTEGER;
                 txFace:     Style;
                 txMode:     INTEGER;
                 txSize:     INTEGER;
                 spExtra:    INTEGER;
                 fgColor:    LongInt;
                 bkColor:    LongInt;
                 colrBit:    INTEGER;
                 patStretch: INTEGER;
                 picSave:    QDHandle;
                 rgnSave:    QDHandle;
                 polySave:   QDHandle;
                 grafProcs:  QDProcsPtr
               END;

VAR thePort:     GrafPtr;
    white:       Pattern;
    black:       Pattern;
    gray:        Pattern;
    ltGray:      Pattern;
    dkGray:      Pattern;
    arrow:       Cursor;
    screenBits:  BitMap;
    randSeed:    LongInt;
```

GrafPort Routines

```
PROCEDURE InitGraf    (globalPtr: QDPtr);
PROCEDURE OpenPort    (gp: GrafPtr);
PROCEDURE InitPort    (gp: GrafPtr);
PROCEDURE ClosePort   (gp: GrafPtr);
PROCEDURE SetPort     (gp: GrafPtr);
PROCEDURE GetPort     (VAR gp: GrafPtr);
PROCEDURE GrafDevice  (device: INTEGER);
PROCEDURE SetPortBits (bm: BitMap);
PROCEDURE PortSize    (width,height: INTEGER);
PROCEDURE MovePortTo  (leftGlobal,topGlobal: INTEGER);
PROCEDURE SetOrigin   (h,v: INTEGER);
```

```
PROCEDURE SetClip     (rgn: RgnHandle);
PROCEDURE GetClip     (rgn: RgnHandle);
PROCEDURE ClipRect    (r: Rect);
PROCEDURE BackPat     (pat: Pattern);
```

Cursor Handling

```
PROCEDURE InitCursor;
PROCEDURE SetCursor      (crsr: Cursor);
PROCEDURE HideCursor;
PROCEDURE ShowCursor;
PROCEDURE ObscureCursor;
```

Pen and Line Drawing

```
PROCEDURE HidePen;
PROCEDURE ShowPen;
PROCEDURE GetPen      (VAR pt: Point);
PROCEDURE GetPenState (VAR pnState: PenState);
PROCEDURE SetPenState (pnState: PenState);
PROCEDURE PenSize     (width,height: INTEGER);
PROCEDURE PenMode     (mode: INTEGER);
PROCEDURE PenPat      (pat: Pattern);
PROCEDURE PenNormal;
PROCEDURE MoveTo      (h,v: INTEGER);
PROCEDURE Move        (dh,dv: INTEGER);
PROCEDURE LineTo      (h,v: INTEGER);
PROCEDURE Line        (dh,dv: INTEGER);
```

Text Drawing

```
PROCEDURE TextFont    (font: INTEGER);
PROCEDURE TextFace    (face: Style);
PROCEDURE TextMode    (mode: INTEGER);
PROCEDURE TextSize    (size: INTEGER);
PROCEDURE SpaceExtra  (extra: INTEGER);
PROCEDURE DrawChar    (ch: CHAR);
PROCEDURE DrawString  (s: Str255);
PROCEDURE DrawText    (textBuf: QDPtr; firstByte,byteCount: INTEGER);
FUNCTION  CharWidth   (ch: CHAR) : INTEGER;
FUNCTION  StringWidth (s: Str255) : INTEGER;
FUNCTION  TextWidth   (textBuf: QDPtr; firstByte,byteCount: INTEGER) :
                       INTEGER;
PROCEDURE GetFontInfo (VAR info: FontInfo);
```

Drawing in Color

```
PROCEDURE ForeColor (color: LongInt);
PROCEDURE BackColor (color: LongInt);
PROCEDURE ColorBit  (whichBit: INTEGER);
```

Calculations with Rectangles

```
PROCEDURE SetRect    (VAR r: Rect; left,top,right,bottom: INTEGER);
PROCEDURE OffsetRect (VAR r: Rect; dh,dv: INTEGER);
PROCEDURE InsetRect  (VAR r: Rect; dh,dv: INTEGER);
FUNCTION  SectRect   (srcRectA,srcRectB: Rect; VAR dstRect: Rect) :
                      BOOLEAN;
PROCEDURE UnionRect  (srcRectA,srcRectB: Rect; VAR dstRect: Rect)
FUNCTION  PtInRect   (pt: Point; r: Rect) : BOOLEAN;
PROCEDURE Pt2Rect    (ptA,ptB: Point; VAR dstRect: Rect);
PROCEDURE PtToAngle  (r: Rect; pt: Point; VAR angle: INTEGER);
FUNCTION  EqualRect  (rectA,rectB: Rect) : BOOLEAN;
FUNCTION  EmptyRect  (r: Rect) : BOOLEAN;
```

Graphic Operations on Rectangles

```
PROCEDURE FrameRect  (r: Rect);
PROCEDURE PaintRect  (r: Rect);
PROCEDURE EraseRect  (r: Rect);
PROCEDURE InvertRect (r: Rect);
PROCEDURE FillRect   (r: Rect; pat: Pattern);
```

Graphic Operations on Ovals

```
PROCEDURE FrameOval  (r: Rect);
PROCEDURE PaintOval  (r: Rect);
PROCEDURE EraseOval  (r: Rect);
PROCEDURE InvertOval (r: Rect);
PROCEDURE FillOval   (r: Rect; pat: Pattern);
```

Graphic Operations on Rounded-Corner Rectangles

```
PROCEDURE FrameRoundRect  (r: Rect; ovalWidth,ovalHeight: INTEGER);
PROCEDURE PaintRoundRect  (r: Rect; ovalWidth,ovalHeight: INTEGER);
PROCEDURE EraseRoundRect  (r: Rect; ovalWidth,ovalHeight: INTEGER);
PROCEDURE InvertRoundRect (r: Rect; ovalWidth,ovalHeight: INTEGER);
PROCEDURE FillRoundRect   (r: Rect; ovalWidth,ovalHeight: INTEGER;
                           pat: Pattern);
```

Graphic Operations on Arcs and Wedges

```
PROCEDURE FrameArc  (r: Rect; startAngle,arcAngle: INTEGER);
PROCEDURE PaintArc  (r: Rect; startAngle,arcAngle: INTEGER);
PROCEDURE EraseArc  (r: Rect; startAngle,arcAngle: INTEGER);
PROCEDURE InvertArc (r: Rect; startAngle,arcAngle: INTEGER);
PROCEDURE FillArc   (r: Rect; startAngle,arcAngle: INTEGER; pat:
                     Pattern);
```

Calculations with Regions

```
FUNCTION  NewRgn :     RgnHandle;
PROCEDURE DisposeRgn  (rgn: RgnHandle);
PROCEDURE CopyRgn     (srcRgn,dstRgn: RgnHandle);
PROCEDURE SetEmptyRgn (rgn: RgnHandle);
PROCEDURE SetRectRgn  (rgn: RgnHandle; left,top,right,bottom: INTEGER);
PROCEDURE RectRgn     (rgn: RgnHandle; r: Rect);
PROCEDURE OpenRgn;
PROCEDURE CloseRgn    (dstRgn: RgnHandle);
PROCEDURE OffsetRgn   (rgn: RgnHandle; dh,dv: INTEGER);
PROCEDURE InsetRgn    (rgn: RgnHandle; dh,dv: INTEGER);
PROCEDURE SectRgn     (srcRgnA,srcRgnB,dstRgn: RgnHandle);
PROCEDURE UnionRgn    (srcRgnA,srcRgnB,dstRgn: RgnHandle);
PROCEDURE DiffRgn     (srcRgnA,srcRgnB,dstRgn: RgnHandle);
PROCEDURE XorRgn      (srcRgnA,srcRgnB,dstRgn: RgnHandle);
FUNCTION  PtInRgn     (pt: Point; rgn: RgnHandle) : BOOLEAN;
FUNCTION  RectInRgn   (r: Rect; rgn: RgnHandle) : BOOLEAN;
FUNCTION  EqualRgn    (rgnA,rgnB: RgnHandle) : BOOLEAN;
FUNCTION  EmptyRgn    (rgn: RgnHandle) : BOOLEAN;
```

Graphic Operations on Regions

```
PROCEDURE FrameRgn  (rgn: RgnHandle);
PROCEDURE PaintRgn  (rgn: RgnHandle);
PROCEDURE EraseRgn  (rgn: RgnHandle);
PROCEDURE InvertRgn (rgn: RgnHandle);
PROCEDURE FillRgn   (rgn: RgnHandle; pat: Pattern);
```

Bit Transfer Operations

```
PROCEDURE ScrollRect (r: Rect; dh,dv: INTEGER; updateRgn: RgnHandle);
PROCEDURE CopyBits   (srcBits,dstBits: BitMap; srcRect,dstRect: Rect;
                      mode: INTEGER; maskRgn: RgnHandle);
```

Pictures

```
FUNCTION  OpenPicture   (picFrame: Rect) : PicHandle;
PROCEDURE PicComment    (kind,dataSize: INTEGER; dataHandle: QDHandle);
PROCEDURE ClosePicture;
PROCEDURE DrawPicture   (myPicture: PicHandle; dstRect: Rect);
PROCEDURE KillPicture   (myPicture: PicHandle);
```

Calculations with Polygons

```
FUNCTION  OpenPoly :  PolyHandle;
PROCEDURE ClosePoly;
PROCEDURE KillPoly   (poly: PolyHandle);
PROCEDURE OffsetPoly (poly: PolyHandle; dh,dv: INTEGER);
```

Graphic Operations on Polygons

```
PROCEDURE FramePoly  (poly: PolyHandle);
PROCEDURE PaintPoly  (poly: PolyHandle);
PROCEDURE ErasePoly  (poly: PolyHandle);
PROCEDURE InvertPoly (poly: PolyHandle);
PROCEDURE FillPoly   (poly: PolyHandle; pat: Pattern);
```

Calculations with Points

```
PROCEDURE AddPt         (srcPt: Point; VAR dstPt: Point);
PROCEDURE SubPt         (srcPt: Point; VAR dstPt: Point);
PROCEDURE SetPt         (VAR pt: Point; h,v: INTEGER);
FUNCTION  EqualPt       (ptA,ptB: Point) : BOOLEAN;
PROCEDURE LocalToGlobal (VAR pt: Point);
PROCEDURE GlobalToLocal (VAR pt: Point);
```

Miscellaneous Utilities

```
FUNCTION  Random :  INTEGER;
FUNCTION  GetPixel (h,v: INTEGER) : BOOLEAN;
PROCEDURE StuffHex (thingPtr: QDPtr; s: Str255);
PROCEDURE ScalePt  (VAR pt: Point; srcRect,dstRect: Rect);
PROCEDURE MapPt    (VAR pt: Point; srcRect,dstRect: Rect);
PROCEDURE MapRect  (VAR r: Rect; srcRect,dstRect: Rect);
PROCEDURE MapRgn   (rgn: RgnHandle; srcRect,dstRect: Rect);
PROCEDURE MapPoly  (poly: PolyHandle; srcRect,dstRect: Rect);
```

Customizing QuickDraw Operations

```
PROCEDURE SetStdProcs (VAR procs: QDProcs);
PROCEDURE StdText     (byteCount: INTEGER; textAddr: QDPtr; numer,denom:
                       Point);
PROCEDURE StdLine     (newPt: Point);
PROCEDURE StdRect     (verb: GrafVerb; r: Rect);
PROCEDURE StdRRect    (verb: GrafVerb; r: Rect; ovalwidth,ovalHeight:
                       INTEGER);
PROCEDURE StdOval     (verb: GrafVerb; r: Rect);
PROCEDURE StdArc      (verb: GrafVerb; r: Rect; startAngle,arcAngle:
                       INTEGER);
PROCEDURE StdPoly     (verb: GrafVerb; poly: PolyHandle);
PROCEDURE StdRgn      (verb: GrafVerb; rgn: RgnHandle);
PROCEDURE StdBits     (VAR srcBits: BitMap; VAR srcRect,dstRect: Rect;
                       mode: INTEGER; maskRgn: RgnHandle);
PROCEDURE StdComment  (kind,dataSize: INTEGER; dataHandle: QDHandle);
FUNCTION  StdTxMeas   (byteCount: INTEGER; textBuf: QDPtr; VAR numer,
                       denom: Point; VAR info: FontInfo) : INTEGER;
PROCEDURE StdGetPic   (dataPtr: QDPtr; byteCount: INTEGER);
PROCEDURE StdPutPic   (dataPtr: QDPtr; byteCount: INTEGER);
```

GLOSSARY

bit image: A collection of bits in memory which have a rectilinear representation. The Macintosh screen is a visible bit image.

bitMap: A pointer to a bit image, the row width of that image, and its boundary rectangle.

boundary rectangle: A rectangle defined as part of a bitMap, which encloses the active area of the bit image and imposes a coordinate system on it. Its top left corner is always aligned around the first bit in the bit image.

character style: A set of stylistic variations, such as bold, italic, and underline. The empty set indicates normal text (no stylistic variations).

clipping: Limiting drawing to within the bounds of a particular area.

clipping region: Same as clipRgn.

clipRgn: The region to which an application limits drawing in a grafPort.

coordinate plane: A two-dimensional grid. In QuickDraw, the grid coordinates are integers ranging from -32768 to +32767, and all grid lines are infinitely thin.

cursor: A 16-by-16-bit image that appears on the screen and is controlled by the mouse; called the "pointer" in other Macintosh documentation.

cursor level: A value, initialized to Ø when the system is booted, that keeps track of the number of times the cursor has been hidden.

empty: Containing no bits, as a shape defined by only one point.

font: The complete set of characters of one typeface, such as Helvetica.

frame: To draw a shape by drawing an outline of it.

global coordinate system: The coordinate system based on the top left corner of the bit image being at (Ø,Ø).

grafPort: A complete drawing environment, including such elements as a bitMap, a subset of it in which to draw, a character font, patterns for drawing and erasing, and other pen characteristics.

grafPtr: A pointer to a grafPort.

handle: A pointer to one master pointer to a dynamic, relocatable data structure (such as a region).

hotSpot: The point in a cursor that is aligned with the mouse position.

kern: To stretch part of a character back under the previous character.

local coordinate system: The coordinate system local to a grafPort, imposed by the boundary rectangle defined in its bitMap.

missing symbol: A character to be drawn in case of a request to draw a character that is missing from a particular font.

pattern: An 8-by-8-bit image, used to define a repeating design (such as stripes) or tone (such as gray).

pattern transfer mode: One of eight transfer modes for drawing lines or shapes with a pattern.

picture: A saved sequence of QuickDraw drawing commands (and, optionally, picture comments) that you can play back later with a single procedure call; also, the image resulting from these commands.

picture comments: Data stored in the definition of a picture which does not affect the picture's appearance but may be used to provide additional information about the picture when it's played back.

picture frame: A rectangle, defined as part of a picture, which surrounds the picture and gives a frame of reference for scaling when the picture is drawn.

pixel: The visual representation of a bit on the screen (white if the bit is Ø, black if it's 1).

point: The intersection of a horizontal grid line and a vertical grid line on the coordinate plane, defined by a horizontal and a vertical coordinate.

polygon: A sequence of connected lines, defined by QuickDraw line-drawing commands.

port: Same as grafPort.

portBits: The bitMap of a grafPort.

portBits.bounds: The boundary rectangle of a grafPort's bitMap.

portRect: A rectangle, defined as part of a grafPort, which encloses a subset of the bitMap for use by the grafPort.

region: An arbitrary area or set of areas on the coordinate plane. The outline of a region should be one or more closed loops.

row width: The number of bytes in each row of a bit image.

solid: Filled in with any pattern.

source transfer mode: One of eight transfer modes for drawing text or transferring any bit image between two bitMaps.

style: See character style.

thePort: A global variable that points to the current grafPort.

transfer mode: A specification of which boolean operation QuickDraw should perform when drawing or when transferring a bit image from one bitMap to another.

visRgn: The region of a grafPort, manipulated by the Window Manager, which is actually visible on the screen.

MACINTOSH USER EDUCATION

The Font Manager: A Programmer's Guide /FMGR/FONT

See Also: Macintosh User Interface Guidelines
The Memory Manager: A Programmer's Guide
Macintosh Operating System Reference Manual
QuickDraw: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Menu Manager: A Programmer's Guide
Programming Macintosh Applications in Assembly Language

Modification History:			
	Preliminary Draft	Caroline Rose	4/2Ø/83
	First Draft (ROM 3.Ø)	Caroline Rose	4/22/83
	Second Draft (ROM 7)	Brad Hacker	2/7/84
	Third Draft	Caroline Rose & Brad Hacker	6/11/84

ABSTRACT

The Font Manager is the part of the Macintosh User Interface Toolbox that supports the use of various character fonts when you draw text with QuickDraw. This manual introduces you to the Font Manager and describes the routines your application can call to get font information. It also describes the data structures of fonts and discusses how the Font Manager communicates with QuickDraw.

Summary of significant changes and additions since last draft:

- The default application font has changed from New York to Geneva.
- Details are now given on the font characterization table (page 13).
- Programmers defining their own fonts must include the characters with ASCII codes $ØØ, $Ø9, and $ØD (page 18).
- The sample location table and offset/width table have been corrected, as has the calculation of the offset in the font record's owTLoc field (page 21).
- Some assembly-language information has been changed and added.

TABLE OF CONTENTS

ABOUT THIS MANUAL

The Font Manager is the part of the Macintosh User Interface Toolbox that supports the use of various character fonts when you draw text with QuickDraw. This manual introduces you to the Font Manager and describes the routines your application can call to get font information. It also describes the data structures of fonts and discusses how the Font Manager communicates with QuickDraw. *** Eventually this will become part of the comprehensive Inside Macintosh manual. ***

Like all documentation about Toolbox units, this manual assumes you're familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's Memory Manager. You should also be familiar with:

- resources, as described in the Resource Manager manual

- the basic concepts and structures behind QuickDraw, particularly bit images and how to draw text

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly-language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it.

The manual begins with an overview of the Font Manager and what you can do with it. It then discusses the font numbers by which fonts are identified, the characters in a font, and the scaling of fonts to different sizes. Next, a section on using the Font Manager introduces its routines and tells how they fit into the flow of your application. This is followed by detailed descriptions of Font Manager procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that will not interest all readers. There's a discussion of how QuickDraw and the Font Manager communicate, followed by a section that describes the format of the data structures used to define fonts, and how QuickDraw uses the data to draw characters. Next is a section that gives the exact format of fonts in a resource file.

Finally, there's a summary of the Font Manager, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE FONT MANAGER

The main function of the Font Manager is to provide font support for QuickDraw. To the Macintosh user, font means the complete set of characters of one typeface; it doesn't include the size of the characters, and usually doesn't include any stylistic variations (such

as bold and italic).

(note)

Usually fonts are defined in the normal style and stylistic variations are applied to them; for example, the italic style simply slants the normal characters. However, fonts may be designed to include stylistic variations in the first place.

The way you identify a font to QuickDraw or the Font Manager is with a font number. Every font also has a name (such as "New York") that's appropriate to include in a menu of available fonts.

The size of the characters, called the font size, is given in points. Here this term doesn't have the same meaning as the "point" that's an intersection of lines on the QuickDraw coordinate plane, but instead is a typographical term that stands for approximately 1/72 inch. The font size measures the distance between the ascent line of one line of text and the ascent line of the next line of single-spaced text (see Figure 1). It assumes 8Ø pixels per inch, the approximate resolution of the Macintosh screen. For example, since an Imagewriter printer has twice the resolution of the screen, high-resolution 9-point output to the printer is actually accomplished with an 18-point font.

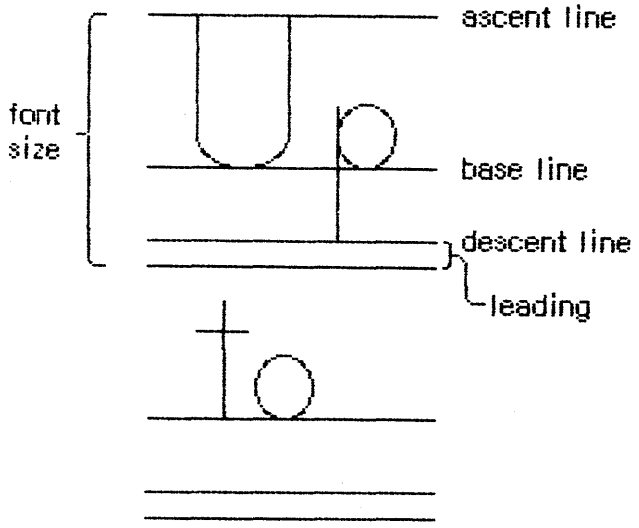


Figure 1. Font Size

(note)

Because measurements cannot be exact on a bit-mapped output device, the actual font size may be slightly different from what it would be in normal typography.

Whenever you call a QuickDraw routine that does anything with text, QuickDraw passes the following information to the Font Manager:

- The font number.

- The character style, which is a set of stylistic variations. The empty set indicates normal text. (See the QuickDraw manual for details.)

- The font size. The size may range from 1 point to 127 points, but for readability should be at least 6 points.

- The horizontal and vertical scaling factors, each of which is represented by a numerator and a denominator (for example, a numerator of 2 and a denominator of 1 indicates 2-to-1 scaling in that direction).

- A Boolean value indicating whether the characters will actually be drawn or not. They will not be drawn, for example, when the QuickDraw function CharWidth is called (since it only measures characters) or when text is drawn after the pen has been hidden (such as by the HidePen procedure or the OpenPicture function, which calls HidePen).

- A number specifying the device on which the characters will be drawn or printed. The number Ø represents the Macintosh screen. The Font Manager can adapt fonts to other devices.

Given this information, the Font Manager provides QuickDraw with information describing the font and--if the characters will actually be drawn--the bits comprising the characters.

Fonts are stored as resources in resource files; the Font Manager calls the Resource Manager to read them into memory. System-defined fonts are stored in the system resource file. You may define your own fonts with the aid of the Resource Editor and include them in the system resource file so they can be shared among applications. *** (The Resource Editor doesn't yet exist, but an interim Font Editor is available from Macintosh Technical Support.) *** In special cases, you may want to store a font in an application's resource file or even in the resource file for a document. It's also possible to store only the character widths and general font information, and not the bits comprising the characters, for those cases where the characters won't actually be drawn.

A font may be stored in any number of sizes in a resource file. If a size is needed that's not available as a resource, the Font Manager scales an available size.

Fonts occupy a large amount of storage: a 12-point font typically occupies about 3K bytes, and a 24-point font, about 1ØK bytes; fonts for use on a high-resolution output device can take up four times as much space as that (up to a maximum of 32K bytes). Fonts normally are purgeable, which means they may be removed from the heap when space is required by the Memory Manager. If you wish, you can call a Font Manager routine to make a font temporarily unpurgeable.

There are also routines that provide information about a font. You can find out the name of a font having a particular font number, or the font number for a font having a particular name. You can also learn whether a font is available in a certain size or will have to be scaled to that size.

FONT NUMBERS

The Font Manager includes the following font numbers for identifying system-defined fonts:

```
CONST systemFont = Ø;  {system font}
      applFont   = 1;  {application font}
      newYork    = 2;
      geneva     = 3;
      monaco     = 4;
      venice     = 5;
      london     = 6;
      athens     = 7;
      sanFran    = 8;
      toronto    = 9;
```

The system font is so called because it's the font used by the system (for drawing menu titles and commands in menus, for example). The name of the system font is Chicago. The size of text drawn by the system in this font is fixed at 12 points (called the system font size).

The application font is the font your application will use unless you specify otherwise. Unlike the system font, the application font isn't a separate font with its own typeface, but is essentially a reference to another font--Geneva, by default. *** In the future, there may be a way for the user to change the application font, perhaps through the Control Panel desk accessory. ***

Assembly-language note: The font number of the application font is stored in the global variable apFontID.

CHARACTERS IN A FONT

A font can consist of up to 255 distinct characters; not all characters need be defined in a single font. Figure 2 on the following page shows the standard printing characters on the Macintosh and their ASCII codes (for example, the ASCII code for "A" is 41 hexadecimal, or 65 decimal).

In addition to its maximum of 255 characters, every font contains a missing symbol that's drawn in case of a request to draw a character that's missing from the font.

FONT SCALING

The information QuickDraw passes to the Font Manager includes the font size and the scaling factors QuickDraw wants to use. The Font Manager determines the font information to return to QuickDraw by looking for the exact size needed among the sizes stored for the font. If the exact size requested isn't available, it then looks for a nearby size that it can scale.

1. It looks first for a font that's twice the size, and scales down that size if there is one.

2. If there's no font that's twice the size, it looks for a font that's half the size, and scales up that size if there is one.

3. If there's no font that's half the size, it looks for a larger size of the font, and scales down the next larger size if there is one.

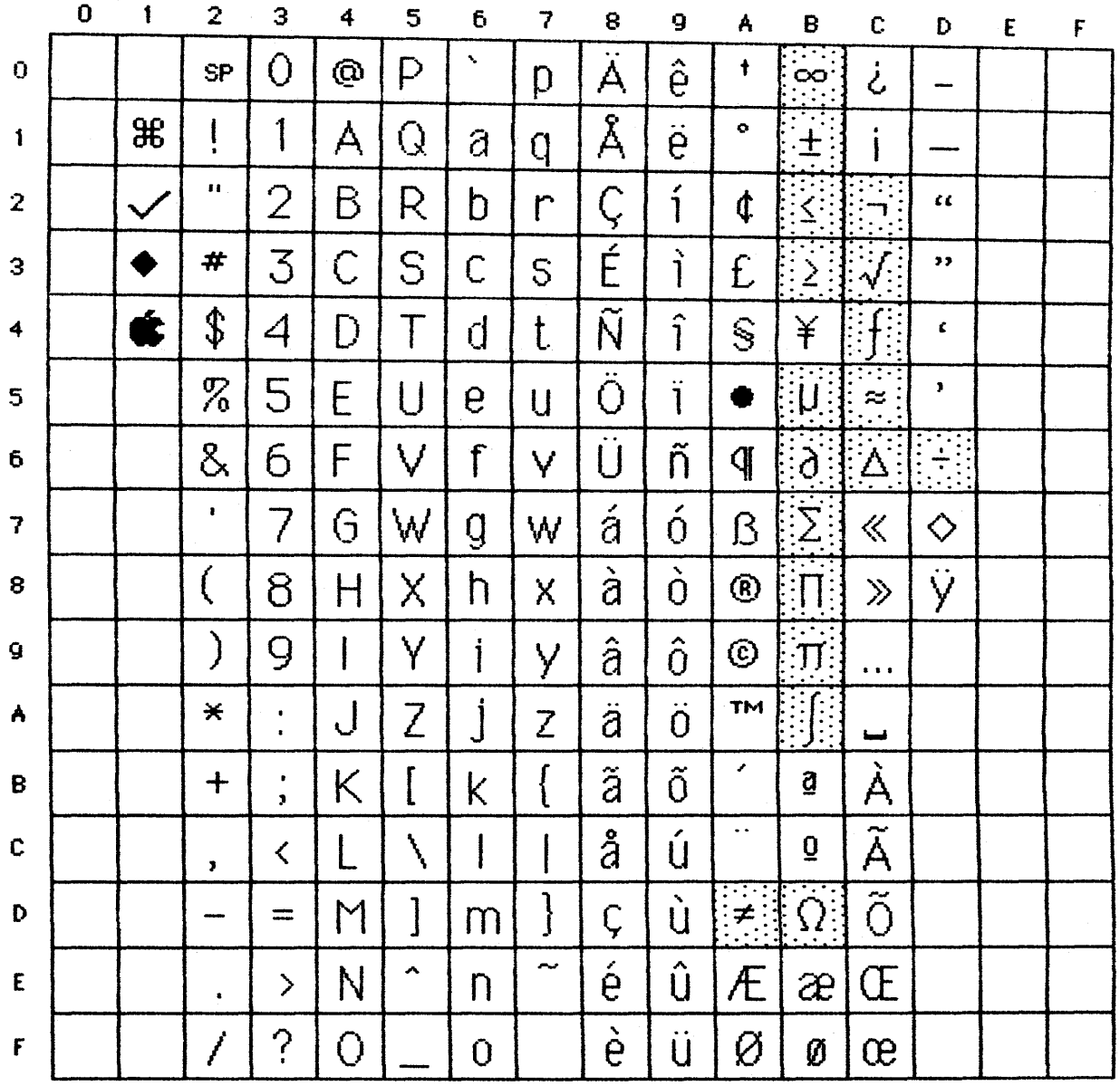
4. If there's no larger size, it looks for a smaller size of the font, and scales up the closest smaller size if there is one.

5. If the font isn't available in any size at all, it uses the application font instead, scaling the font to the proper size.

6. If the application font isn't available in any size at all, it uses the system font instead, scaling the font to the proper size.

Scaling looks best when the scaled size is an even multiple of an available size.

Assembly-language note: You can use the global variable fScaleDisable to defeat scaling, if desired. Normally, fScaleDisable is Ø. If you set it to a nonzero value, the Font Manager will look for the size as described above but will return the font unscaled.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			SP	0	@	P	`	p	Ä	ê	†	∞	¿	–		
1		⌘	!	1	A	Q	a	q	Å	ë	°	±	¡	—		
2		✓	"	2	B	R	b	r	Ç	í	¢	≤	¬	“		
3		◆	#	3	C	S	c	s	É	ì	£	≥	√	”		
4			$	4	D	T	d	t	Ñ	î	§	¥	ƒ	‘		
5			%	5	E	U	e	u	Ö	ï	•	µ	≈	’		
6			&	6	F	V	f	v	Ü	ñ	¶	∂	Δ	÷		
7			'	7	G	W	g	w	á	ó	ß	∑	«	◇		
8			(	8	H	X	h	x	à	ò	®	∏	»	ÿ		
9			)	9	I	Y	i	y	â	ô	©	π	…			
A			*	:	J	Z	j	z	ä	ö	™	∫	␣			
B			+	;	K	[	k	{	ã	õ	´	ª	À			
C			,	<	L	\	l	|	å	ú	¨	º	Ã			
D			–	=	M	]	m	}	ç	ù	≠	Ω	Õ			
E			.	>	N	^	n	~	é	û	Æ	æ	Œ			
F			/	?	O	_	o		è	ü	Ø	ø	œ			

SP stands for a space.

␣ stands for a nonbreaking space, same width as numbers.

The first four characters are only in the system font (Chicago).
The shaded characters are only in the Geneva, Monaco and system fonts.

ASCII codes $9D through $FF are reserved for future expansion.

Figure 2. Font Characters

USING THE FONT MANAGER

This section introduces you to the Font Manager routines and how they fit into the general flow of an application program. The routines themselves are described in detail in the next section.

The InitFonts procedure initializes the Font Manager; you should call it after initializing QuickDraw but before initializing the Window Manager.

You can set up a menu of fonts in your application by using the Menu Manager procedure AddResMenu (see the Menu Manager manual for details). When the user chooses a menu item from the font menu, call the Menu Manager procedure GetItem to get the name of the corresponding font, and then the Font Manager function GetFNum to get the font number. The GetFontName function does the reverse of GetFNum: given a font ID, it returns the font name.

In a menu of font sizes in your application, you may want to let the user know which sizes the current font is available in and therefore will not require scaling. You can call the RealFont function to find out whether a font is available in a given size.

If you know you'll be using a font a lot and don't want it to be purged, you can use the SetFontLock procedure to make the font unpurgeable during that time.

Advanced programmers who want to write their own font editors or otherwise manipulate fonts can access fonts directly with the SwapFont function.

FONT MANAGER ROUTINES

This section describes all the Font Manager procedures and functions. The routines are presented in their Pascal form; for information on using them from assembly language, see the manual Programming Macintosh Applications in Assembly Language.

Initializing the Font Manager

PROCEDURE InitFonts;

InitFonts initializes the Font Manager. If the system font isn't already in memory, InitFonts reads it into memory. Call this procedure once before all other Font Manager routines or any Toolbox routine that will call the Font Manager.

Getting Font Information

```
PROCEDURE GetFontName (fontNum: INTEGER; VAR theName: Str255);
```

GetFontName returns in theName the name of the font having the font number fontNum. If there's no such font, GetFontName returns the empty string.

Assembly-language note: The macro you invoke to call GetFontName from assembly language is named _GetFName.

```
PROCEDURE GetFNum (fontName: Str255; VAR theNum: INTEGER);
```

GetFNum returns in theNum the font number for the font having the given fontName. If there's no such font, GetFNum returns Ø.

```
FUNCTION RealFont (fontNum: INTEGER; size: INTEGER) : BOOLEAN;
```

RealFont returns TRUE if the font having the font number fontNum is available in the given size in a resource file, or FALSE if the font has to be scaled to that size.

Keeping Fonts in Memory

```
PROCEDURE SetFontLock (lockFlag: BOOLEAN);
```

SetFontLock applies to the font in which text was most recently drawn; it makes the font unpurgeable if lockFlag is TRUE or purgeable if lockFlag is FALSE. Since fonts are normally purgeable, this procedure is useful for making a font temporarily unpurgeable.

Advanced Routine

The following low-level routine will not normally be used by an application directly, but may be of interest to advanced programmers who want to bypass the QuickDraw routines that deal with text.

```
FUNCTION SwapFont (inRec: FMInput) : FMOutPtr;
```

SwapFont returns a pointer to an FMOutput record containing the size, style, and other information about an adapted version of the font requested in the given FMInput record. (FMInput and FMOutput records are explained in the following section.) SwapFont is called by QuickDraw every time a QuickDraw routine that does anything with text is used. If you want to call SwapFont yourself, you must build an FMInput record and then use the returned pointer to access the resulting FMOutput record.

Assembly-language note: The macro you invoke to call SwapFont from assembly language is named _FMSwapFont.

COMMUNICATION BETWEEN QUICKDRAW AND THE FONT MANAGER

This section describes the data structures that allow QuickDraw and the Font Manager to exchange information. It also discusses the communication that may occur between the Font Manager and the driver of the device on which the characters are being drawn or printed. You can skip this section if you want to change fonts, character style, and font sizes by calling QuickDraw and aren't interested in the lower-level data structures and routines of the Font Manager. To understand this section fully, you'll have to be familiar with device drivers and the Device Manager. *** (Device Manager manual doesn't yet exist.) ***

Whenever you call a QuickDraw routine that does anything with text, QuickDraw requests information from the Font Manager about the characters. The Font Manager performs any necessary calculations and returns the requested information to QuickDraw. As illustrated in Figure 3, this information exchange occurs via two data structures, a font input record (type FMInput) and a font output record (type FMOutput).

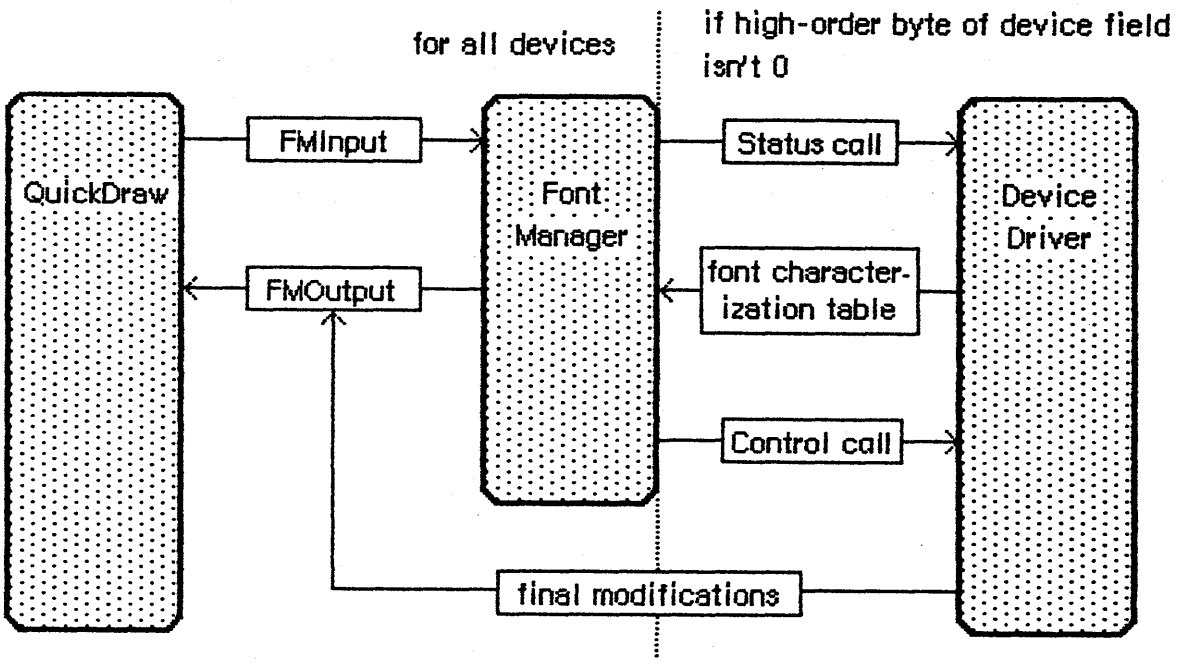


Figure 3. Communication About Fonts

First, QuickDraw passes the Font Manager a font input record:

```
TYPE FMInput = PACKED RECORD
                 family:   INTEGER;  {font number}
                 size:     INTEGER;  {font size}
                 face:     Style;    {character style}
                 needBits: BOOLEAN;  {TRUE if drawing}
                 device:   INTEGER;  {device-specific information}
                 numer:    Point;    {numerators of scaling factors}
                 denom:    Point     {denominators of scaling factors}
               END;
```

The first three fields contain the font number, size, and character style that QuickDraw wants to use. The needBits field indicates whether the characters actually will be drawn or not. If the characters are being drawn, all of the information describing the font, including the bit image comprising the characters, will be read into memory. If the characters aren't being drawn and there's a resource consisting of only the character widths and general font information, that resource will be read instead.

The high-order byte of the device field contains a device driver reference number. From the driver reference number, the Font Manager can determine the optimum stylistic variations on the font to produce the highest quality printing or drawing available on a device (as explained below). The low-order byte of the device field is ignored by the Font Manager but may contain information used by the device driver.

The numer and denom fields contain the scaling factors to be used; numer.v divided by denom.v gives the vertical scaling, and numer.h

divided by denom.h gives the horizontal scaling.

The Font Manager takes the FMInput record and asks the Resource Manager for the font. If the requested size isn't available, the Font Manager scales another size to match (as described previously).

Then the Font Manager gets the font characterization table via the device field. If the high-order byte of the device field is Ø, the Font Manager gets the font characterization table for the screen (which is stored in the Font Manager). If the high-order byte of the device field is nonzero, the Font Manager calls the status routine of the device driver having that reference number, and the status routine returns a font characterization table. The status routine may use the value of the low-order byte of the device field to determine the font characterization table it returns.

(note)

If you want to make your own calls to the device driver's status routine, the refNum parameter of the Status function must contain the driver reference number from the font input record's device field, the csCode parameter must be 8, and the csParam parameter must contain a pointer to the following: a pointer to where the device driver should put the font characterization table followed by an integer containing the value of the font input record's device field.

Figure 4 shows the structure of a font characterization table and, on the right, the values it contains for the Macintosh screen and Imagewriter printer driver.

		screen	Imagewriter
byte 0	dots per vertical inch on device	80	80
2	dots per horizontal inch on device	80	80
4	bold characteristics	0, 1, 1	0, 2, 2
7	italic characteristics	1, 8, 1	1, 8, 2
10	not used	0, 0, 0	0, 0, 0
13	outline characteristics	5, 1, 1	5, 1, 2
16	shadow characteristics	5, 2, 2	5, 2, 4
19	condensed characteristics	0, 0, -1	0, 0, -2
22	extended characteristics	0, 0, 1	0, 0, 2
25	underline characteristics	1, 1, 1	1, 3, 2

Figure 4. Font Characterization Table

The first two words of the font characterization table contain the number of dots per inch on the device. The remainder of the table consists of 3-byte triplets providing information about the different stylistic variations. For all but the triplet defining the underline characteristics:

- The first byte in the triplet indicates which byte beyond the bold field of the FMOutput record (see below) is affected by the triplet.

- The second byte contains the amount to be stored in the affected field.

- The third byte indicates the amount by which the extra field of the FMOutput record is to be incremented (starting from Ø).

The triplet defining the underline characteristics indicates the amount by which the FMOutput record's ulOffset, ulShadow, and ulThick fields (respectively) should be incremented.

Based on the information in the font characterization table, the Font Manager determines the optimum ascent, descent, and leading, so that the highest quality printing or drawing available will be produced. It then stores this information in a font output record:

```
TYPE FMOutput = PACKED RECORD
                  errNum:     INTEGER;     {not used}
                  fontHandle: Handle;      {handle to font record}
                  bold:       Byte;        {bold factor}
                  italic:     Byte;        {italic factor}
                  ulOffset:   Byte;        {underline offset}
                  ulShadow:   Byte;        {underline shadow}
                  ulThick:    Byte;        {underline thickness}
                  shadow:     Byte;        {shadow factor}
                  extra:      SignedByte;  {width of style}
                  ascent:     Byte;        {ascent}
                  descent:    Byte;        {descent}
                  widMax:     Byte;        {maximum character width}
                  leading:    SignedByte;  {leading}
                  unused:     Byte;        {not used}
                  numer:      Point;    {numerators of scaling factors}
                  denom:      Point     {denominators of scaling factors}
                END;
```

ErrNum is reserved for future use, and is set to Ø. FontHandle is a handle to the font record of the font, as described in the next section. Bold, italic, ulOffset, ulShadow, ulThick, and shadow are all fields that modify the way stylistic variations are done; their values are taken from the font characterization table, and are used by QuickDraw. (You'll need to experiment with these values if you want to determine exactly how they're used.) Extra indicates the number of pixels that each character has been widened by stylistic variation. For example, using the values shown in the rightmost column of Figure 4, the extra field for bold italic characters would be 4. Ascent, descent, widMax, and leading are the same as the fields of the FontInfo record returned by the QuickDraw procedure GetFontInfo. Numer and denom contain the scaling factors.

Just before returning this record to QuickDraw, the Font Manager calls the device driver's control routine to allow the driver to make any final modifications to the record. Finally, the font information is returned to QuickDraw via a pointer to the record, defined as follows:

```
TYPE FMOutPtr = ^FMOutput;
```

(note)

If you want to make your own calls to the device driver's control routine, the refNum parameter of the Control function must contain the driver reference number from the font input record's device field, the csCode parameter must be 8, and the csParam parameter must contain a pointer to the following: a pointer to the font output record followed by an integer containing the value of the font input record's device field.

FORMAT OF A FONT

This section describes the data structure that defines a font; you need to read it only if you're going to define your own fonts with the Resource Editor *** doesn't yet exist *** or write your own font editor.

Each character in a font is defined by pixels arranged in rows and columns. This pixel arrangement is called a character image; it's the image inside each of the character rectangles shown in Figure 5.

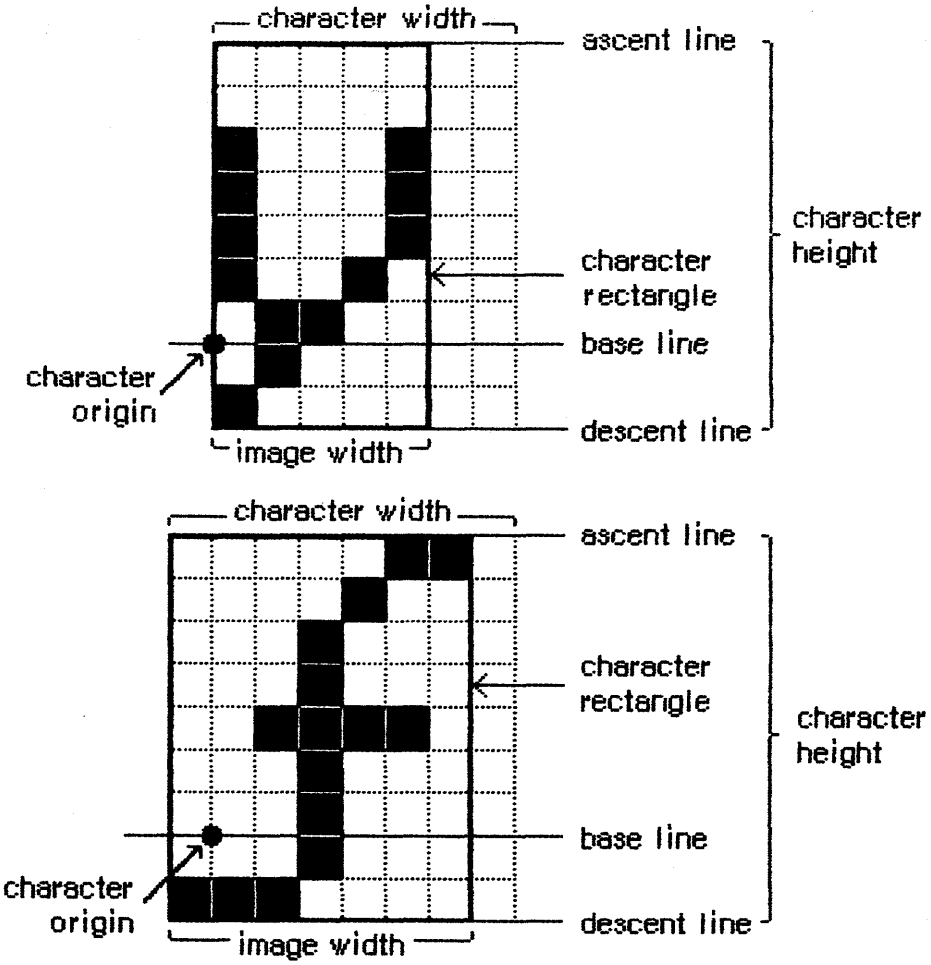


Figure 5. Character Images

The base line is a horizontal line coincident with the bottom of each character, excluding descenders. The character origin is a point on the base line used as a reference location for drawing the character. Conceptually the base line is the line that the pen is on when it starts drawing a character, and the characer origin is the point where the pen starts drawing.

The character rectangle is a rectangle enclosing the character image; its sides are defined by the image width and the character height:

- The image width is simply the horizontal extent of the character image, which varies among characters in the font. It may or may not include space on either side of the character; to minimize the amount of memory required to store the font, it should not include space.

- The character height is the number of pixels from the ascent line to the descent line (which is the same for all characters in the font).

The image width is different from the character width, which is the distance to move the pen from this character's origin to the next while drawing--in other words, the image width plus the amount of blank space to leave before the next character. The character width may be Ø, in which case the character that follows will be superimposed on this character (useful for accents, underscores, and so on). Characters whose image width is Ø (such as a space) can have a nonzero character width.

Characters in a proportional font all have character widths proportional to their image width, whereas characters in a fixed-width font all have the same character width.

Characters can kern; that is, they can overlap adjacent characters. The first character in Figure 5 above doesn't kern, but the second one kerns left.

In addition to the terms used to describe individual characters, there are terms describing features of the font as a whole (see Figure 6).

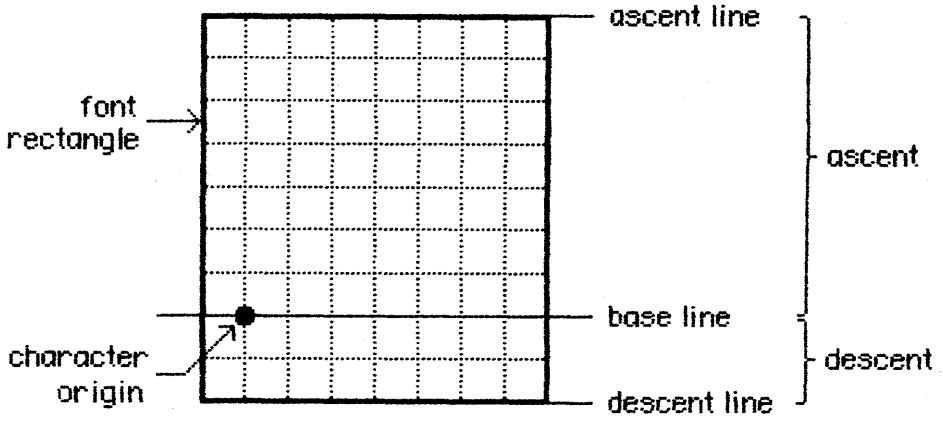


Figure 6. Features of Fonts

The font rectangle is somewhat analogous to a character rectangle. Imagine that all the character images in the font are superimposed with their origins coinciding. The smallest rectangle enclosing all the superimposed images is the font rectangle.

The ascent is the distance from the base line to the top of the font rectangle, and the descent is the distance from the base line to the bottom of the font rectangle.

The character height is the vertical extent of the font rectangle. The maximum character height is 127 pixels. The maximum width of the font rectangle is 254 pixels.

The leading is the amount of blank space to draw between lines of single-spaced text--the number of pixels between the descent line of one line of text and the ascent line of the next line of text.

Finally, for each character in a font there's a character offset. As illustrated in Figure 7, the character offset is simply the difference in position of the character rectangle for a given character and the font rectangle.

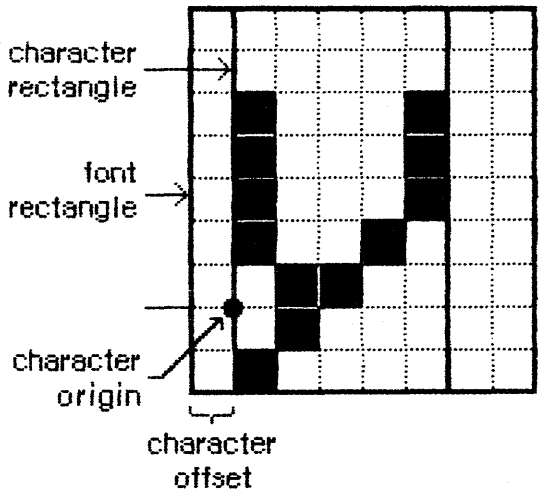


Figure 7. Character Offset

Every font has a bit image that contains a complete sequence of all its character images (see Figure 8 on the following page). The number of rows in the bit image is equivalent to the character height. The character images in the font are stored in the bit image as though the characters were laid out horizontally (in ASCII order, by convention) along a common base line.

The bit image doesn't have to contain a character image for every character in the font. Instead, any characters marked as being missing from the font are omitted from the bit image. When QuickDraw tries to draw such characters, a missing symbol is drawn instead. The missing symbol is stored in the bit image after all the other character images.

(warning)

Every font **must** have a missing symbol. The characters with ASCII codes Ø (NUL), $Ø9 (horizontal tab), and $ØD (return) must **not** be missing from the font; usually they'll be zero-length, but you may want to store a space for the tab character.

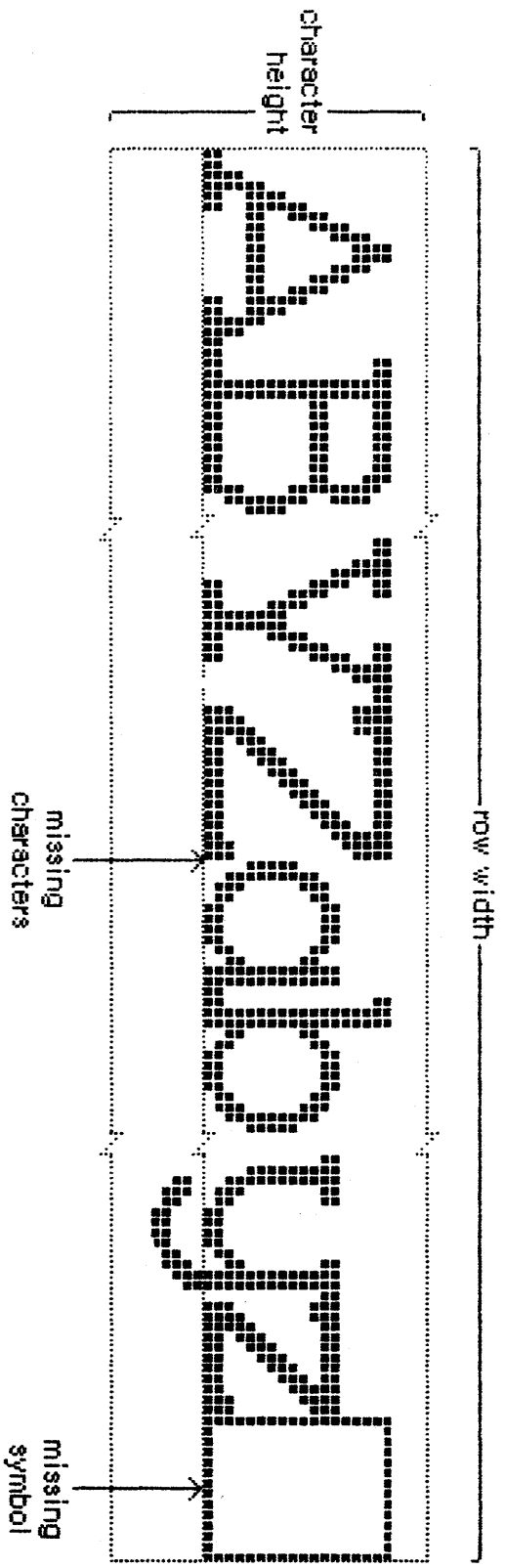


Figure 8. Partial Bit Image for a Font

Font Records

The information describing a font is contained in a data structure called a font record, which contains the following:

- the font type (fixed-width or proportional)
- the ASCII code of the first character and the last character in the font
- the maximum character width and maximum amount any character kerns
- the character height, ascent, descent, and leading
- the bit image of the font
- a location table, which is an array of words specifying the location of each character image within the bit image
- an offset/width table, which is an array of words specifying the character offset and character width for each character in the font.

For every character, the location table contains a word that specifies the bit offset to the location of that character's image in the bit image. The entry for a character missing from the font contains the same value as the entry for the next character. The last word of the table contains the offset to one bit beyond the end of the bit image (that is, beyond the character image for the missing symbol). The image width of each character is determined from the location table by subtracting the bit offset to that character from the bit offset to the next character in the table.

There's also one word in the offset/width table for every character: the high-order byte specifies the character offset and the low-order byte specifies the character width. Missing characters are flagged in this table by a word value of -1. The last word is also -1, indicating the end of the table.

Figure 9 illustrates a sample location table and offset/width table corresponding to the bit image in Figure 6.

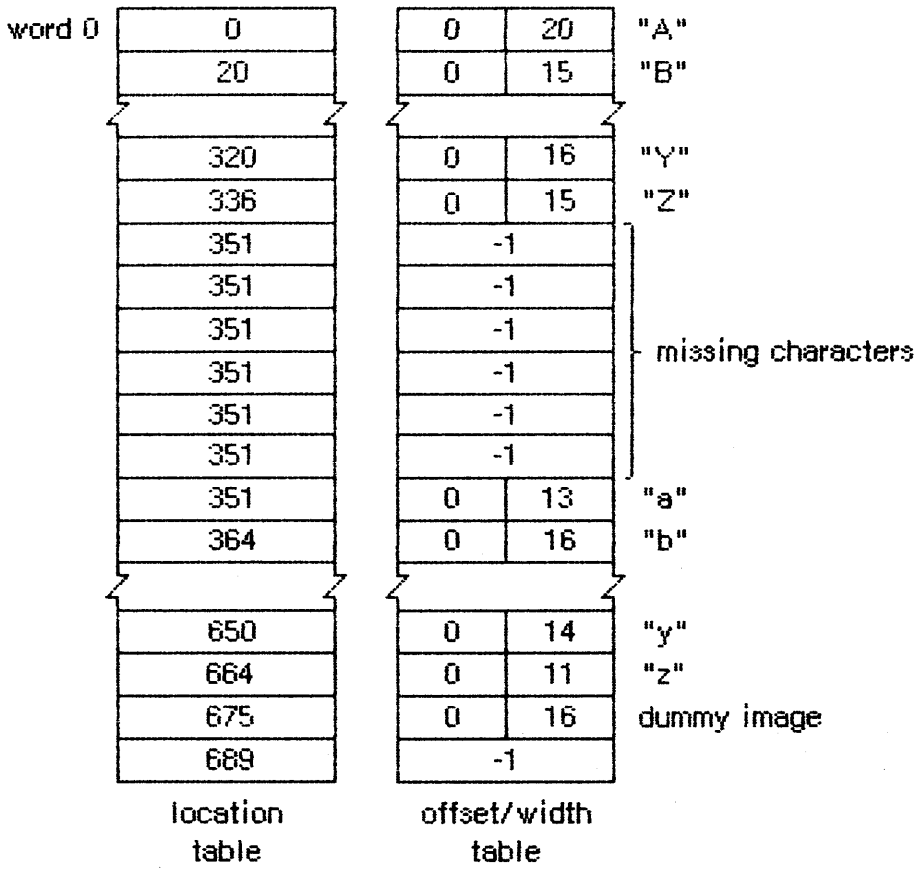


Figure 9. Sample Location Table and Offset/Width Table

A font record is referred to by a handle that you can get by calling the SwapFont function or the Resource Manager function GetResource. The data type for a font record is as follows:

```
TYPE FontRec = RECORD
                 fontType:  INTEGER;   {font type}
                 firstChar: INTEGER;   {ASCII code of first character}
                 lastChar:  INTEGER;   {ASCII code of last character}
                 widMax:    INTEGER;   {maximum character width}
                 kernMax:   INTEGER;   {maximum character kern}
                 nDescent:  INTEGER;   {negative of descent}
                 fRectWid:  INTEGER;   {width of font rectangle}
                 chHeight:  INTEGER;   {character height}
                 owTLoc:    INTEGER;   {offset to offset/width table}
                 ascent:    INTEGER;   {ascent}
                 descent:   INTEGER;   {descent}
                 leading:   INTEGER;   {leading}
                 rowWords:  INTEGER;   {row width of bit image / 2}
               { bitImage:  ARRAY [1..rowWords, 1..chHeight] OF INTEGER; }
                                       {bit image}
               { locTable:  ARRAY [firstChar..lastChar+2] OF INTEGER;    }
                                       {location table}
               { owTable:   ARRAY [firstChar..lastChar+2] OF INTEGER;    }
                                       {offset/width table}
                END;
```

(note)
The variable-length arrays appear as comments because they're not valid Pascal syntax; they're used only as conceptual aids.

The fontType field must contain one of the following predefined constants:

```
CONST propFont  = $9ØØØ;  {proportional font}
      fixedFont = $BØØØ;  {fixed-width font}
```

The values in the widMax, kernMax, nDescent, fRectWid, chHeight, ascent, descent, and leading fields all specify a number of pixels. KernMax indicates the largest number of pixels any character kerns, and should always be negative or Ø, because kerning is specified by negative numbers (the kerned pixels are to the left of the character origin). NDescent must be set to the negative of the descent.

The owTLoc field contains a word offset from itself to the offset/width table; it's equivalent to

```
4 + (rowWords * chHeight) + (lastChar - firstChar + 3) + 1
```

(warning)
Remember, the offset and row width in a font record are given in **words**, not bytes.

Normally, the Resource Editor will change the fields in a font record for you. You shouldn't have to change any fields unless you edit the font without the aid of the Resource Editor.

Assembly-language note: The global variable romFontØ contains a handle to the font record for the system font.

Font Widths

A resource can be defined that consists of only the character widths and general font information--everything but the font's bit image and location table. If there is such a resource, it will be read in whenever QuickDraw doesn't need to draw the text, such as when you call one of the routines CharWidth, HidePen, or OpenPicture (which calls HidePen). The FontRec data type described above, minus the rowWords, bitImage, and locTable fields, reflects the structure of this type of resource. The owTLoc field will contain 4, and the fontType field will contain the following predefined constant:

```
CONST fontWid = $ACBØ;  {font width data}
```

How QuickDraw Draws Text

This section provides a conceptual discussion of the steps QuickDraw takes to draw characters (without scaling or stylistic variations such as bold and outline). Basically, QuickDraw simply copies the character image onto the drawing area at a specific location.

1. Take the initial pen location as the character origin for the first character.

2. Check the word in the offset/width table for the character to see if it's -1. The word to check is entry (charCode - firstChar), where charCode is the ASCII code of the character to be drawn.

   2a. The character exists if the entry in the offset/width table isn't -1. Determine the character offset and character width from the bytes of this same word. Find the character image at the location in the bit image specified by the location table. Calculate the image width by subtracting this word from the succeeding word in the location table. Determine the number of pixels the character kerns by subtracting kernMax from the character offset.

   2b. The character is missing if the entry in the offset/width table is -1; information about the missing symbol is needed. Determine the missing symbol's character offset and character width from the next-to-last word in the offset/width table. Find the missing symbol at the location in the bit image specified by the next-to-last word in the location table (lastChar - firstChar + 1). Calculate the image width by

subtracting the next-to-last word in the location table from the last word (lastChar - firstChar + 2). Determine the number of pixels the missing symbol kerns by subtracting kernMax from the character offset.

3. Move the pen to the left the number of pixels that the character kerns. Move the pen up the number of pixels specified by the ascent.

4. If the fontType field is fontWid, skip to step 5; otherwise, copy each row of the character image onto the screen or paper, one row at a time. The number of bits to copy from each word is given by the image width, and the number of words is given by the chHeight field.

5. If the fontType field is fontWid, move the pen to the right the number of pixels specified by the character width. If fontType is fixedFont, move the pen to the right the number of pixels specified by the widMax field.

6. Return to step 2.

FONTS IN A RESOURCE FILE

This section contains details about fonts in resource files that most programmers need not be concerned about, since they can use the Resource Editor *** eventually *** to define fonts. It's included here to give background information to those who are interested.

Every size of a font is stored as a separate resource. The resource type for a font is 'FONT'. The resource data for a font is simply a font record:

Number of bytes	Contents
2 bytes	FontType field of font record
2 bytes	FirstChar field of font record
2 bytes	LastChar field of font record
2 bytes	WidMax field of font record
2 bytes	KernMax field of font record
2 bytes	NDescent field of font record
2 bytes	FRectWid field of font record
2 bytes	ChHeight field of font record
2 bytes	OWTLoc field of font record
2 bytes	Ascent field of font record
2 bytes	Descent field of font record
2 bytes	Leading field of font record
2 bytes	RowWords field of font record
n bytes	Bit image of font n = 2 * rowWords * chHeight
m bytes	Location table of font m = 2 * (lastChar - firstChar + 3)
m bytes	Offset/width table of font m = 2 * (lastChar - firstChar + 3)

The resource type 'FWID' is used to store only the character widths and general information for a font; its resource data is a font record without the rowWords field, bit image, and location table.

As shown in Figure 1Ø, the resource ID of a font is composed of two parts: bits 7 to 15 are the font number, and bits Ø to 6 are the font size. Thus the resource ID corresponding to a given font number and size is

(128 * font number) + font size

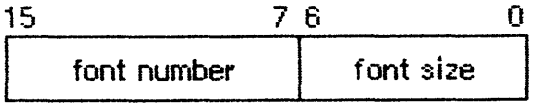


Figure 1Ø. Resource ID for a Font

Since Ø is not a valid font size, the resource ID having Ø in the size field is used to provide only the name of the font: the name of the resource is the font name. For example, for a font named Griffin and numbered 4ØØ, the resource naming the font would have a resource ID of 512ØØ and the resource name 'Griffin'. Size 1Ø of that font would be stored in a resource numbered 5121Ø.

Font numbers Ø through 127 are reserved for fonts provided by Apple, and font numbers 128 through 383 are reserved for assignment, by Apple, to software vendors. Each font will be assigned a unique number, and that font number should be used to identify only that font (for example, font number 9 will always be Toronto). Font numbers 384 through 511 are available for your use in whatever way you wish.

SUMMARY OF THE FONT MANAGER

Constants

```
CONST { Font numbers }

      systemFont = Ø;  {system font}
      applFont   = 1;  {application font}
      newYork    = 2;
      geneva     = 3;
      monaco     = 4;
      venice     = 5;
      london     = 6;
      athens     = 7;
      sanFran    = 8;
      toronto    = 9;

      { Font types }

      propFont  = $9ØØØ;  {proportional font}
      fixedFont = $BØØØ;  {fixed-width font}
      fontWid   = $ACBØ;  {font width data}
```

Data Types

```
TYPE FMInput = PACKED RECORD
                 family:   INTEGER; {font number}
                 size:     INTEGER; {font size}
                 face:     Style;   {character style}
                 needBits: BOOLEAN; {TRUE if drawing}
                 device:   INTEGER; {device-specific information}
                 numer:    Point;   {numerators of scaling factors}
                 denom:    Point    {denominators of scaling factors }
               END;
```

```
FMOutPtr = ^FMOutput;
FMOutput = PACKED RECORD
             errNum:     INTEGER;     {not used}
             fontHandle: Handle;      {handle to font record}
             bold:       Byte;        {bold factor}
             italic:     Byte;        {italic factor}
             ulOffset:   Byte;        {underline offset}
             ulShadow:   Byte;        {underline shadow}
             ulThick:    Byte;        {underline thickness}
             shadow:     Byte;        {shadow factor}
             extra:      SignedByte;  {width of style}
             ascent:     Byte;        {ascent}
             descent:    Byte;        {descent}
             widMax:     Byte;        {maximum character width}
             leading:    SignedByte;  {leading}
             unused:     Byte;        {not used}
             numer:      Point;       {numerators of scaling factors}
             denom:      Point        {denominators of scaling factors}
           END;

FontRec = RECORD
            fontType:  INTEGER;   {font type}
            firstChar: INTEGER;   {ASCII code of first character}
            lastChar:  INTEGER;   {ASCII code of last character}
            widMax:    INTEGER;   {maximum character width}
            kernMax:   INTEGER;   {maximum character kern}
            nDescent:  INTEGER;   {negative of descent}
            fRectMax:  INTEGER;   {width of font rectangle}
            chHeight:  INTEGER;   {character height}
            owTLoc:    INTEGER;   {offset to offset/width table}
            ascent:    INTEGER;   {ascent}
            descent:   INTEGER;   {descent}
            leading:   INTEGER;   {leading}
            rowWords:  INTEGER;   {row width of bit image / 2}
          { bitImage:  ARRAY [1..rowWords, 1..chHeight] OF INTEGER; }
                                  {bit image}
          { locTable:  ARRAY [firstChar..lastChar+2] OF INTEGER;    }
                                  {location table}
          { owTable:   ARRAY [firstChar..lastChar+2] OF INTEGER     }
                                  {offset/width table}
           END;
```

Routines

Initializing the Font Manager

```
PROCEDURE InitFonts;
```

Getting Font Information

```
PROCEDURE GetFontName (fontNum: INTEGER; VAR theName: Str255);
PROCEDURE GetFNum     (fontName: Str255; VAR theNum: INTEGER);
FUNCTION  RealFont    (fontNum: INTEGER; size: INTEGER) : BOOLEAN;
```

Keeping Fonts in Memory

```
PROCEDURE SetFontLock (lockFlag: BOOLEAN);
```

Advanced Routine

```
FUNCTION SwapFont (inRec: FMInput) : FMOutPtr;
```

Assembly-Language Information

Constants

```
; Font numbers

sysFont     .EQU    Ø   ;system font
applFont    .EQU    1   ;application font
newYork     .EQU    2
geneva      .EQU    3
monaco      .EQU    4
venice      .EQU    5
london      .EQU    6
athens      .EQU    7
sanFran     .EQU    8
toronto     .EQU    9

; Font types

propFont      .EQU    $9ØØØ   ;proportional font
fixedFont     .EQU    $BØØØ   ;fixed-width font
fontWid       .EQU    $ACBØ   ;font width data

; Control and Status call code

fMgrCtl1      .EQU    8       ;code used to get and modify font
                              ; characterization table
```

Font Input Record Data Structure

```
fmInFamily          Font number
fmInSize            Font size
fmInFace            Character style
```

fmInNeedBits	TRUE if drawing
fmInDevice	Device-specific information
fmInNumer	Numerators of scaling factors
fmInDenom	Denominators of scaling factors

Font Output Record Data Structure

*** these offsets don't exist yet ***	
fmOutError	Not used
fmOutFontHandle	Handle to font record
fmOutBold	Bold factor
fmOutItalic	Italic factor
fmOutUlOffset	Underline offset
fmOutUlShadow	Underline shadow
fmOutUlThick	Underline thickness
fmOutShadow	Shadow factor
fmOutExtra	Width of style
fmOutAscent	Ascent
fmOutDescent	Descent
fmOutWidMax	Maximum character width
fmOutLeading	Leading
fmOutUnused	Not used
fmOutNumer	Numerators of scaling factors
fmOutDenom	Denominators of scaling factors

Font Record Data Structure

fFormat	Font type
fMinChar	ASCII code of first character
fMaxChar	ASCII code of last character
fMaxWd	Maximum character width
fBBOX	Maximum character kern
fBBOY	Negative of descent
fBBDX	Width of font rectangle
fBBDY	Character height
fLength	Offset to offset/width table
fAscent	Ascent
fDescent	Descent
fLeading	Leading
fRaster	Row width of bit image / 2

Special Macro Names

Routine name	Macro name
GetFontName	_GetFName
SwapFont	_FMSwapFont

Variables

Name	Size	Contents
apFontID	2 bytes	Font number of application font
fScaleDisable	1 byte	Nonzero to disable scaling
romFontØ	4 bytes	Handle to font record for system font

GLOSSARY

application font: The font your application will use unless you specify otherwise--Geneva, by default.

ascent: The vertical distance from a font's base line to its ascent line.

base line: A horizontal line coincident with the bottom of each character in a font, excluding descenders.

character height: The vertical distance from a font's ascent line to its descent line.

character image: The bit image that defines a character.

character offset: The horizontal separation between a character rectangle and a font rectangle.

character origin: The point on a base line used as a reference location for drawing a character.

character rectangle: A rectangle enclosing an entire character image. Its sides are defined by the image width and the character height.

character width: The distance to move the pen from one character's origin to the next; equivalent to the image width plus the amount of blank space to leave before the next character.

descent: The vertical distance from a font's base line to its descent line. fixed-width font: A font whose characters all have the same width.

font: The complete set of characters of one typeface.

font characterization table: A table of parameters in a device driver that specifies how best to adapt fonts to that device.

font number: The number by which you identify a font to QuickDraw or the Font Manager.

font record: A data structure that contains all the information describing a font.

font rectangle: The smallest rectangle enclosing all the character images in a font, if the images were all superimposed over the same character origin.

font size: The size of a font in points; equivalent to the distance between the ascent line of one line of text and the ascent line of the next of line of single-spaced text.

image width: The horizontal extent of a character image.

kern: To draw part of a character so that it overlaps an adjacent character.

leading: The amount of blank vertical space between the descent line of one line of text and the ascent line of the next line of single-spaced text.

location table: An array of words (one for each character in a font) that specifies the location of each character's image in the font's bit image.

missing symbol: A character to be drawn in case of a request to draw a character that's missing from a particular font.

offset/width table: An array of words that specifies the character offsets and character widths of all characters in a font.

point: The intersection of a horizontal grid line and a vertical grid line on the coordinate plane, defined by a horizontal and a vertical coordinate; also, a typographical term meaning approximately 1/72 inch.

proportional font: A font whose characters all have character widths that are proportional to their image width.

scaling factor: A value, given as a fraction, that specifies the amount a character should be stretched or shrunk before it's drawn.

style: Same as character style.

system font: The font that the system uses (in menus, for example). Its name is Chicago.

system font size: The size of text drawn by the system in the system font; 12 points.

MACINTOSH USER EDUCATION

The Event Manager: A Programmer's Guide /EMGR/EVENTS

See also: Macintosh User Interface Guidelines
Macintosh Operating System Reference Manual
QuickDraw: A Programmer's Guide
The Window Manager: A Programmer's Guide
The Desk Manager: A Programmer's Guide
The Menu Manager: A Programmer's Guide
The Control Manager: A Programmer's Guide

Modification History: First Draft (ROM 4) S. Chernicoff 6/2Ø/83

ABSTRACT

The Macintosh Event Manager is your program's link to its human user, allowing it to monitor the user's actions with the mouse, keyboard, and keypad. A typical Macintosh application program is event-driven: it 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. The Event Manager is also used for various purposes within the Toolbox itself, such as to coordinate the ordering and display of windows on the screen. Finally, you can use the Event Manager as a means of communication between parts of your own program.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Event Manager, the part of the Macintosh User Interface Toolbox that allows your program to monitor the user's actions with the mouse, keyboard, and keypad. *** Eventually it will become part of a larger manual describing the entire Toolbox. *** The Event Manager is also used for various purposes within the Toolbox itself, such as to coordinate the ordering and display of windows on the screen. Finally, you can use the Event Manager as a means of communication between parts of your own program.

(eye)

> This manual describes version 4 of the Macintosh ROM. If you're using a different version, the Event Manager may not work exactly as described here.

Actually, there are two Event Managers: one in the Operating System and one in the Toolbox. The Toolbox Event Manager calls the one in the Operating System and serves as an interface between it and your application program; it also adds some features that aren't present at the Operating System level, such as the window management facilities mentioned above. This manual describes the Toolbox Event Manager, which is ordinarily the one your program will be dealing with. All references to "the Event Manager" should be understood to refer to the Toolbox Event Manager. For information on the Operating System's Event Manager, see the Macintosh Operating System Reference Manual.

Like all Toolbox documentation, this manual assumes you are familiar with the Macintosh User Interface Guidelines and with Lisa Pascal. You should also have at least a general notion of what the Window Manager, Desk Manager, Menu Manager, Control Manager, and Resource Manager do. It would also be helpful to have some familiarity with a Macintosh application program as an illustration of the concepts presented here.

The manual begins with an introduction to the Event Manager and what you can do with it. It then discusses the various types of event, their relative priority, and how the user's keyboard actions, in particular, are reported in the form of events. Next come sections on the structure of event records, which contain all the pertinent information about each event, and event masks, which some of the Event Manager routines expect as parameters.

A section on using the Event Manager introduces its routines and tells how they fit into the flow of your application program. This is followed by detailed descriptions of all Event Manager procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that will not be of interest to all readers. Special information is given on the Event Manager's journaling mechanism, which allows your program's interactions with the user to be recorded and played back later; on the format used in resource files for storing a keyboard configuration, which determines

what character each key on the keyboard stands for; and on how to use the Event Manager routines from assembly language.

Finally, there are an appendix containing detailed information on the standard Macintosh character set and keyboard configuration, a quick-reference summary of the Event Manager data structures and routines, and a glossary of terms used in this manual.

ABOUT THE EVENT MANAGER

The Macintosh Event Manager is your program's link to its human user. Whenever the user presses the mouse button, types on the keyboard or keypad, or inserts a disk in a disk drive, your program is notified by means of an event. A typical Macintosh application progam is event-driven: it 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.

Although the Event Manager's primary purpose is to monitor the user's actions and pass them to your program in an orderly way, it also serves as a convenient mechanism for sending signals from one part of a program to another. For instance, the Window Manager uses events to coordinate the ordering and display of windows as the user activates and deactivates them and moves them around on the Macintosh screen. You can also define your own types of event and use them in any way your application calls for.

Events waiting to be processed are kept in the event queue. In principle, the event queue is a FIFO (first-in-first-out) list: events are added to the queue (posted) at one end and retrieved from the other. You can think of the queue as a funnel that collects events from a variety of sources and feeds them to your program on demand, in the order they occurred. (There are a few exceptions to the strict FIFO ordering, which will be discussed later.)

(eye)

> The event queue has a limited capacity *** (currently 3Ø events, but may change) ***. When the queue becomes full, the Event Manager begins throwing out old events to make room for new ones as they're posted. The event thrown out is always the oldest one in the queue.

Using the Event Manager, your program can:

- Retrieve events one at a time from the event queue
- Control which types of event get posted and which are ignored
- Post events of its own
- Read the current state of the keyboard, keypad, and mouse button

- Monitor the location of the mouse

- Read the system clock to find out how much time has elapsed since the system was last started up

Another important service provided by the Event Manager is journaling. This feature enables your program to record all its interactions with the Event Manager and play them back later.

EVENT TYPES

Events are of various types, depending on their origin and meaning. Some report actions by the user, some are generated by the Window Manager, some *** (not yet implemented) *** arise in the Macintosh's low-level input/output drivers, and some may be generated by your program itself for its own purposes. Some events are handled by the Desk Manager before your program ever sees them; others are left for your program to handle in its own way.

The most important event types, the ones the Event Manager was created to handle, are those that record actions by the user:

- Mouse down and mouse up events occur when the user presses or releases the mouse button.

- Key down and key up events occur when the user presses or releases a key on the keyboard or keypad. The Event Manager also automatically generates auto-key events when the user presses and holds down a repeating key. Together, these three event types are called keyboard events.

- Disk inserted events occur when the user inserts a disk into a disk drive.

- Abort events occur when the user presses a special combination of keys. *** Tentatively the combination is Command-period (Command-.), but this may change; there's also some possibility that more than one key combination will be provided to interrupt a running program in different ways or for different purposes. *** An abort event signals the program to stop whatever it's doing and return control directly to the user, allowing the user to interrupt a time-consuming process or regain control of a runaway program. An abort event can also be generated by the Event Manager's own journaling mechanism, signaling the program to reset itself to some standard initial state before replaying a journal.

(hand)
Mere movements of the mouse are not reported as events. If necessary, your program can keep track of them by periodically asking the Event Manager for the current location of the mouse.

The following event types are used by the Window Manager to coordinate the display of windows on the screen:

- Activate events are generated whenever an inactive window becomes active or vice versa. They generally occur in pairs (that is, one window is deactivated and another activated at the same time).

- Update events occur when a window's contents need to be redrawn, usually as a result of the user's opening, closing, activating, or moving a window.

Two more event types (I/O driver events and network events) are reserved for use by the low-level input/output system. *** At present, these types are not used at all. *** In addition, your program can define as many as four event types of its own and use them for whatever purposes you like.

One final type of event is the null event, which is what the Event Manager returns if it has no other events to report.

PRIORITY OF EVENTS

It was stated earlier that in principle the event queue is a FIFO list--that is, events are retrieved from the queue in the order they were originally posted. Actually, the way in which various types of event are generated and detected causes some to have higher priority than others. Furthermore, when you ask the Event Manager for an event, you can specify a particular type or types that are of interest. This can also alter the strict FIFO order, by causing some events to be passed over in favor of others that were actually posted later. Everything said in the following discussion is understood to be limited to the event types you've specifically requested in your Event Manager call.

The Event Manager always returns the highest-priority event available of the requested type(s). The priority ranking is as follows:

1. Activate (window becoming inactive before window becoming active)

2. Mouse down, mouse up, key down, key up, disk inserted, abort, network, I/O driver, application-defined (all in FIFO order)

3. Auto-key

4. Update (in front-to-back order)

5. Null

Activate events take priority over all others; they are detected in a special way, and are never actually placed in the event queue. The Event Manager checks for pending activate events before looking in the event queue, so it will always return such an event if one is available. Because of the special way activate events are detected,

there can never be more than two such events pending at the same time: one for a window becoming inactive and another for a window becoming active. If there's one of each, the event for the window becoming inactive is reported first.

Category 2 includes most of the possible event types. Within this category, events are normally retrieved from the queue in the order they were posted.

If no event is available in categories 1 and 2, the Event Manager next checks to see whether the appropriate conditions hold for an auto-key event. (These conditions are described in detail in the next section.) If so, it generates one and returns it to your program.

Next in priority are update events. Like activate events, these are not placed in the event queue, but are detected in another way. If no higher-priority event is available, the Event Manager checks for windows whose contents need to be redrawn. If it finds one, it generates and returns an update event for that window. Windows are checked in the order in which they're displayed on the screen, from front to back, so if two or more windows need to be updated, an update event will be generated for the frontmost such window.

Finally, if no other event is available, the Event Manager returns a null event.

KEYBOARD EVENTS

Every key on the Macintosh keyboard and the optional keypad generates key down and key up events when pressed and released. (Exceptions are the modifier keys--Shift, Caps Lock, Command *** name may change ***, and Option. These keys are treated specially, as described below, and generate no keyboard events of their own.) In addition, the Event Manager itself generates auto-key events whenever you request an event and all of the following conditions apply:

- No higher-priority event of the requested type(s) is available
- The user is currently holding down a key other than a modifier key
- The appropriate time interval (see below) has elapsed since the last keyboard event
- Auto-key events are one of the types you've requested
- Auto-key events are one of the types currently being posted into the event queue

Two different time intervals are taken into account. Auto-key events begin to be generated after a certain initial delay has elapsed since the original key down event (that is, since the key was originally pressed). Thereafter, they are generated each time a certain repeat

interval has elapsed since the last auto-key event. The initial settings for these two intervals are 16 ticks (sixtieths of a second) for the initial delay and 4 ticks for the repeat interval. The user can adjust these settings to individual preference with the control panel desk accessory.

When the user presses, holds down, or releases a key, the resulting keyboard event identifies the key in two different ways: with a key code designating the key itself and a character code designating the character the key stands for. Character codes are given in the extended version of ASCII (the American Standard Code for Information Interchange) used by Macintosh and Lisa; see the Appendix for further information.

The association between keys and characters is defined by a keyboard configuration. The particular character a key generates depends on three things:

- The key itself
- The keyboard configuration currently in effect
- Which, if any, of the modifier keys were held down when the key was pressed

As mentioned earlier, the modifier keys don't generate keyboard events of their own. Instead, they modify the meaning of the other keys by changing the character codes that those keys generate. For example, under the standard Macintosh keyboard configuration, the "C" key generates a lowercase letter c when pressed by itself; when pressed with the Shift or Caps Lock key down, it generates a capital C; with the Option key down, a lowercase c with a cedilla (ç), used in French, Portuguese, and a few other foreign languages; and with Option and Shift or Option and Caps Lock down, a capital C with a cedilla (Ç). The state of each of the option keys is also reported in a field of the event record (see next section), where your program can examine it directly.

Keyboard configurations are handled as resources and stored in resource files. The standard keyboard configuration gives each key its normal ASCII character code according to the standard Macintosh keyboard layout, as shown in the Appendix. When the Option key is held down, most keys generate special characters with codes between 128 and 255 ($8Ø and $FF), included in the extended character set for business, scientific, and international use.

(hand)

Notice that under the standard keyboard configuration only the Shift, Caps Lock, and Option keys actually modify the character a key stands for: the Command key has no effect on the character code generated. (Keyboard configurations other than the standard may take the Command key into account.) Similarly, character codes for the keypad are affected only by the Shift key. To

find out whether the Command key was down at the time of an event (or Caps Lock or Option in the case of one generated from the keypad), you have to examine the appropriate field of the event record.

Normally you'll just want to use the standard keyboard configuration, which is read from the system resource file every time the Macintosh is started up. Other keyboard configurations can be used to reconfigure the keyboard for foreign use or for nonstandard layouts such as the Dvorak arrangement. In rare cases, you may want to define your own keyboard configuration to suit your program's special needs. For information on how to install an alternate keyboard configuration or define one of your own, see "Resource Format for Keyboard Configurations" and "Notes for Assembly-Language Programmers", below.

EVENT RECORDS

Every event is represented internally by an event record containing all pertinent information about that event. The event record includes the following information:

- The type of event
- The time the event was posted
- The location of the mouse at the time the event was posted
- The state of the mouse button and modifier keys at the time the event was posted
- Any additional information required for a particular type of event, such as which key the user pressed or which window is being activated

This information is filled into the event record for every event--even for null events, which just mean that nothing special has happened.

Event records are defined as follows:

```
TYPE EventRecord = RECORD
                     what:      INTEGER;
                     message:   LongInt;
                     when:      LongInt;
                     where:     Point;
                     modifiers: INTEGER
                   END;
```

The what field contains an event code identifying the type of the event. The Event Manager can handle a maximum of 16 different event types, denoted by event codes from Ø to 15. The following standard event codes are built into the Event Manager as predefined constants:

```
CONST nullEvent   = Ø;   {null}
      mouseDown   = 1;   {mouse down}
      mouseUp     = 2;   {mouse up}
      keyDown     = 3;   {key down}
      keyUp       = 4;   {key up}
      autoKey     = 5;   {auto-key}
      updateEvt   = 6;   {update}
      diskEvt     = 7;   {disk inserted}
      activateEvt = 8;   {activate}
      abortEvt    = 9;   {abort}
      networkEvt  = 1Ø;  {network}
      driverEvt   = 11;  {I/O driver}
      app1Evt     = 12;  {application-defined}
      app2Evt     = 13;  {application-defined}
      app3Evt     = 14;  {application-defined}
      app4Evt     = 15;  {application-defined}
```

The when field contains the time the event was posted, in ticks (sixtieths of a second) since the system was last started up.

The where field gives the location of the mouse at the time the event was posted, expressed in global coordinates.

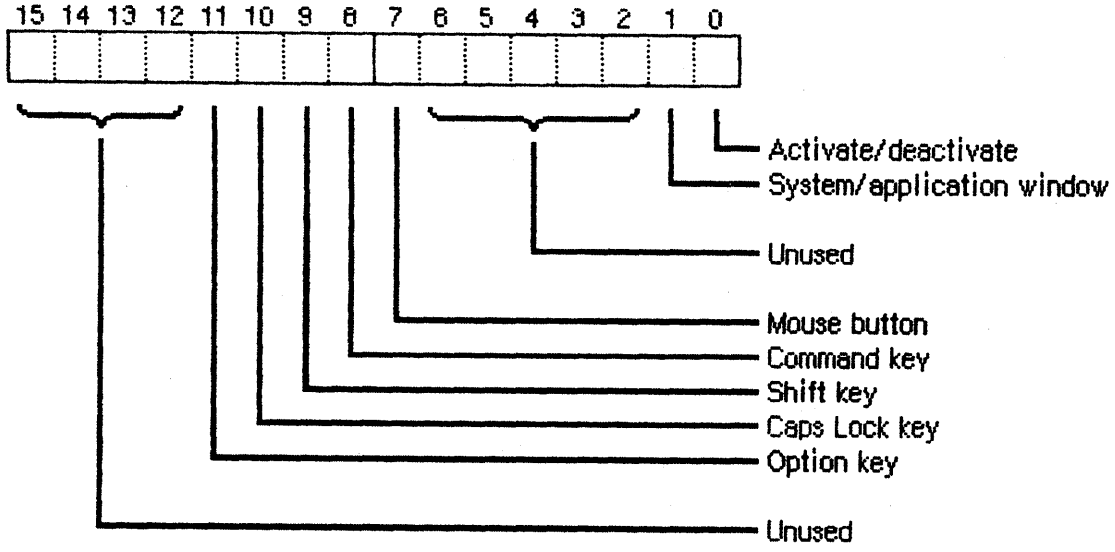


Figure 1. Modifier Bits

The modifiers field gives the state of the mouse button and the modifier keys at the time the event was posted, as shown below and in Figure 1. A 1 in any bit position means that that key or button was down; Ø means it was up. (Following the customary convention, the bit positions are numbered from right to left, starting from Ø at the low-order end; see Figure 1.)

Bit	Meaning
15-12	Unused
11	Option key
1Ø	Caps Lock key
9	Shift key
8	Command key *** (name may change) ***
7	Mouse button
6-2	Unused
1-Ø	Used only by activate events (see below)

For activate events, the low-order bit of the modifiers field (bit Ø) is set to 1 if a window is being activated, or to Ø if it is being deactivated. When one window is deactivated and another is activated at the same time (as is usually the case), bit 1 of the modifiers field is set to 1 if one of the windows involved belongs to your application program and the other is a system window (a window not created by your program, such as one containing a desk accessory); if they're both system or both application windows, this bit is set to Ø. You can use this information to take some special action when the active window changes from an application window to a system window or vice versa: for example, you might want to hide a menu or dim some of its items when a system window becomes active and restore them when control returns to one of your program's own windows.

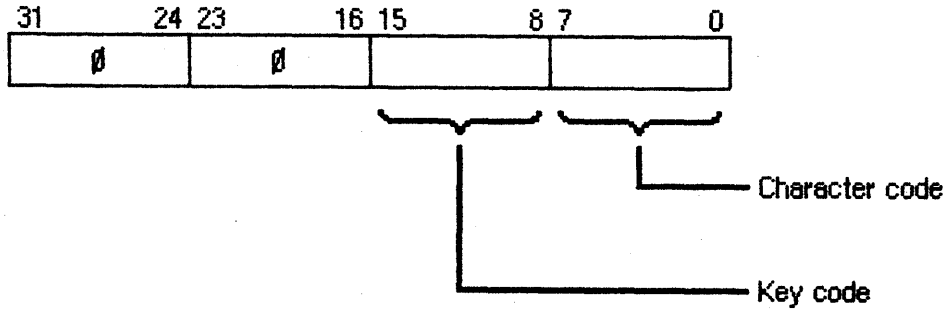


Figure 2. Event Message Format for Keyboard Events

The message field contains the event message, which conveys extra information specific to a particular event type:

- For keyboard events, the event message identifies the key that was pressed or released, as shown in Figure 2. The low-order byte (message MOD 256) contains the character code for the key, depending on the keyboard configuration currently in effect and on which, if any, of the modifier keys were held down. Under the

standard keyboard configuration this is just the normal ASCII code associated with the key, which is usually the information your program needs. The third byte (message DIV 256) gives the key code, useful in special cases (a music generator, for example) where you want to treat the keyboard as a set of buttons unrelated to specific characters. Detailed information on key and character codes for the standard Macintosh keyboard configuration is given in the Appendix. The first two bytes of the message are set to Ø.

- For disk inserted events, the event message gives the drive number of the disk drive: 1 for the Macintosh's built-in drive, 2 for the external drive, if any. Numbers greater than 2 denote additional disk drives connected through the serial port. By the time your program receives a disk inserted event, the system will already have attempted to mount the volume that was inserted. If for any reason the attempt was unsuccessful (for example, if the user has inserted an unformatted disk), the high-order word of the event message will contain the error code returned by the Operating System; see the Operating System manual for further details.

- For activate and update events, the event message is a pointer to the window affected.

- For abort events, the event message identifies the key that the user pressed in order to interrupt the program. The format is the same as described above for keyboard events. For abort events generated by the Event Manager's own journaling mechanism, the message field is set to Ø.

- For application-defined event types, you can use the event message for whatever information your application calls for.

- For mouse down, mouse up, and null events, the event message is meaningless and should be ignored.

EVENT MASKS

Several of the Event Manager routines can be restricted to a specific event type or group of types. For instance, instead of just requesting the next available event, you can ask specifically for the next keyboard event.

You specify which event types a particular Event Manager call applies to by supplying an event mask as a parameter. This is an integer in which each of the 16 bit positions stands for an event type, as shown in Figure 3. Notice that the bit position representing a given type corresponds to the event code for that type. For example, update events (type code 6) are specified by bit 6 of the mask, counting from Ø at the right (low-order) end. A 1 bit at that position means that this Event Manager call applies to update events; a Ø means it doesn't.

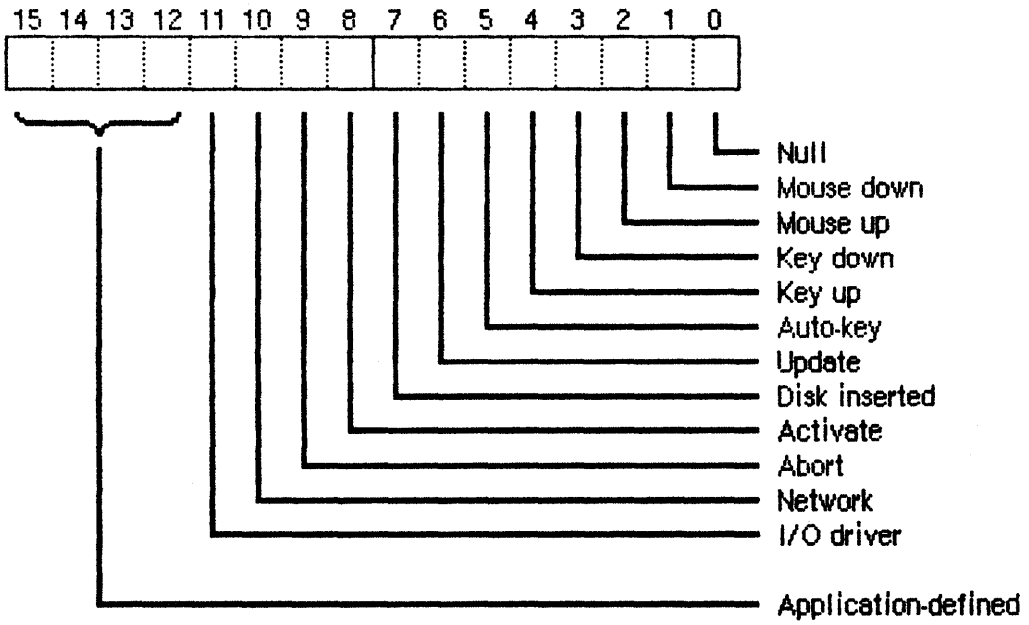


Figure 3. Event Mask

Masks for each single event type are built into the Event Manager as predefined constants:

```
CONST nullMask     = 1;       {null}
      mDownMask    = 2;       {mouse down}
      mUpMask      = 4;       {mouse up}
      keyDownMask  = 8;       {key down}
      keyUpMask    = 16;      {key up}
      autoKeyMask  = 32;      {auto-key}
      updateMask   = 64;      {update}
      diskMask     = 128;     {disk inserted}
      activMask    = 256;     {activate}
      abortMask    = 512;     {abort}
      networkMask  = 1Ø24;    {network}
      driverMask   = 2Ø48;    {I/O driver}
      appl Mask    = 4Ø96;    {application-defined}
      app2Mask     = 8192;    {application-defined}
      app3Mask     = 16384;   {application-defined}
      app4Mask     = -32768;  {application-defined}
```

There's also a predefined mask consisting of all 1 bits, to designate every event type:

```
CONST everyEvent = -1;
```

You can form any mask you need by combining these mask constants with integer addition and subtraction. For example, to specify any keyboard event, you can use a mask of

```
keyDownMask + keyUpMask + autoKeyMask
```

For any event except an update, you can use

```
everyEvent - updateMask
```

(hand)
Recommended programming practice is always to use an event mask of everyEvent unless there is a specific reason not to. This ensures that all events will be processed in their natural order.

In addition to the mask parameters to individual Event Manager routines, there's also a global system event mask, which controls which event types get posted into the event queue. Only those events corresponding to 1 bits in the system event mask are posted; those with Ø bits are ignored. When the system is started up, the system event mask is initially set to post all except key up events--that is, it is initialized to

```
everyEvent - keyUpMask
```

(Key up events are meaningless for most applications, and your program will usually want to ignore them anyway.) If necessary for your particular application, you can change the setting of the system event mask with the Event Manager procedure SetEventMask.

USING THE EVENT MANAGER

This section discusses how the Event Manager routines fit into the general flow of your program and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

Before using the Event Manager, you should call the Window Manager procedure InitWindows: parts of the Event Manager rely on the Window Manager's data structures and will not work properly unless those structures have been properly initialized. It's also usually a good idea to call FlushEvents(everyEvent,Ø), to empty the event queue of any stray events left over from before your program was started up (such as keystrokes typed to the Finder).

As noted earlier, most application programs are event-driven. Such programs typically have a main loop that repeatedly calls GetNextEvent to retrieve the next available event, then uses a CASE statement to decide what type of event it is and take whatever action is appropriate.

Your program is only expected to respond to those events that are directly related to its own operations. Events that are of interest only to the system, or that pertain only to system windows, are intercepted and handled by the Desk Manager, but are still reported back to your program by GetNextEvent. After calling GetNextEvent, you

should test its Boolean result to find out whether your program needs to respond to the event: TRUE means the event is of interest to your program, FALSE means you can ignore it.

(hand)

Events handled by the system include activate and update events for system windows; all keyboard and mouse up events when a system window is active, if the window contains a desk accessory that is prepared to handle the event; and network events if there's a desk accessory present that will handle them. Further details are given in the Desk Manager manual.

On receiving a mouse down event, you should first call the Window Manager function FindWindow to find out where on the screen the mouse button was pressed; you can then respond in whatever way is appropriate. Depending on the part of the screen the button was pressed in, this may involve calls to Toolbox routines such as the Menu Manager function MenuSelect, the Desk Manager procedure SystemClick, the Window Manager routines SelectWindow, DragWindow, GrowWindow, and TrackGoAway, and the Control Manager routines FindControl, TrackControl, and DragControl. See the relevant Toolbox manuals for details.

(hand)

If your program attaches some special significance to double mouse clicks, you can detect them by comparing the time and location of each mouse down event with those of the previous such event. If the two events are sufficiently close to each other in time and space--separated by not more than, say, half a second (30 ticks) and three pixels--you can consider them a double click and respond accordingly.

When one of your own windows is active, you should respond to keyboard and mouse up events in whatever way your application calls for. For example, when the user types a character on the keyboard, you might want to insert that character into the document displayed in an active document window. For keyboard events, you should first check the modifiers field to see whether the character was typed with the Command key held down: if so, the user may have been choosing a menu item by typing its keyboard equivalent. To find out, pass the character that was typed to the Menu Manager function MenuKey. If that character, combined with the Command key, stands for a menu item, MenuKey will return a nonzero result identifying the item. You can then do whatever is appropriate to respond to that menu item, just as if the user had chosen it with the mouse. If MenuKey's result is 0, the user has typed a key combination that has no menu equivalent; your program may then want to respond in some other way.

(hand)

Under the Macintosh User Interface Guidelines, the keyboard's usual auto-repeat property doesn't apply to Command-key combinations that stand for menu items. When

you receive a nonzero result from MenuKey, you should execute the corresponding menu command only if the event you're responding to was a mouse down event; if it was an auto-key event, just ignore it and go on to the next event.

When you receive an activate event for one of your own windows, the Window Manager will already have done all of the normal "housekeeping" associated with the event, such as highlighting or unhighlighting the window. You can then take any further action of your own that your application may require, such as showing or hiding a scroll bar or highlighting or unhighlighting a selection.

On receiving an update event for one of your own windows, you should usually call the Window Manager procedure BeginUpdate, redraw the window's contents, then call EndUpdate.

When you receive a disk inserted event, the Desk Manager will already have responded to the event by attempting to mount the new volume just inserted in the disk drive. Usually there's nothing more for your program to do, but GetNextEvent returns TRUE anyway, giving you an opportunity to take some further action if your application demands it. If the attempt to mount the volume was unsuccessful, there will be a nonzero error code in the high-order word of the event message; in this case you might want to take some special action, such as displaying an alert box containing an error message.

If the event you receive is an abort event, first check to see whether it was generated by the user or by the Event Manager's own journaling mechanism. For user-generated abort events, your program should stop whatever it's doing and return to its main loop to process the next available event; for those that originate in the journaling mechanism, it should reset its internal state as appropriate to prepare for replaying a journal.

(hand)
During any particularly time-consuming operation, your program should check for abort events periodically to allow the user to interrupt the operation from the keyboard.

Network events are handled by the Desk Manager as long as there's a desk accessory present that can respond to them. If GetNextEvent returns a TRUE result for a network event, then no such desk accessory is present; your program should normally just ignore the event.
*** The exact meaning and use of I/O driver events is not yet specified, so (for the time being) you needn't worry about how to respond to them. ***

If you're using your own event types for internal communication between parts of your program, you can use PostEvent to post them into the event queue. When you receive them back from GetNextEvent, you can respond to them in whatever way is appropriate for your application.

To "peek" at pending events without removing them from the event queue, use EventAvail instead of GetNextEvent. To remove all events of a given type or types from the queue, use FlushEvents. To control which event types get posted into the queue, or to cause certain types to be ignored, use SetEventMask.

In addition to receiving the user's mouse and keyboard actions in the form of events, you can directly read the keyboard (and keypad), mouse location, and state of the mouse button by calling GetKeys, GetMouse, and Button, respectively. To follow the mouse when the user drags it with the button down, use StillDown or WaitMouseUp.

Finally, you can read the current setting of the system clock at any time by calling TickCount.

EVENT MANAGER ROUTINES

This section describes all the Event Manager procedures and functions. They are presented in their Pascal form; for information on using them from assembly language, see "Using the Toolbox from Assembly Language" *** (doesn't exist, but see QuickDraw manual) *** and also "Notes for Assembly-Language Programmers" in this manual.

Accessing Events

```
FUNCTION GetNextEvent (eventMask: INTEGER; VAR theEvent: EventRecord) :
            BOOLEAN;
```

GetNextEvent returns the next available event of a specified type or types and removes it from the event queue. The event is returned as the value of the parameter theEvent; eventMask specifies which event types are of interest. GetNextEvent will return the next available event of any type designated by a 1 bit in the mask, subject to the priority rules discussed above under "Priority of Events". Event types corresponding to Ø bits in the mask are ignored. If no event of any of the designated types is available, GetNextEvent returns a null event, regardless of the setting of the eventMask bit for null events.

(eye)

> Since update events are never actually placed in the event queue, GetNextEvent can't remove them from the queue before returning them, as it does with other events. If your program doesn't take some explicit action to "clear" the update event, it will keep getting the same event back again. The normal way of clearing an update event is with BeginUpdate and EndUpdate; further explanation can be found in the Window Manager manual.

Before reporting an event to your program, GetNextEvent first calls the Desk Manager function 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 function result of FALSE to notify your program that it can ignore this event; a function result of TRUE means that your program should handle the event itself. The Desk Manager normally intercepts the following events:

- All activate and update events directed to a system window
- All keyboard and mouse up events if the currently active window is a system window and contains a desk accessory that is prepared to handle the event
- All network events if there is a desk accessory present that can handle them

The Desk Manager also responds to disk inserted events by attempting to mount the volume that has just been inserted; but in this case GetNextEvent returns TRUE to allow your program to take some further action if appropriate. All other events (including all mouse down events, regardless of which window is active) are left for your program to handle. See the Desk Manager manual for further details.

```
FUNCTION EventAvail (eventMask: INTEGER; VAR theEvent: EventRecord) :
            BOOLEAN;
```

EventAvail returns in theEvent the next available event of the type or types specified by eventMask, but does not remove the event from the event queue. This allows you to "peek" at pending events while still leaving them in the queue for later processing. In all other respects, EventAvail works exactly the same as GetNextEvent (see above).

Posting and Removing Events

```
PROCEDURE PostEvent (eventCode: INTEGER; eventMsg: LongInt);
```

PostEvent places in the event queue an event of the type designated by eventCode, with the event message specified by eventMsg. The main use of this procedure is for posting events of your own application-defined types. It's also sometimes useful for placing an event back in the queue after you've removed it with GetNextEvent. Notice, however, that in this case the system clock time, mouse location, and state of the mouse button and modifier keys will be changed from their original values to those in effect at the time the event is reposted.

(eye)

Be very careful about posting any but your own application-defined events into the queue. For example, attempting to post an activate or update event will

interfere with the internal operation of the Event Manager, since such events are detected in other ways and are not normally placed in the queue at all. If you repost a mouse event, the mouse location associated with it will be changed, possibly altering its meaning; reposting a keyboard event may cause modifier information to be lost or characters to be transposed from the order in which the user originally typed them. In general, you should avoid using PostEvent for any but your own events unless you're sure you know what you're doing.

```
PROCEDURE FlushEvents (eventMask,stopMask: INTEGER);
```

FlushEvents removes from the event queue all events of the type(s) specified by eventMask, up to, but not including, the first event of any type specified by stopMask. To remove all events of a particular type or types, use a stopMask value of Ø. You might use FlushEvents, for example, on receiving an abort event, to remove any mouse or keyboard events that may have occurred before the program was interrupted.

(hand)
When your program is first started up, it's usually a good idea to call FlushEvents(everyEvent,Ø) to empty the event queue of any stray events that may have been left lying around, such as unprocessed keystrokes typed to the Finder.

Reading the Mouse

```
PROCEDURE GetMouse (VAR mouseLoc: Point);
```

GetMouse returns the current mouse location as the value of the parameter mouseLoc. The location is expressed in the local coordinate system of the current grafPort (which might be, for example, the currently active window). Notice that this differs from the mouse location stored in the where field of an event record, which is given in global coordinates.

```
FUNCTION Button : BOOLEAN;
```

The Button function returns the current state of the mouse button: TRUE if the button is down, FALSE if it isn't.

```
FUNCTION StillDown : BOOLEAN;
```

Called after a mouse down event, StillDown tests whether the mouse button is still down. It returns TRUE if the button is currently down

and there are no more mouse events (mouse ups or later mouse downs) pending in the event queue. This is a true test of whether the button is still down from the original press--unlike Button (see above), which returns TRUE whenever the button is currently down, even if it has been released and pressed again since the original mouse down event.

```
FUNCTION WaitMouseUp : BOOLEAN;
```

WaitMouseUp works exactly the same as StillDown (see above), except that if the button is not still down from the original press, WaitMouseUp removes the corresponding mouse up event before returning FALSE.

Reading the Keyboard and Keypad

```
PROCEDURE GetKeys (VAR theKeys: KeyMap);
```

GetKeys reads the current state of the keyboard (and keypad, if any) and returns it in the form of a keyMap:

```
TYPE KeyMap = PACKED ARRAY [1..128] OF BOOLEAN;
```

Each element of the keyMap is TRUE if the corresponding key is down, FALSE if it isn't. The correspondence between elements of the keyMap and keys on the keyboard and keypad is shown in Table 1. KeyMap elements corresponding to blank entries in the table are unused. Notice that GetKeys doesn't distinguish between the two Shift keys or the two Option keys.

Element	Key	Element	Key
Ø	A	48	Tab
1	S	49	Space bar
2	D	5Ø	`
3	F	51	Backspace
4	H	52	Enter
5	G	53	
6	Z	54	
7	X	55	Command *** (name may change) ***
8	C	56	Shift
9	V	57	Caps Lock
1Ø		58	Option
11	B	59	
12	Q	6Ø	
13	W	61	
14	E	62	
15	R	63	
16	Y	64	
17	T	65	. (keypad)
18	1	66	* (keypad)
19	2	67	
2Ø	3	68	
21	4	69	
22	6	7Ø	+ (keypad)
23	5	71	Clear (keypad)
24	=	72	, (keypad)
25	9	73	
26	7	74	
27	-	75	
28	8	76	Enter (keypad)
29	Ø	77	/ (keypad)
3Ø	]	78	- (keypad)
31	O	79	
32	U	8Ø	
33	[	81	
34	I	82	Ø (keypad)
35	P	83	1 (keypad)
36	Return	84	2 (keypad)
37	L	85	3 (keypad)
38	J	86	4 (keypad)
39	'	87	5 (keypad)
4Ø	K	88	6 (keypad)
41	;	89	7 (keypad)
42	\	9Ø	
43	,	91	8 (keypad)
44	/	92	9 (keypad)
45	N	93	
46	M	94	
47	.	95	
		96-127	(Unused)

Table 1. KeyMap Elements

Miscellaneous Utilities

PROCEDURE SetEventMask (theMask: INTEGER);

SetEventMask sets the system event mask to the specified value. This mask controls the posting of events into the event queue. Only event types corresponding to 1 bits in the mask are posted; all others are ignored. The initial setting for the system event mask is to post all except key up events.

SetEventMask is useful if for some reason you want to know when keys are released as well as when they're pressed, or if you know that some other event type is of no interest to your program and needn't be posted. For example, if your program attaches no special meaning to mouse up events, you may want to dispense with them; or you might want to eliminate keyboard repeat by preventing auto-key events from being posted.

(hand)

> Since space in the event queue is limited, it's generally a good idea to disable any event type that you know your program has no use for.

The system event mask has no effect on activate or update events, since these events are detected in other ways and are never actually posted into the event queue.

FUNCTION TickCount : LongInt;

TickCount returns the current value of the system clock, which gives the elapsed time in ticks (sixtieths of a second) since the system was last started up.

JOURNALING

Using the Event Manager's journaling mechanism, all of a program's interactions with the Event Manager can be recorded and later played back, just as if they were happening for the first time. A journal is a record of all calls to the Event Manager routines GetNextEvent, EventAvail, GetMouse, Button, GetKeys, and TickCount. When a journal is being recorded, every call to any of these routines is sent to a special input/output driver and recorded in the journal, along with the result returned.

When the journal is played back, the same Event Manager calls read their results back from the journal instead of directly from the mouse, keyboard, keypad, and system clock. To the application program, the results it receives from the Event Manager in response to these calls

look exactly as if they were coming directly from the user. Since the program is event-driven, its behavior is completely determined by this stream of results. In particular, the sequence of calls the program issues to the Event Manager while replaying the journal will exactly match those that occurred when the journal was originally recorded. Since the results the Event Manager sends back are taken from the journal, the same sequence of events that occurred when the journal was recorded will be reproduced when the journal is played back.

(eye)

> Null events are not fully recorded in the journal: the fact that a null event was generated is recorded, but not the contents of the event record's fields. When the journal is played back, this information--the time the event was posted, the mouse location, and the state of the mouse button and modifier keys--is lost; the contents of the when, where, and modifiers fields are meaningless. If there's any chance your program may be executed from a journal instead of by direct interaction with the user, it should not rely in any way on the contents of a null event's fields.

The user can control journal recording and playback with the journaling desk accessory. It can also be controlled by the application program itself, but only from the assembly-language level: see "Notes for Assembly-Language Programmers", below, for details. *** The exact method of controlling the journaling mechanism has not been finally determined and will probably change. ***

RESOURCE FORMAT FOR KEYBOARD CONFIGURATIONS

The keyboard configuration, which translates the keys the user presses on the keyboard and keypad into the characters they represent, is treated as a resource and read from a resource file. The standard Macintosh keyboard configuration is stored in the system resource file and is read automatically when the Macintosh is started up. One way to substitute an alternate keyboard configuration--for example, for foreign use--is to use the Resource Editor *** (which doesn't yet exist) *** to replace the standard configuration with the new one in the system resource file. Then the next time the Macintosh is restarted, it will read the new keyboard configuration instead of the standard one.

(hand)

> It's also possible for a running program to install a new keyboard configuration "on the fly". This can only be done in assembly language; details are given in the next section.

Actually, the keyboard configuration is a pair of machine-language configuration routines, one for the keyboard and one for the keypad. These routines accept a key code, along with the state of the modifier

keys, as input and return the corresponding character code as output. The arguments and result are passed directly in machine registers, so the routines must be written in assembly language, not in Pascal.

The keyMap index (see Table 1) for the key to be translated is passed to the configuration routine in register D2. Register D1 contains the fourth word (indices 48 to 63) of the current keyMap, which includes the status bits for the four modifier keys at the positions shown in Figure 4. All other bits in this word should be ignored. The configuration routine is expected to return a character code in register D0; it should preserve the contents of all other registers. If the specified key combination doesn't correspond to any character, the configuration routine should return 0: in this case, no keyboard event will be generated.

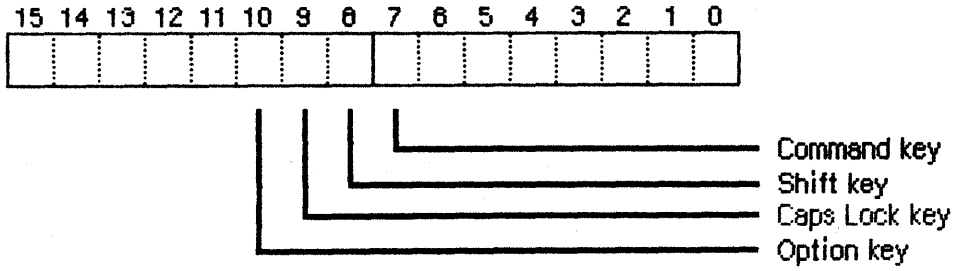


Figure 4. Modifier Bits for Configuration Routines

When the Macintosh is started up, two configuration routines are read from the system resource file. Both have a resource type of 'KEYC'; the resource ID is 1 for the keyboard routine and 2 for the keypad routine. The resource data for a resource of this type is just the machine code for the routine. The first byte of code is assumed to be the entry point for executing the routine.

NOTES FOR ASSEMBLY-LANGUAGE PROGRAMMERS

Information about how to use the User Interface Toolbox from assembly language is given elsewhere. *** For now, see the QuickDraw manual. *** This section contains special notes of interest to programmers who will be using the Event Manager from assembly language.

The primary aid to assembly-language programmers is a file named TOOLEQU.TEXT. If you use .INCLUDE to include this file when you assemble your program, all the Event Manager constants, offsets to locations of global variables, and offsets into fields of structured types will be available in symbolic form.

In assembly language, you can control the operation of the journaling mechanism by setting the global variable JournalFlag. Setting this variable to a positive, nonzero value turns on journal recording; setting it negative turns on playback; setting it to Ø turns journaling off.

The global variables Key1Trans and Key2Trans are used to hold pointers to the keyboard and keypad configuration routines, respectively. You can replace either or both of these routines "on the fly" by the following steps:

1. Call the Resource Manager function GetResource (or GetNamedResource) to find the new configuration routine in its resource file, read it into memory, and get a handle to it.

2. Use the Operating System call RecoverHandle to convert the existing routine pointer from Key1Trans or Key2Trans into a handle.

3. Use the Operating System call DisposHandle to free the storage occupied by the old routine.

4. Convert the handle you received from the Resource Manager into a pointer and store it in Key1Trans (for a keyboard routine) or Key2Trans (for a keypad routine).

APPENDIX: STANDARD KEY AND CHARACTER CODES

The following tables show the key and character codes used by Macintosh and the characters assigned to keys on the keyboard and keypad under the standard Macintosh keyboard configuration. All key and character codes are given in hexadecimal; for the benefit of readers with only ten fingers, there's a hexadecimal/decimal conversion table at the end of this Appendix.

Table 2 shows the extended ASCII character set used by Macintosh and Lisa. The first digit of the hexadecimal character code is shown at the top of the table, the second down the left side. For example, character code $47 stands for the capital letter G, which appears in the table at the intersection of column 4 and row 7.

Character codes between $2Ø and $7E have their normal ASCII meanings. Codes between $8Ø and $CA denote special characters included in the extended character set for business, scientific, and international use;

codes from $CB to $FF are unassigned. ASCII control characters ($ØØ to $1F, as well as $2Ø and $7F) are identified in the table by their traditional ASCII abbreviations:

Code	Abbr.	Meaning	Code	Abbr.	Meaning
$ØØ	NUL	Null	$1Ø	DLE	Data Link Escape
$Ø1	SOH	Start of Header	$11	DC1	Device Control 1
$Ø2	STX	Start of Text	$12	DC2	Device Control 2
$Ø3	ETX	End of Text	$13	DC3	Device Control 3
$Ø4	EOT	End of Tape	$14	DC4	Device Control 4
$Ø5	ENQ	Enquiry	$15	NAK	Negative Acknowledge
$Ø6	ACK	Acknowledge	$16	SYN	Synchronous Idle
$Ø7	BEL	Bell	$17	ETB	End Transmission Block
$Ø8	BS	Backspace	$18	CAN	Cancel
$Ø9	HT	Horizontal Tab	$19	EM	End of Medium
$ØA	LF	Line Feed	$1A	SUB	Substitute
$ØB	VT	Vertical Tab	$1B	ESC	Escape
$ØC	FF	Form Feed	$1C	FS	Field Separator
$ØD	CR	Carriage Return	$1D	GS	Group Separator
$ØE	SO	Shift Out	$1E	RS	Record Separator
$ØF	SI	Shift In	$1F	US	Unit Separator
$2Ø	SP	Space	$7F	DEL	Delete

However, most of these characters have no special meaning on Macintosh and cannot be generated from the Macintosh keyboard under the standard keyboard configuration. The exceptions are the following:

Code	Character	Key
$Ø3	ETX	Enter (keyboard and keypad)
$Ø8	BS	Backspace
$Ø9	HT	Tab
$ØD	CR	Return
$1B	ESC	Clear (keypad)
$1C	FS	Left arrow (keypad)
$1D	GS	Right arrow (keypad)
$1E	RS	Up arrow (keypad)
$1F	US	Down arrow (keypad)
$2Ø	SP	Space bar

In addition, as shown in the table, codes from $11 to $15 denote special characters used on the Macintosh screen, such as the open and solid Apple characters. These characters are intended exclusively for use on the screen, and have no keyboard or keypad equivalents under the standard keyboard configuration.

The characters shaded in the table are accented letters used in various foreign languages. Under the standard keyboard configuration, these characters cannot be typed directly from the keyboard. Instead, they are generated by first typing the accent or diacritical mark alone, followed by the letter to be accented. For example, a lowercase letter e with a grave accent (è, character code $8F) is produced by typing a grave accent (`, code $6Ø) followed by a lowercase e (code $65). The Macintosh keyboard driver will *** (eventually) *** translate such two-

character sequences involving diacriticals into the corresponding single accented letters.

Tables 3 and 4 show the hexadecimal key codes corresponding to keys on the Macintosh keyboard and keypad, respectively. Modifier keys are not shown, since they never generate keyboard events of their own.

Table 5 shows the hexadecimal character codes generated by each key on the keyboard under the standard keyboard configuration. Table 5a gives the character generated when the key is pressed by itself, Table 5b when it is pressed with the Shift key held down, Table 5c the Caps Lock key, Table 5d the Option key, and Table 5e the Option and Shift or Option and Caps Lock keys. Again, the modifier keys themselves are not shown.

Table 6 shows the hexadecimal character codes for the keypad under the standard keyboard configuration. Table 6a gives the character generated when the key is pressed by itself, Table 6b when it is pressed with the Shift key held down.

Finally, Table 7 is a conversion table between hexadecimal and decimal. To convert a two-digit hexadecimal number to decimal, find its first digit at the top of the table and its second down the left side. The decimal equivalent is found at the intersection of that column and row. For example, hexadecimal $6C is equivalent to decimal 1Ø8, found at the intersection of column 6 and row C. To convert a decimal number to hexadecimal, find the number in the body of the table and read its first and second hexadecimal digits from the head of that column and row, respectively. For example, decimal 227 is in column E and row 3, so its hexadecimal equivalent is $E3.

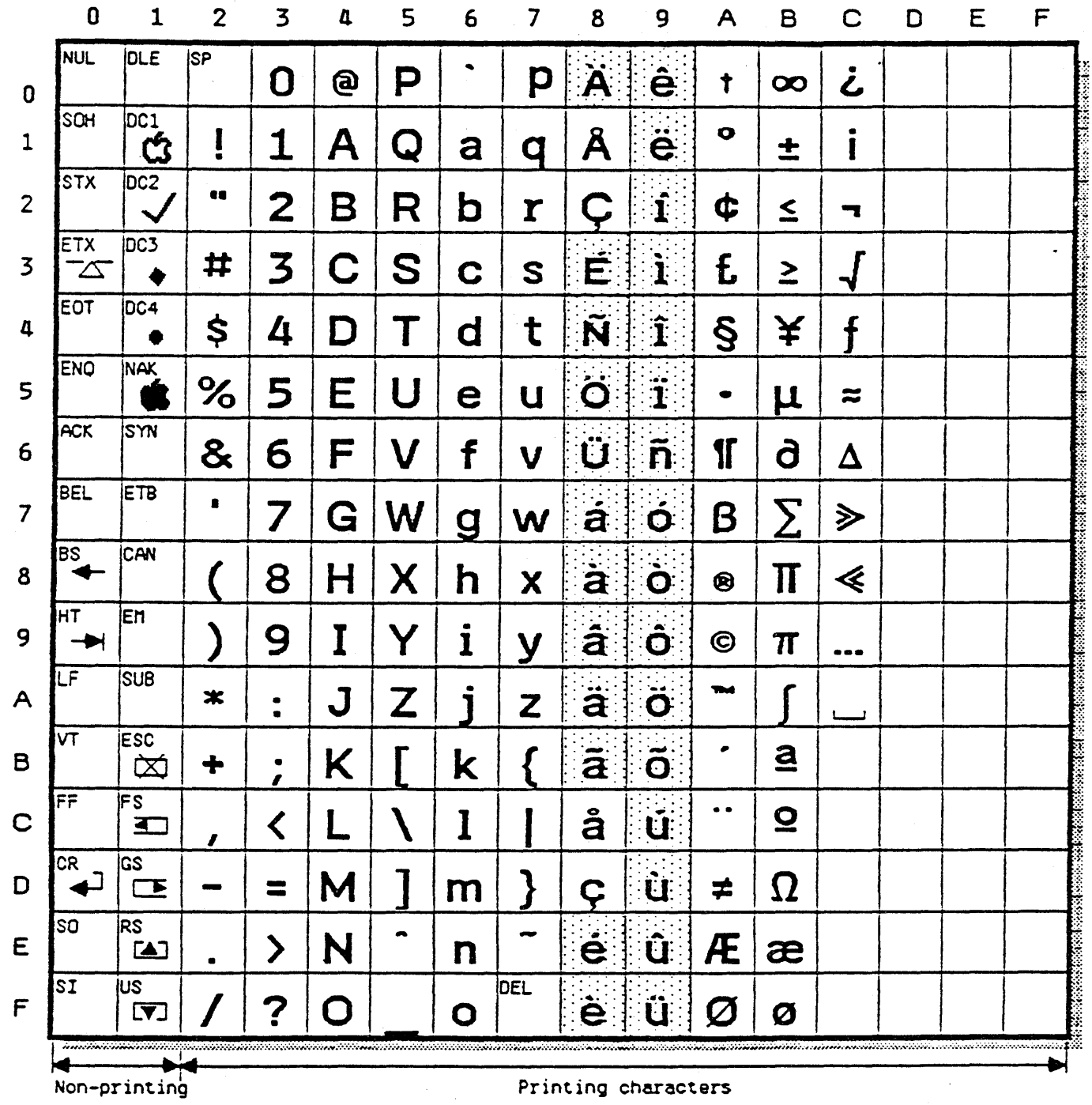


Table 2. Macintosh and Lisa Extended ASCII Character Set

Key:	[`]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[Ø]	[-]	[=]	[Backspace]
Code:	$32	$12	$13	$14	$15	$17	$16	$1A	$1C	$19	$1D	$1B	$18	$33

Key:	[Tab]	[Q]	[W]	[E]	[R]	[T]	[Y]	[U]	[I]	[O]	[P]	[[]	[]]	[\]
Code:	$3Ø	$ØC	$ØD	$ØE	$ØF	$11	$1Ø	$2Ø	$22	$1F	$23	$21	$1E	$2A

Key:	[A]	[S]	[D]	[F]	[G]	[H]	[J]	[K]	[L]	[;]	[´]	[Return]
Code:	$ØØ	$Ø1	$Ø2	$Ø3	$Ø5	$Ø4	$26	$28	$25	$29	$27	$24

Key:	[Z]	[X]	[C]	[V]	[B]	[N]	[M]	[,]	[.]	[/]
Code:	$Ø6	$Ø7	$Ø8	$Ø9	$ØB	$2D	$2E	$2B	$2F	$2C

Key:	[Space]	[Enter]
Code:	$31	$34

Table 3. Key Codes for Macintosh Keyboard

Key:	[Clear]	[-]	[+]	[*]
Code:	$47	$4E	$46	$42

Key:	[7]	[8]	[9]	[/]
Code:	$59	$5B	$5C	$4D

Key:	[4]	[5]	[6]	[,]
Code:	$56	$57	$58	$48

Key:	[1]	[2]	[3]	[E]
Code:	$53	$54	$55	[n]
				[t]
				[e]

Key:	[Ø]	[.]	[r]
Code:	$52	$41	$4C

Table 4. Key Codes for Macintosh Keypad

Key:	[`]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[Ø]	[-]	[=]	[Backspace]
Code:	$6Ø	$31	$32	$33	$34	$35	$36	$37	$38	$39	$3Ø	$2D	$3D	$Ø8
Char:	`	1	2	3	4	5	6	7	8	9	Ø	-	=	BS

Key:	[Tab]	[Q]	[W]	[E]	[R]	[T]	[Y]	[U]	[I]	[O]	[P]	[[]	[]]	[\]
Code:	$Ø9	$71	$77	$65	$72	$74	$79	$75	$69	$6F	$7Ø	$5B	$5D	$5C
Char:	HT	q	w	e	r	t	y	u	i	o	p	[	]	\

Key:	[A]	[S]	[D]	[F]	[G]	[H]	[J]	[K]	[L]	[;]	[´]	[Return]
Code:	$61	$73	$64	$66	$67	$68	$6A	$6B	$6C	$3B	$27	$ØD
Char:	a	s	d	f	g	h	j	k	l	;	´	CR

Key:	[Z]	[X]	[C]	[V]	[B]	[N]	[M]	[,]	[.]	[/]
Code:	$7A	$78	$63	$76	$62	$6E	$6D	$2C	$2E	$2F
Char:	z	x	c	v	b	n	m	,	.	/

Key:	[	Space	]	[Enter]
Code:		$2Ø		$Ø3
Char:		SP		ETX

(a) Unshifted

Table 5. Standard Character Codes for Macintosh Keyboard

Key:	[`]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[Ø]	[-]	[=]	[Backspace]
Code:	$7E	$21	$4Ø	$23	$24	$25	$5E	$26	$2A	$28	$29	$5F	$2B	$Ø8
Char:	~	!	@	#	$	%	^	&	*	(	)	_	+	BS

Key:	[Tab]	[Q]	[W]	[E]	[R]	[T]	[Y]	[U]	[I]	[O]	[P]	[[]	[]]	[\]
Code:	$Ø9	$51	$57	$45	$52	$54	$59	$55	$49	$4F	$5Ø	$7B	$7D	$7C
Char:	HT	Q	W	E	R	T	Y	U	I	O	P	{	}	|

Key:	[A]	[S]	[D]	[F]	[G]	[H]	[J]	[K]	[L]	[;]	[']	[Return]
Code:	$41	$53	$44	$46	$47	$48	$4A	$4B	$4C	$3A	$22	$ØD
Char:	A	S	D	F	G	H	J	K	L	:	"	CR

Key:	[Z]	[X]	[C]	[V]	[B]	[N]	[M]	[,]	[.]	[/]
Code:	$5A	$58	$43	$56	$42	$4E	$4D	$3C	$3E	$3F
Char:	Z	X	C	V	B	N	M	<	>	?

Key:	[	Space	]	[Enter]
Code:		$2Ø		$Ø3
Char:		SP		ETX

(b) Shift Key Down

Key:	[`]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[Ø]	[-]	[=]	[Backspace]
Code:	$6Ø	$31	$32	$33	$34	$35	$36	$37	$38	$39	$3Ø	$2D	$3D	$Ø8
Char:	`	1	2	3	4	5	6	7	8	9	Ø	-	=	BS

Key:	[Tab]	[Q]	[W]	[E]	[R]	[T]	[Y]	[U]	[I]	[O]	[P]	[[]	[]]	[\]
Code:	$Ø9	$51	$57	$45	$52	$54	$59	$55	$49	$4F	$5Ø	$5B	$5D	$5C
Char:	HT	Q	W	E	R	T	Y	U	I	O	P	[	]	\

Key:	[A]	[S]	[D]	[F]	[G]	[H]	[J]	[K]	[L]	[;]	[']	[Return]
Code:	$41	$53	$44	$46	$47	$48	$4A	$4B	$4C	$3B	$27	$ØD
Char:	A	S	D	F	G	H	J	K	L	;	´	CR

Key:	[Z]	[X]	[C]	[V]	[B]	[N]	[M]	[,]	[.]	[/]
Code:	$5A	$58	$43	$56	$42	$4E	$4D	$2C	$2E	$2F
Char:	Z	X	C	V	B	N	M	,	.	/

Key:	[	Space	]	[Enter]
Code:		$2Ø		$Ø3
Char:		SP		ETX

(c) Caps Lock Key Down

Table 5. Standard Character Codes for Macintosh Keyboard (continued)

Key:	[`]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[Ø]	[-]	[=]	[Backspace]
Code:	$6Ø	$C1	$AA	$A3	$A2	$BØ	$A4	$A6	$A5	$BB	$BC	$B1	$AD	$Ø8
Char:	`	¡	™	£	¢	∞	§	¶	•	ª	º	±	≠	BS

Key:	[Tab]	[Q]	[W]	[E]	[R]	[T]	[Y]	[U]	[I]	[O]	[P]	[[]	[]]	[\]
Code:	$Ø9	$A1	$B7	$AB	$A8	$AØ	$B4	$AC	$ØØ	$BF	$B9	$B5	$C8	$ØØ
Char:	HT	œ	Σ	´	®	†	¥	¨		ø	π	μ	«	

Key:		[A]	[S]	[D]	[F]	[G]	[H]	[J]	[K]	[L]	[;]	[´]	[Return]
Code:		$8C	$A7	$B6	$C4	$A9	$5E	$C6	$ØØ	$C2	$BD	$BE	$ØD
Char:		å	ß	∂	ƒ	©	^	Δ		¬	Ω	æ	CR

Key:		[Z]	[X]	[C]	[V]	[B]	[N]	[M]	[,]	[.]	[/]
Code:		$ØØ	$C5	$8D	$C3	$BA	$7E	$ØØ	$B2	$B3	$CØ
Char:			≈	ç	√	∫	~		≤	≥	¿

Key:		[	Space	]	[Enter]
Code:			$2Ø		$Ø3
Char:			SP		ETX

(d) Option Key Down

Key:	[`]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[Ø]	[-]	[=]	[Backspace]
Code:	$6Ø	$C1	$AA	$A3	$A2	$BØ	$A4	$A6	$A5	$BB	$BC	$B1	$AD	$Ø8
Char:	`	¡	™	£	¢	∞	§	¶	•	ª	º	±	≠	BS

Key:	[Tab]	[Q]	[W]	[E]	[R]	[T]	[Y]	[U]	[I]	[O]	[P]	[[]	[]]	[\]
Code:	$Ø9	$A1	$B7	$AB	$A8	$AØ	$B4	$AC	$ØØ	$AF	$B8	$B5	$C7	$ØØ
Char:	HT	‚	Σ	´	®	†	¥	¨		ø	∏	μ	»	

Key:		[A]	[S]	[D]	[F]	[G]	[H]	[J]	[K]	[L]	[;]	[´]	[Return]
Code:		$81	$A7	$B6	$C4	$A9	$5E	$C6	$ØØ	$C2	$BD	$AE	$ØD
Char:		Å	ß	∂	ƒ	©	^	Δ		¬	Ω	Æ	CR

Key:		[Z]	[X]	[C]	[V]	[B]	[N]	[M]	[,]	[.]	[/]
Code:		$ØØ	$C5	$82	$C3	$BA	$7E	$ØØ	$B2	$B3	$CØ
Char:			≈	ç	√	∫	~		≤	≥	¿

Key:		[	Space	]	[Enter]
Code:			$2Ø		$Ø3
Char:			SP		ETX

(e) Option and Shift or Option and Caps Lock Keys Down

Table 5. Standard Character Codes for Macintosh Keyboard (continued)

Key:	[Clear]	[-]	[+]	[*]
Code:	$1B	$2D	$1C	$1D
Char:	ESC	-	←	→

Key:	[7]	[8]	[9]	[/]
Code:	$37	$38	$39	$1E
Char:	7	8	9	↑

Key:	[4]	[5]	[6]	[,]
Code:	$34	$35	$36	$1F
Char:	4	5	6	↓

Key:	[1]	[2]	[3]	[E]
Code:	$31	$32	$33	[n]
Char:	1	2	3	[t]
				[e]

Key:	[Ø]	[.]	[r]
Code:	$3Ø	$2E	$Ø3
Char:	Ø	.	ETX

(a) Unshifted

Key:	[Clear]	[-]	[+]	[*]
Code:	$1B	$2D	$2B	$2A
Char:	ESC	-	+	*

Key:	[7]	[8]	[9]	[/]
Code:	$37	$38	$39	$2F
Char:	7	8	9	/

Key:	[4]	[5]	[6]	[,]
Code:	$34	$35	$36	$2C
Char:	4	5	6	,

Key:	[1]	[2]	[3]	[E]
Code:	$31	$32	$33	[n]
Char:	1	2	3	[t]
				[e]

Key:	[Ø]	[.]	[r]
Code:	$3Ø	$2E	$Ø3
Char:	Ø	.	ETX

(b) Shift Key Down

Table 6. Standard Character Codes for Macintosh Keypad

Second digit	First digit															
.	Ø	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Ø	Ø	16	32	48	64	8Ø	96	112	128	144	16Ø	176	192	2Ø8	224	24Ø
1	1	17	33	49	65	81	97	113	129	145	161	177	193	2Ø9	225	241
2	2	18	34	5Ø	66	82	98	114	13Ø	146	162	178	194	21Ø	226	242
3	3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
4	4	2Ø	36	52	68	84	1ØØ	116	132	148	164	18Ø	196	212	228	244
5	5	21	37	53	69	85	1Ø1	117	133	149	165	181	197	213	229	245
6	6	22	38	54	7Ø	86	1Ø2	118	134	15Ø	166	182	198	214	23Ø	246
7	7	23	39	55	71	87	1Ø3	119	135	151	167	183	199	215	231	247
8	8	24	4Ø	56	72	88	1Ø4	12Ø	136	152	168	184	2ØØ	216	232	248
9	9	25	41	57	73	89	1Ø5	121	137	153	169	185	2Ø1	217	233	249
A	1Ø	26	42	58	74	9Ø	1Ø6	122	138	154	17Ø	186	2Ø2	218	234	25Ø
B	11	27	43	59	75	91	1Ø7	123	139	155	171	187	2Ø3	219	235	251
C	12	28	44	6Ø	76	92	1Ø8	124	14Ø	156	172	188	2Ø4	22Ø	236	252
D	13	29	45	61	77	93	1Ø9	125	141	157	173	189	2Ø5	221	237	253
E	14	3Ø	46	62	78	94	11Ø	126	142	158	174	19Ø	2Ø6	222	238	254
F	15	31	47	63	79	95	111	127	143	159	175	191	2Ø7	223	239	255

Table 7. Hexadecimal/Decimal Conversion Table

SUMMARY OF THE EVENT MANAGER

```
CONST nullEvent    = Ø;        {null}
      mouseDown    = 1;        {mouse down}
      mouseUp      = 2;        {mouse up}
      keyDown      = 3;        {key down}
      keyUp        = 4;        {key up}
      autoKey      = 5;        {auto-key}
      updateEvt    = 6;        {update}
      diskEvt      = 7;        {disk inserted}
      activateEvt  = 8;        {activate}
      abortEvt     = 9;        {abort}
      networkEvt   = 1Ø;       {network}
      driverEvt    = 11;       {I/O driver}
      applEvt      = 12;       {application-defined}
      app2Evt      = 13;       {application-defined}
      app3Evt      = 14;       {application-defined}
      app4Evt      = 15;       {application-defined}

      nullMask     = 1;        {null}
      mDownMask    = 2;        {mouse down}
      mUpMask      = 4;        {mouse up}
      keyDownMask  = 8;        {key down}
      keyUpMask    = 16;       {key up}
      autoKeyMask  = 32;       {auto-key}
      updateMask   = 64;       {update}
      diskMask     = 128;      {disk inserted}
      activMask    = 256;      {activate}
      abortMask    = 512;      {abort}
      networkMask  = 1Ø24;     {network}
      driverMask   = 2Ø48;     {I/O driver}
      applMask     = 4Ø96;     {application-defined}
      app2Mask     = 8192;     {application-defined}
      app3Mask     = 16384;    {application-defined}
      app4Mask     = -32768;   {application-defined}

      everyEvent = -1;

TYPE EventRecord = RECORD
                     what:      INTEGER;
                     message:   LongInt;
                     when:      LongInt;
                     where:     Point;
                     modifiers: INTEGER
                   END;

     KeyMap = PACKED ARRAY [1..128] OF BOOLEAN;
```

Accessing Events

```
FUNCTION GetNextEvent (eventMask: INTEGER; VAR theEvent: EventRecord) :
                       BOOLEAN;
FUNCTION EventAvail   (eventMask: INTEGER; VAR theEvent: EventRecord) :
                       BOOLEAN;
```

Posting and Removing Events

```
PROCEDURE PostEvent   (eventCode: INTEGER; eventMsg: LongInt);
PROCEDURE FlushEvents (eventMask,stopMask: INTEGER);
```

Reading the Mouse

```
PROCEDURE GetMouse      (VAR mouseLoc: Point);
FUNCTION  Button :      BOOLEAN;
FUNCTION  StillDown :   BOOLEAN;
FUNCTION  WaitMouseUp : BOOLEAN;
```

Reading the Keyboard and Keypad

```
PROCEDURE GetKeys (VAR theKeys: KeyMap);
```

Miscellaneous Utilities

```
PROCEDURE SetEventMask (theMask: INTEGER);
FUNCTION  TickCount :  LongInt;
```

GLOSSARY

abort event: An event generated when the user presses a special combination of keys *** (tentatively Command-.) ***, or when the Event Manager's journaling mechanism wants a program to prepare for replaying a journal.

activate event: An event generated by the Window Manager when a window changes from active to inactive or vice versa.

auto-key event: An event generated periodically when the user presses and holds down a key on the keyboard or keypad.

character code: An integer representing the character that a key or combination of keys on the keyboard or keypad stands for.

configuration routine: A machine-language routine that defines a particular keyboard configuration by translating a key code, together with the state of the modifier keys, into a corresponding character code.

disk inserted event: An event generated when the user inserts a disk in a disk drive.

event: A notification to an application program of some occurrence that the program must respond to.

event code: An integer representing a particular type of event.

event mask: A parameter passed to an Event Manager routine specifying which types of event the routine is to be applied to.

event message: A field of an event record containing information specific to the particular type of event.

event queue: The Event Manager's list of pending events waiting to be processed.

event record: The internal representation of an event, where the Event Manager stores all pertinent information about that event.

I/O driver event: An event generated by one of the Macintosh's input/output drivers. *** (Not yet implemented.) ***

journal: A record of all of a program's interactions with the Event Manager over a period of time, which can be played back in order to reproduce the original session.

keyboard configuration: A resource that defines a particular keyboard layout by associating a character code with each key or combination of keys on the keyboard or keypad.

keyboard event: An event generated when the user presses, releases, or holds down a key on the keyboard or keypad; any key down, key up, or auto-key event.

key code: An integer representing a key on the keyboard or keypad, without reference to the character that key stands for.

key down event: An event generated when the user presses a key on the keyboard or keypad.

key up event: An event generated when the user releases a key on the keyboard or keypad.

modifier key: A key (Shift, Caps Lock, Option, or Command) that generates no keyboard events of its own, but changes the meaning of those generated by other keys.

mouse down event: An event generated when the user presses the mouse button.

mouse up event: An event generated when the user releases the mouse button.

network event: An event generated by the Macintosh's network driver. *** (Not yet implemented.) ***

null event: An event returned by the Event Manager when it has no other events to report.

post: To place an event in the event queue for later processing.

system event mask: A global event mask that controls which types of event get posted into the event queue.

update event: An event generated by the Window Manager when a window's contents need to be redrawn.

MACINTOSH USER EDUCATION

The Window Manager: A Programmer's Guide /WMGR/WINDOW

See Also: Macintosh User Interface Guidelines
The Memory Manager: A Programmer's Guide
QuickDraw: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Control Manager: A Programmer's Guide
The Desk Manager: A Programmer's Guide
The Dialog Manager: A Programmer's Guide
Toolbox Utilities: A Programmer's Guide
Programming Macintosh Applications in Assembly Language

Modification History:			
	First Draft	Pam Stanton-Wyman	8/16/82
	Interim Release	Caroline Rose	9/3Ø/82
	Second Draft	Caroline Rose	1Ø/8/82
	Revised	Caroline Rose	11/2/82
	Third Draft (ROM 2.1)	Caroline Rose	3/1/83
	Fourth Draft (ROM 7)	Caroline Rose	8/25/83
	Fifth Draft	Caroline Rose & Brent Davis	5/3Ø/84

ABSTRACT

Windows play an important part in Macintosh applications, since all information presented by an application appears in windows. The Window Manager provides routines for creating and manipulating windows. This manual describes those routines along with related concepts and data types.

Summary of significant changes and additions since last draft:

- New window definition IDs have been added (page 8) and the diameters of curvature for an rDocProc type of window can now be varied (page 9).
- The discussion of how a window is drawn has been corrected and refined (page 15).
- Assembly-language notes were added where appropriate, and the summary was updated to include all assembly-language information.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Window Manager, a major component of the Macintosh User Interface Toolbox. *** Eventually it will become part of the comprehensive Inside Macintosh manual. *** The Window Manager allows you to create, manipulate, and dispose of windows in a way that's consistent with the Macintosh User Interface Guidelines.

Like all Toolbox documentation, this manual assumes you're familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's Memory Manager. You should also be familiar with the following:

- Resources, as discussed in the Resource Manager manual.

- The basic concepts and structures behind QuickDraw, particularly points, rectangles, regions, grafPorts, and pictures. You don't have to know the QuickDraw routines in order to use the Window Manager, though you'll be using QuickDraw to draw inside a window.

- The Toolbox Event Manager. Some Window Manager routines are called only in response to certain events.

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly-language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it.

The manual begins with an introduction to the Window Manager and what you can do with it. It then discusses some basic concepts about windows: the relationship between windows and grafPorts; the various regions of a window; and the relationship between windows and resources. Following this is a discussion of window records, where the Window Manager keeps all the information it needs about a window. There are also sections on what happens when a window is drawn and when a window becomes active or inactive.

Next, a section on using the Window Manager introduces its routines and tells how they fit into the flow of your application program. This is followed by detailed descriptions of all Window Manager procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that will not interest all readers: special information is provided for programmers who want to define their own windows, and the exact formats of the resources related to windows are described.

Finally, there's a summary of the Window Manager for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE WINDOW MANAGER

The Window Manager is a tool for dealing with windows on the Macintosh screen. The screen represents a working surface or desktop; graphic objects appear on the desktop and can be manipulated with the mouse. A window is an object on the desktop that presents information, such as a document or a message. Windows can be any size or shape, and there can be one or many of them, depending on the application.

Some types of windows are predefined. One of these is the standard document window, as illustrated in Figure 1. Every document window has a title bar containing a title that's centered and in the system font and system font size. In addition, a particular document window may or may not have a close box or a size box; you'll learn in this manual how to implement them. There may also be scroll bars along the bottom and/or right edge of a document window. Scroll bars are controls, and are supported by the Control Manager.

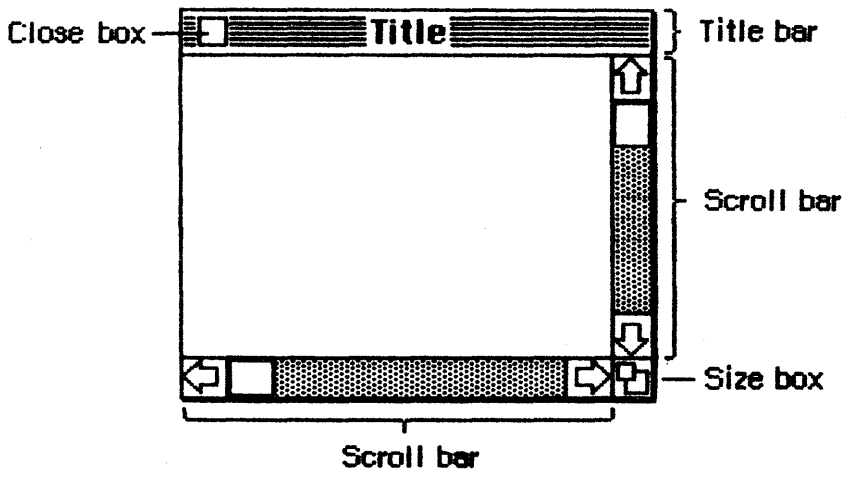


Figure 1. An Active Document Window

Your application can easily create standard types of windows such as document windows, and can also define its own types of windows. Some windows may be created indirectly for you when you use other parts of the Toolbox; an example is the window the Dialog Manager creates to display an alert box. Windows created either directly or indirectly by an application are collectively called application windows. There's also a class of windows called system windows; these are the windows in which desk accessories are displayed.

The document window shown in Figure 1 above is the frontmost (active) window, the one that will be acted on when the user types, gives commands, or whatever is appropriate to the application being used. Its title bar is highlighted--displayed in a distinctive visual way--so that the window will stand out from other, inactive windows that may be on the screen. Since a close box, size box, and scroll bars will have

an effect only in an active window, none of them appear in an inactive window (see Figure 2).

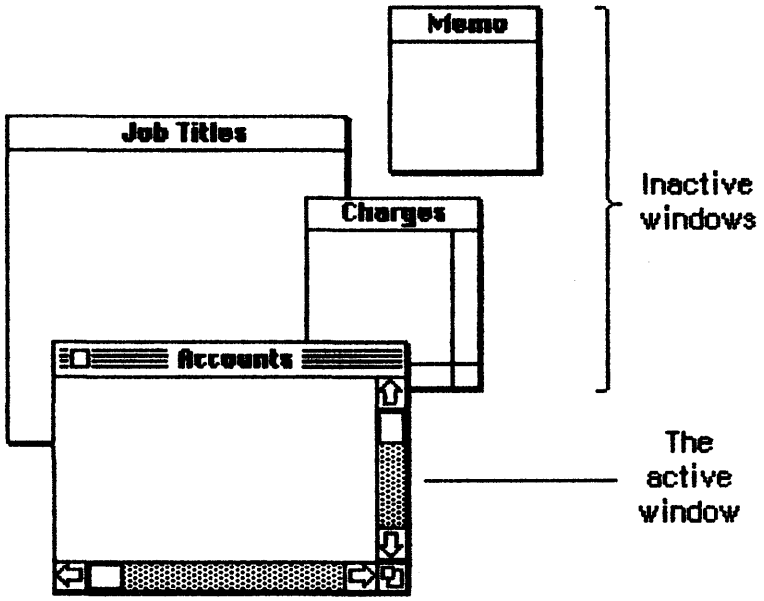


Figure 2. Overlapping Document Windows

(note)

If a document window has neither a size box nor scroll bars, the lines delimiting those areas aren't drawn, as in the Memo window in Figure 2.

An important function of the Window Manager is to keep track of overlapping windows. You can draw in any window without running over onto windows in front of it. You can move windows to different places on the screen, change their plane (their front-to-back ordering), or change their size, all without concern for how the various windows overlap. The Window Manager keeps track of any newly exposed areas and provides a convenient mechanism for you to ensure that they're properly redrawn.

Finally, you can easily set up your application so that mouse actions cause these standard responses inside a document window, or similar responses inside other windows:

- Clicking anywhere in an inactive window makes it the active window by bringing it to the front and highlighting its title bar.

- Clicking inside the close box of the active window closes the window. Depending on the application, this may mean that the window disappears altogether, or a representation of the window (such as an icon) may be left on the desktop.

- Dragging anywhere inside the title bar of a window (except in the close box, if any) pulls an outline of the window across the

screen, and releasing the mouse button moves the window to the new location. If the window isn't the active window, it becomes the active window unless the Command key was also held down. A window can never be moved completely off the screen; by convention, it can't be moved such that the visible area of the title bar is less than four pixels square.

- Dragging inside the size box of the active window changes the size of the window.

WINDOWS AND GRAFPORTS

It's easy for applications to use windows: to the application, a window is a grafPort that it can draw into like any other with QuickDraw routines. When you create a window, you specify a rectangle that becomes the portRect of the grafPort in which the window contents will be drawn. The bitMap for this grafPort, its pen pattern, and other characteristics are the same as the default values set by QuickDraw, except for the character font, which is set to the application font. These characteristics will apply whenever the application draws in the window, and they can easily be changed with QuickDraw routines (SetPort to make the grafPort the current port, and other routines as appropriate).

There is, however, more to a window than just the grafPort that the application draws in. In a standard document window, for example, the title bar and outline of the window are drawn by the Window Manager, not by the application. The part of a window that the Window Manager draws is called the window frame, since it usually surrounds the rest of the window. For drawing window frames, the Window Manager creates a grafPort that has the entire screen as its portRect; this grafPort is called the Window Manager port.

WINDOW REGIONS

Every window has the following two regions:

- the content region: the area that your application draws in
- the structure region: the entire window; its complete "structure" (the content region plus the window frame)

The content region is bounded by the rectangle you specify when you create the window (that is, the portRect of the window's grafPort); for a document window, it's the entire portRect. This is where your application presents information and where the size box and scroll bars of a document window are located. By convention, clicking in the content region of an inactive window makes it the active window.

(note)

The results of clicking and dragging that are discussed here don't happen automatically; you have to make the right Window Manager calls to cause them to happen.

A window may also have any of the regions listed below. Clicking or dragging in one of these regions causes the indicated action.

- A go-away region within the window frame. Clicking in this region of the active window closes the window.

- A drag region within the window frame. Dragging in this region pulls an outline of the window across the screen, moves the window to a new location, and makes it the active window unless the Command key was held down.

- A grow region, usually within the content region. Dragging in this region of the active window changes the size of the window. In a document window, the grow region is in the content region, but in windows of your own design it may be in either the content region or the window frame.

Figure 3 illustrates the various regions of a standard document window and its window frame.

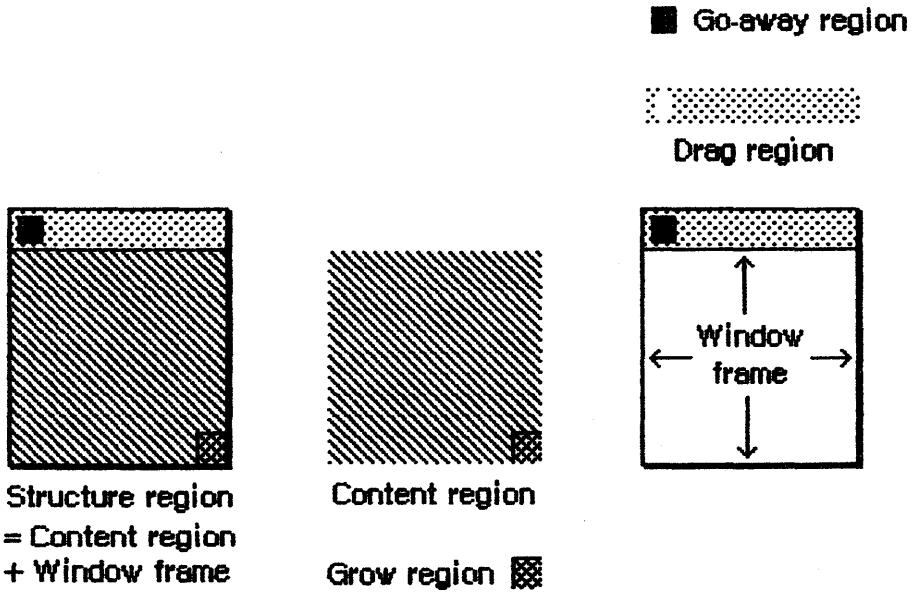


Figure 3. Document Window Regions and Frame

An example of a window that has no drag region is the window that displays an alert box. On the other hand, you could design a window whose drag region is the entire structure region and whose content region is empty; such a window might present a fixed picture rather than information that's to be manipulated.

Another important window region is the update region. Unlike the regions described above, the update region is dynamic rather than fixed: the Window Manager keeps track of all areas of the content

region that have to be redrawn and accumulates them into the update region. For example, if you bring to the front a window that was overlapped by another window, the Window Manager adds the formerly overlapped (now exposed) area of the front window's content region to its update region. You'll also accumulate areas into the update region yourself; the Window Manager provides update region maintenance routines for this purpose.

WINDOWS AND RESOURCES

The general appearance and behavior of a window is determined by a routine called its window definition function, which is stored as a resource in a resource file. The window definition function performs all actions that differ from one window type to another, such as drawing the window frame. The Window Manager calls the window definition function whenever it needs to perform one of these type-dependent actions (passing it a message that tells which action to perform).

The system resource file includes window definition functions for the standard document window and other predefined types of windows. If you want to define your own, nonstandard window types, you'll have to write your own window definition functions for them, as described later in the section "Defining Your Own Windows".

When you create a window, you specify its type with a window definition ID, which tells the Window Manager the resource ID of the definition function for that type of window. You can use one of the following constants as a window definition ID to refer to a predefined type of window (see Figure 4):

```
CONST documentProc  = Ø;  {standard document window}
      dBoxProc      = 1;  {alert box or modal dialog box}
      plainDBox     = 2;  {plain box}
      altDBoxProc   = 3;  {plain box with shadow}
      noGrowDocProc = 4;  {document window without size box}
      rDocProc      = 16; {rounded-corner window}
```

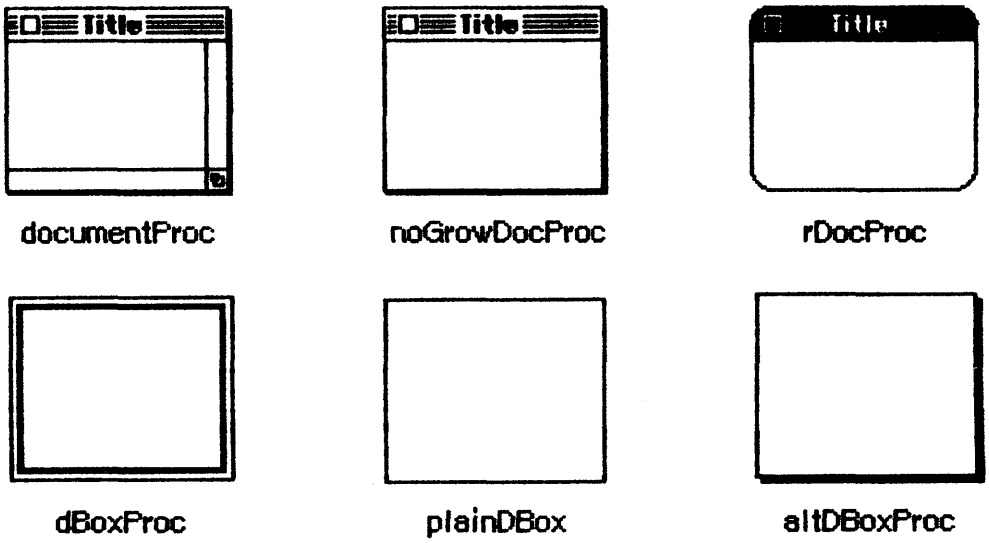


Figure 4. Predefined Types of Windows

DocumentProc represents a standard document window that may or may not contain a size box; noGrowDocProc is exactly the same except that the window must **not** contain a size box. If you're working with a number of document windows that need to be treated similarly, but some will have size boxes and some won't, you can use documentProc for all of them. If none of the windows will have size boxes, however, it's more convenient to use noGrowDocProc.

The dBoxProc type of window resembles an alert box or a "modal" dialog box (the kind that requires the user to respond before doing any other work on the desktop). It's a rectangular window with no go-away region, drag region, or grow region and with a two-pixel-thick border two pixels in from the edge. It has no special highlighted state because alerts and modal dialogs are always displayed in the frontmost window. PlainDBox and altDBoxProc are variations of dBoxProc: plainDBox is just a plain box with no inner border, and altDBoxProc has a two-pixel-thick shadow instead of a border.

The rDocProc type of window is like a document window with no grow region, with rounded corners, and with a method of highlighting that inverts the entire title bar (that is, changes white to black and vice versa). It's sometimes used for desk accessories. Rounded-corner windows are drawn by the QuickDraw procedure FrameRoundRect, which requires that the diameters of curvature be passed as parameters. For an rDocProc type of window, the diameters of curvature are both 16. You can add a number from 1 to 7 to rDocProc to get different diameters:

Window definition ID	Diameters of curvature
rDocProc	16, 16
rDocProc + 1	4, 4
rDocProc + 2	6, 6
rDocProc + 3	8, 8
rDocProc + 4	1Ø, 1Ø
rDocProc + 5	12, 12
rDocProc + 6	2Ø, 2Ø
rDocProc + 7	24, 24

To create a window, the Window Manager needs to know not only the window definition ID but also other information specific to this window, such as its title (if any), its location, and its plane. You can supply all the needed information in individual parameters to a Window Manager routine or, better yet, you can store it as a single resource in a resource file and just pass the resource ID. This type of resource is called a window template. Using window templates simplifies the process of creating a number of windows of the same type. More important, it allows you to isolate specific window descriptions from your application's code. Translation of window titles into a foreign language, for example, would require only a change to the resource file.

(note)

You can create window templates and store them in resource files with the aid of the Resource Editor *** eventually (for now, the Resource Compiler) ***. The Resource Editor relieves you of having to know the exact format of a window template, but for interested programmers this information is given in the section "Formats of Resources for Windows".

WINDOW RECORDS

The Window Manager keeps all the information it requires for its operations on a particular window in a window record. The window record contains the following:

- The grafPort for the window.
- A handle to the window definition function.
- A handle to the window's title, if any.
- The window class, which tells whether the window is a system window, a dialog or alert window, or a window created directly by the application.
- A handle to the window's control list, which is a list of all the controls, if any, in the window. The Control Manager maintains this list.

- A pointer to the next window in the window list, which is a list of all windows ordered according to their front-to-back positions on the desktop.

*** The handle to the window's title has a data type that you may want to use yourself elsewhere; it's defined in the Memory Manager as follows:

```
TYPE Str255       = STRING[255];
     StringPtr    = ^Str255;
     StringHandle = ^StringPtr;
```

Forthcoming Memory Manager documentation will include this. ***

The window record also contains an indication of whether the window is currently visible or invisible. These terms refer only to whether the window is drawn in its plane, not necessarily whether you can see it on the screen. If, for example, it's completely overlapped by another window, it's still "visible" even though it can't be seen in its current location.

The 32-bit reference value field of the window record is reserved for use by your application. You specify an initial reference value when you create a window, and can then read or change the reference value whenever you wish. For example, it might be a handle to data associated with the window, such as a TextEdit edit record.

Finally, a window record may contain a handle to a QuickDraw picture of the window contents. The application can swap out the code and data that draw the window contents if desired, and instead use this picture. For more information, see "How a Window is Drawn".

The data type for a window record is called WindowRecord. A window record is referred to by a pointer, as discussed further under "Window Pointers" below. You can store into and access most of the fields of a window record with Window Manager routines, so normally you don't have to know the exact field names. Occasionally--particularly if you define your own type of window--you may need to know the exact structure; it's given below under "The WindowRecord Data Type".

Window Pointers

There are two types of pointer through which you can access windows: WindowPtr and WindowPeek. Most programmers will only need to use WindowPtr.

The Window Manager defines the following type of window pointer:

```
TYPE WindowPtr = GrafPtr;
```

It can do this because the first thing stored in a window record is the window's grafPort. This type of pointer can be used to access fields of the grafPort or can be passed to QuickDraw routines that expect

pointers to grafPorts as parameters. The application might call such routines to draw into the window, and the Window Manager itself calls them to perform many of its operations. The Window Manager gets the additional information it needs from the rest of the window record beyond the grafPort.

In some cases, however, a more direct way of accessing the window record may be necessary or desirable. For this reason, the Window Manager also defines the following type of window pointer:

```
TYPE WindowPeek = ^WindowRecord;
```

Programmers who want to access WindowRecord fields directly must use this type of pointer (which derives its name from the fact that it lets you "peek" at the additional information about the window). A WindowPeek pointer is also used wherever the Window Manager will not be calling QuickDraw routines and will benefit from a more direct means of getting to the data stored in the window record.

Assembly-language note: From assembly language, of course, there's no type checking on pointers, and the two types of pointer are equal.

The WindowRecord Data Type

For those who want to know more about the data structure of a window record or who will be defining their own types of windows, the exact data structure is given here.

```
TYPE WindowRecord =
            RECORD
              port:          GrafPort;    {window's grafPort}
              windowKind:    INTEGER;     {window class}
              visible:       BOOLEAN;     {TRUE if visible}
              hilited:       BOOLEAN;     {TRUE if highlighted}
              goAwayFlag:    BOOLEAN;     {TRUE if has go-away region}
              spareFlag:     BOOLEAN;     {reserved for future use}
              strucRgn:      RgnHandle;   {structure region}
              contRgn:       RgnHandle;   {content region}
              updateRgn:     RgnHandle;   {update region}
              windowDefProc: Handle;      {window definition function}
              dataHandle:    Handle;      {data used by windowDefProc}
              titleHandle:   StringHandle; {window's title}
              titleWidth:    INTEGER;     {width of title in pixels}
              controlList:   Handle;      {window's control list}
              nextWindow:    WindowPeek;  {next window in window list}
              windowPic:     PicHandle;   {picture for drawing window}
              refCon:        LongInt      {window's reference value}
            END;
```

The port is the window's grafPort.

WindowKind identifies the window class. If negative, it means the window is a system window (it's the desk accessory's reference number, as described in the Desk Manager manual). It may also be one of the following predefined constants:

```
CONST dialogKind = 2;  {dialog or alert window}
      userKind   = 8;  {window created directly by the application}
```

WindowKind values 1 through 7 are reserved for system use. UserKind is stored in this field when a window is created directly by application calls to the Window Manager (rather than indirectly through the Dialog Manager, as for dialogKind); for such windows the application can in fact set the window class to any value greater than 8 if desired.

When visible is TRUE, the window is currently visible.

Hilited and goAwayFlag are checked by the window definition function when it draws the window frame, to determine whether the window should be highlighted and whether it should have a go-away region. For a document window, this means that if hilited is TRUE, the title bar of the window is highlighted, and if goAwayFlag is also TRUE, a close box appears in the highlighted title bar.

(note)

The Window Manager sets the visible and hilited flags to TRUE by storing 255 in them rather than 1. This may cause problems in Lisa Pascal; to be safe, you should check for the truth or falsity of these flags by comparing ORD of the flag to Ø. For example, you would check to see if the flag is TRUE with ORD(myWindow.visible) <> Ø.

StrucRgn, contRgn, and updateRgn are region handles, as defined in QuickDraw, to the structure region, content region, and update region of the window. These regions are all in global coordinates.

WindowDefProc is a handle to the window definition function for this type of window. When you create a window, you identify its type with a window definition ID, which is converted into a handle and stored in the windowDefProc field. Thereafter, the Window Manager uses this handle to access the definition function; you should never need to access this field directly.

(note)

The high-order byte of the windowDefProc field contains some additional information that the Window Manager gets from the window definition ID; for details, see the section "Defining Your Own Windows". Also note that if you write your own window definition function, you can include it as part of your application's code and just store a handle to it in the windowDefProc field.

DataHandle is reserved for use by the window definition function. If the window is one of your own definition, your window definition function may use this field to store and access any desired information. If no more than four bytes of information are needed, the definition function can store the information directly in the dataHandle field rather than use a handle. For example, the definition function for rounded-corner windows uses this field to store the diameters of curvature.

TitleHandle is a stringHandle to the window's title, if any.

TitleWidth is the width, in pixels, of the window's title in the system font and system font size. This width is determined by the Window Manager and is normally of no concern to the application.

ControlList is a handle to the window's control list.

NextWindow is a pointer to the next window in the window list, that is, the window behind this window. If this window is the farthest back (with no windows between it and the desktop), nextWindow is NIL.

Assembly-language note: The global variable windowList contains a pointer to the first window in the window list. Remember that any window in the list may be invisible.

WindowPic is a handle to a QuickDraw picture of the window contents, or NIL if the application will draw the window contents in response to an update event, as described under "How a Window is Drawn", below.

RefCon is the window's reference value field, which the application may store into and access for any purpose.

(note)

Notice that the go-away, drag, and grow regions are not included in the window record. Although these are conceptually regions, they don't necessarily have the formal data structure for regions as defined in QuickDraw. The window definition function determines where these regions are, and it can do so with great flexibility.

Assembly-language note: The global constant windowSize equals the length in bytes of a window record.

HOW A WINDOW IS DRAWN

When a window is drawn or redrawn, the following two-step process usually takes place: the Window Manager draws the window frame and the application draws the window contents.

To perform the first step of this process, the Window Manager calls the window definition function with a request that the window frame be drawn. It manipulates regions of the Window Manager port as necessary before calling the window definition function, to ensure that only what should and must be drawn is actually drawn on the screen. Depending on a parameter passed to the routine that created the window, the window definition function may or may not draw a go-away region in the window frame (a close box in the title bar, for a document window).

Usually the second step is that the Window Manager generates an update event to get the application to draw the window contents. It does this by accumulating in the update region the areas of the window's content region that need updating. The Toolbox Event Manager periodically checks to see if there's any window whose update region is not empty; if it finds one, it reports (via the GetNextEvent function) that an update event has occurred, and passes along the window pointer in the event message. (If it finds more than one such window, it issues an update event for the frontmost one, so that update events are reported in front-to-back order.) The application should respond as follows:

1. Call BeginUpdate. This procedure temporarily replaces the visRgn of the window's grafPort with the intersection of the visRgn and the update region. It then sets the update region to the empty region; this "clears" the update event so it won't be reported again.

2. Draw the window contents, entirely or in part. Normally it's more convenient to draw the entire content region, but it suffices to draw only the visRgn. In either case, since the visRgn is limited to where it intersects the old update region, only the parts of the window that require updating will actually be drawn on the screen.

3. Call EndUpdate, which restores the normal visRgn.

Figure 5 illustrates the effect of BeginUpdate and EndUpdate on the visRgn and update region of a window that's redrawn after being brought to the front.

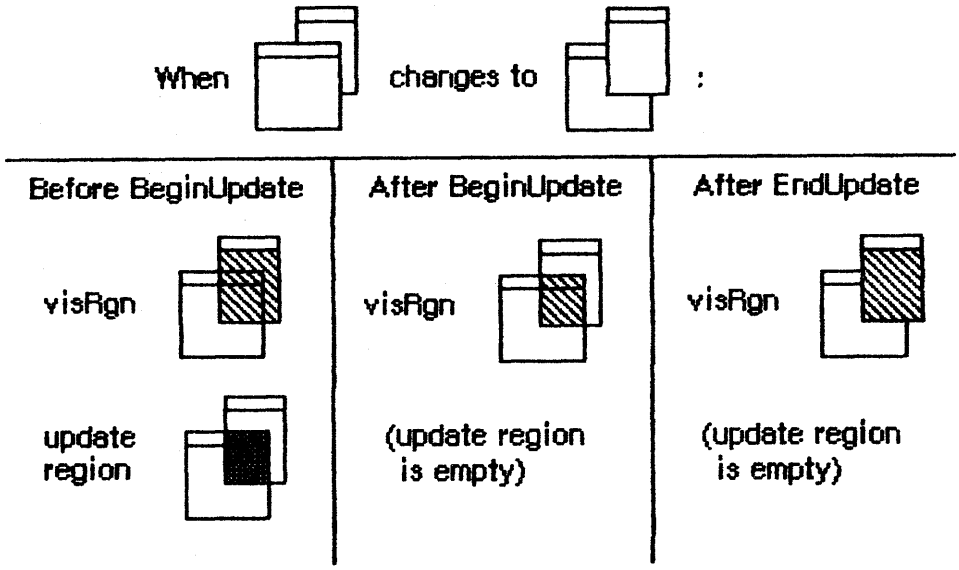


Figure 5. Updating Window Contents

If you choose to draw only the visRgn in step 2 above, there are various ways you can check to see whether what you need to draw falls in that region. With the QuickDraw functions PtInRgn and RectInRgn, you can check whether a point or rectangle lies in the visRgn. Or it may be more convenient to look at the visRgn's enclosing rectangle, which is stored in its bBox field. The QuickDraw functions PtInRect and SectRect let you check for intersection with a rectangle.

To be able to respond to update events for one of its windows, the application has to keep track of the window's contents, usually in a data structure. In most cases, it's best **never** to draw immediately into a window; when you need to draw something, just keep track of it and add the area where it should be drawn to the window's update region (by calling one of the Window Manager's update region maintenance routines, InvalRect and InvalRgn). Do the actual drawing only in response to an update event. Usually this will simplify the structure of your application considerably, but be aware of the following possible problems:

- This method isn't convenient to apply to areas that aren't easily defined by a rectangle or a region; in those cases, you would just draw directly into the window.

- If you find that sometimes there's too long a delay before the update event happens, you can "force" update events where necessary by calling GetNextEvent with a mask that accepts only that type of event.

The Window Manager allows an alternative to the update event mechanism that may be useful for some windows: a handle to a QuickDraw picture may be stored in the window record. If this is done, the Window Manager doesn't generate an update event to get the application to draw the window contents; instead, it calls the QuickDraw procedure

DrawPicture to draw the picture whose handle is stored in the window record (and it does all the necessary region manipulation). If the amount of storage occupied by the picture is less than the size of the code and data necessary to draw the window contents, and the application can swap out that code and data, this drawing method is more economical (and probably faster) than the usual updating process.

Assembly-language note: The global variables saveUpdate and paintWhite are flags that determine whether the Window Manager will generate any update events and whether it will paint the update region of a window white before generating an update event, respectively. Normally they're both set, but you can clear them to prevent the behavior that they control; for example, clearing paintWhite is useful if the background of the window isn't white. The Window Manager sets both flags periodically, so you should clear the appropriate flag just before each situation you wish it to apply to.

MAKING A WINDOW ACTIVE: ACTIVATE EVENTS

A number of Window Manager routines change the state of a window from inactive to active or from active to inactive. For each such change, the Window Manager generates an activate event, passing along the window pointer in the event message and, in the modifiers field of the event record, bits that indicate the following:

- Whether this window has become active or inactive. (If active, the activeFlag bit is set; if inactive, it's Ø.)

- Whether the active window is changing from an application window to a system window or vice versa. (If so, the changeFlag bit is set; otherwise, it's Ø.)

When the Toolbox Event Manager finds out from the Window Manager that an activate event has been generated, it passes the event on to the application (via the GetNextEvent function). Activate events have the highest priority of any type of event.

Usually when one window becomes active another becomes inactive, and vice versa, so activate events are most commonly generated in pairs. When this happens, the Window Manager generates first the event for the window becoming inactive, and then the event for the window becoming active. Sometimes only a single activate event is generated, such as when there's only one window in the window list, or when the active window is permanently disposed of (since it no longer exists).

Activate events for dialog and alert windows are handled by the Dialog Manager. In response to activate events for windows created directly

by your application, you might take actions such as the following:

- In a document window containing a size box or scroll bars, erase the size box icon or scroll bars when the window becomes inactive and redraw them when it becomes active.

- In a window that contains text being edited, remove the highlighting or blinking vertical bar from the text when the window becomes inactive and restore it when the window becomes active.

- Enable or disable a menu or certain menu items as appropriate to match what the user can do when the window becomes active or inactive.

Assembly-language note: The global variable curActivate contains a pointer to a window for which an activate event has been generated; the event, however, may not yet have been reported to the application via GetNextEvent, so you may be able to keep the event from happening by clearing curActivate. Similarly, you may be able to keep a deactivate event from happening by clearing the global variable curDeactive.

USING THE WINDOW MANAGER

This section discusses how the Window Manager routines fit into the general flow of an application program and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

To use the Window Manager, you must have previously called InitGraf to initialize QuickDraw and InitFonts to initialize the Font Manager. The first Window Manager routine to call is the initialization routine InitWindows, which draws the desktop and the (empty) menu bar.

Where appropriate in your program, use NewWindow or GetNewWindow to create any windows you need; these functions return a window pointer, which you can then use to refer to the window. NewWindow takes descriptive information about the window from its parameters, whereas GetNewWindow gets the information from window templates in a resource file. You can supply a pointer to the storage for the window record or let it be allocated by the routine creating the window; when you no longer need a window, call CloseWindow if you supplied the storage, or DisposeWindow if not.

When the Toolbox Event Manager function GetNextEvent reports that an update event has occurred, call BeginUpdate, draw the visRgn or the entire content region, and call EndUpdate (see "How a Window is Drawn",

above). You can also use InvalRect or InvalRgn to prepare a window for updating, and ValidRect or ValidRgn to temporarily protect portions of the window from updating.

When drawing the contents of a window that contains a size box in its content region, you'll draw the size box if the window is active or just the lines delimiting the size box and scroll bar areas if it's inactive. The FrontWindow function tells you which is the active window; the DrawGrowIcon procedure helps you draw the size box or delimiting lines. You'll also call the latter procedure when an activate event occurs that makes the window active or inactive.

(note)

Although unlikely, it's possible that a desk accessory may not be set up to handle update or activate events, so GetNextEvent may return TRUE for a system window's update or activate event. For this reason, it's a good idea to check whether such an event applies to one of your own windows rather than a system window, and ignore it if it.

When GetNextEvent reports a mouse-down event, call the FindWindow function to find out which part of which window the mouse button was pressed in.

- If it was pressed in the content region of an inactive window, make that window the active window by calling SelectWindow.

- If it was pressed in the grow region of the active window, call GrowWindow to pull around an image that shows the window's size will change, and then SizeWindow to actually change the size.

- If it pressed in the drag region of any window, call DragWindow, which will pull an outline of the window across the screen, move the window to a new location, and, if the window is inactive, make it the active window (unless the Command key was held down).

- If it was pressed in the go-away region of the active window, call TrackGoAway to handle the highlighting of the go-away region and to determine whether the mouse is inside the region when the button is released. Then do whatever is appropriate as a response to this mouse action in the particular application. For example, call CloseWindow or DisposeWindow if you want the window to go away permanently, or HideWindow if you want it to disappear temporarily.

(note)

If the mouse button was pressed in the content region of an active window (but not in the grow region), call the Control Manager function FindControl if the window contains controls. If it was pressed in a system window, call the Desk Manager procedure SystemClick. See the Control Manager and Desk Manager manuals for details.

The procedure that simply moves a window without pulling around an outline of it, MoveWindow, can be called at any time, as can SizeWindow --though the application should not surprise the user by taking these actions unexpectedly. There are also routines for changing the title of a window, placing a window behind another window, and making a window visible or invisible. Call these Window Manager routines wherever needed in your program.

WINDOW MANAGER ROUTINES

This section describes first the Window Manager procedures and functions that are used in most applications, and then the low-level routines for use by programmers who have their own ideas about what to do with windows. All routines are presented in their Pascal form; for information on using them from assembly language, see Programming Macintosh Applications in Assembly Language.

Initialization and Allocation

PROCEDURE InitWindows;

InitWindows initializes the Window Manager. It creates the Window Manager port; you can get a pointer to this port with the GetWMgrPort procedure. InitWindows draws the desktop and the (empty) menu bar. Call this procedure once before all other Window Manager routines.

(note)

InitWindows creates the Window Manager port as a nonrelocatable block in the application heap. For information on how this may affect your application's use of memory, see the Memory Manager manual. *** (A section on how to survive with limited memory will be added to that manual.) ***

Assembly-language note: InitWindows draws as the desktop the region whose handle is in the global variable grayRgn (normally a rounded-corner rectangle occupying the entire screen, minus the menu bar). It paints this region with the pattern in the global variable deskPattern (normally gray). Any subsequent time that the desktop needs to be drawn, such as when a new area of it is exposed after a window is closed or moved, the Window Manager calls the procedure whose pointer is stored in the global variable deskHook, if any (normally deskHook is Ø). The deskHook procedure is called with Ø in DØ to distinguish this use of it from its use in responding to clicks on the desktop (as discussed in the description of FindWindow); it should respond by painting thePort^.clipRgn with deskPattern and then

doing anything else it wants.

```
PROCEDURE GetWMgrPort (VAR wPort:  GrafPtr);
```

GetWMgrPort returns in wPort a pointer to the Window Manager port.

Assembly-language note: This pointer is stored in the global variable wMgrPort.

```
FUNCTION NewWindow (wStorage: Ptr; boundsRect: Rect; title: Str255;
            visible: BOOLEAN; procID: INTEGER; behind: WindowPtr;
            goAwayFlag: BOOLEAN; refCon: LongInt) : WindowPtr;
```

NewWindow creates a window as specified by its parameters, adds it to the window list, and returns a windowPtr to the new window. It allocates space for the structure and content regions of the window and asks the window definition function to calculate those regions.

WStorage is a pointer to where to store the window record. For example, if you've declared the variable wRecord of type WindowRecord, you can pass @wRecord as the first parameter to NewWindow. If you pass NIL for wStorage, the window record will be allocated on the heap; in that case, though, the record will be nonrelocatable, and so you risk ending up with a fragmented heap. You should therefore not pass NIL for wStorage unless your program has an unusually large amount of memory available or has been set up to dispose of windows dynamically. Even then, you should avoid passing NIL for wStorage if there's no limit to the number of windows that your application can open. *** (Some of this may be moved to the Memory Manager manual when that manual is updated to have a section on how to survive with limited memory.) ***

BoundsRect, a rectangle given in global coordinates, determines the window's size and location. It becomes the portRect of the window's grafPort; note, however, that the portRect is in local coordinates. NewWindow makes the QuickDraw call SetOrigin(Ø,Ø), so that the top left corner of the portRect will be (Ø,Ø).

(note)
The bitMap, pen pattern, and other characteristics of the window's grafPort are the same as the default values set by the OpenPort procedure in QuickDraw, except for the character font, which is set to the application font rather than the system font. Note, however, that the SetOrigin(Ø,Ø) call changes the coordinates of the grafPort's portBits.bounds and visRgn as well as its

portRect.

Title is the window's title. If the title of a document window is longer than will fit in the title bar, only as much of the beginning of the title as will fit is displayed.

If the visible parameter is TRUE, NewWindow draws the window. First it calls the window definition function to draw the window frame; if goAwayFlag is also TRUE and the window is frontmost (as specified by the behind parameter, below), it draws a go-away region in the frame. Then it generates an update event for the entire window contents.

ProcID is the window definition ID, which leads to the window definition function for this type of window. The window definition IDs for the predefined types of windows are listed above under "Windows and Resources". Window definition IDs for windows of your own design are discussed later under "Defining Your Own Windows".

The behind parameter determines the window's plane. The new window is inserted in back of the window pointed to by this parameter. To put the new window behind all other windows, use behind=NIL. To place it in front of all other windows, use behind=POINTER(-1); in this case, NewWindow will unhighlight the previously active window, highlight the window being created, and generate appropriate activate events.

RefCon is the window's reference value, set and used only by your application.

NewWindow also sets the window class in the window record to indicate that the window was created directly by the application.

```
FUNCTION GetNewWindow (windowID: INTEGER; wStorage: Ptr; behind:
            WindowPtr) : WindowPtr;
```

Like NewWindow (above), GetNewWindow creates a window as specified by its parameters, adds it to the window list, and returns a windowPtr to the new window. The only difference between the two functions is that instead of having the parameters boundsRect, title, visible, procID, goAwayFlag, and refCon, GetNewWindow has a single windowID parameter, where windowID is the resource ID of a window template that supplies the same information as those parameters. The wStorage and behind parameters of GetNewWindow have the same meaning as in NewWindow.

```
PROCEDURE CloseWindow (theWindow: WindowPtr);
```

CloseWindow removes the given window from the screen and deletes it from the window list. It releases the memory occupied by all data structures associated with the window, but **not** the memory taken up by the window record itself. Call this procedure when you're done with a window if you supplied NewWindow or GetNewWindow a pointer to the window storage (in the wStorage parameter) when you created the window.

Any update events for the window are discarded. If the window was the frontmost window and there was another window behind it, the latter window is highlighted and an appropriate activate event is generated.

```
PROCEDURE DisposeWindow (theWindow: WindowPtr);
```

DisposeWindow calls CloseWindow (above) and then releases the memory occupied by the window record. Call this procedure when you're done with a window if you let the window record be allocated on the heap when you created the window (by passing NIL as the wStorage parameter to NewWindow or GetNewWindow).

Assembly-language note: The macro you invoke to call DisposeWindow from assembly language is named _DisposWindow.

Window Display

These procedures affect the appearance or plane of a window but not its size or location.

```
PROCEDURE SetWTitle (theWindow: WindowPtr; title: Str255);
```

SetWTitle sets theWindow's title to the given string, performing any necessary redrawing of the window frame.

```
PROCEDURE GetWTitle (theWindow: WindowPtr; VAR title: Str255);
```

GetWTitle returns theWindow's title as the value of the title parameter.

```
PROCEDURE SelectWindow (theWindow: WindowPtr);
```

SelectWindow makes theWindow the active window as follows: it unhighlights the previously active window, brings theWindow in front of all other windows, highlights theWindow, and generates the appropriate activate events. Call this procedure if there's a mouse-down event in the content region of an inactive window.

```
PROCEDURE HideWindow (theWindow: WindowPtr);
```

HideWindow makes theWindow invisible. If theWindow is the frontmost window and there's a window behind it, HideWindow also unhighlights theWindow, brings the window behind it to the front, highlights that window, and generates appropriate activate events (see Figure 6). If

theWindow is already invisible, HideWindow has no effect.

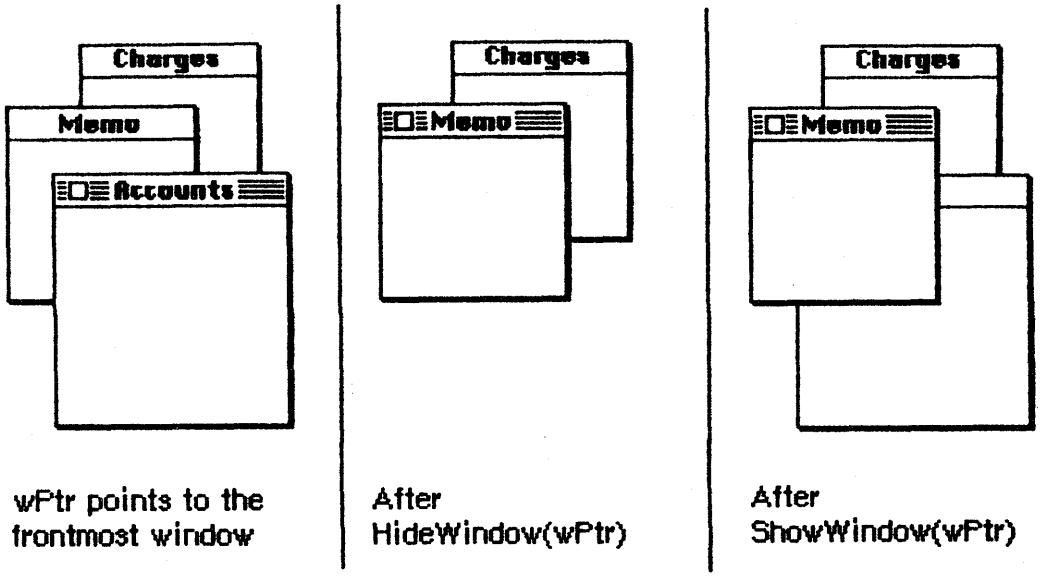


Figure 6. Hiding and Showing Document Windows

PROCEDURE ShowWindow (theWindow: WindowPtr);

ShowWindow makes theWindow visible. It does not change the front-to-back ordering of the windows. Remember that if you previously hid the frontmost window with HideWindow, HideWindow will have brought the window behind it to the front; so if you then do a ShowWindow of the window you hid, it will no longer be frontmost (see Figure 6 above). If theWindow is already visible, ShowWindow has no effect.

(note)

Although it's inadvisable, you can create a situation where the frontmost window is invisible. If you do a ShowWindow of such a window, it will highlight the window if it's not already highlighted and will generate an activate event to force this window from inactive to active.

PROCEDURE ShowHide (theWindow: WindowPtr; showFlag: BOOLEAN);

If showFlag is FALSE, ShowHide makes theWindow invisible if it's not already invisible and has no effect if it is already invisible. If showFlag is TRUE, ShowHide makes theWindow visible if it's not already visible and has no effect if it is already visible. Unlike HideWindow and ShowWindow, ShowHide never changes the highlighting or front-to-back ordering of windows or generates activate events.

(warning)

Use this procedure carefully, and only in special circumstances where you need more control than allowed by

HideWindow and ShowWindow.

```
PROCEDURE HiliteWindow (theWindow: WindowPtr; fHilite: BOOLEAN);
```

If fHilite is TRUE, this procedure highlights theWindow if it's not already highlighted and has no effect if it is highlighted. If fHilite is FALSE, HiliteWindow unhighlights theWindow if it is highlighted and has no effect if it's not highlighted. The exact way a window is highlighted depends on its window definition function.

Normally you won't have to call this procedure, since you should call SelectWindow to make a window active, and SelectWindow takes care of the necessary highlighting changes. Highlighting a window that isn't the active window is contrary to the Macintosh User Interface Guidelines.

```
PROCEDURE BringToFront (theWindow: WindowPtr);
```

BringToFront brings theWindow to the front of all other windows and redraws the window as necessary. Normally you won't have to call this procedure, since you should call SelectWindow to make a window active, and SelectWindow takes care of bringing the window to the front. If you do call BringToFront, however, remember to call HiliteWindow to make the necessary highlighting changes.

```
PROCEDURE SendBehind (theWindow: WindowPtr; behindWindow: WindowPtr);
```

SendBehind sends theWindow behind behindWindow, redrawing any exposed windows. If behindWindow is NIL, it sends theWindow behind all other windows. If theWindow is the active window, it unhighlights theWindow, highlights the new active window, and generates the appropriate activate events.

(warning)

Do not use SendBehind to deactivate a previously active window. Calling SelectWindow to make a window active takes care of deactivating the previously active window.

(note)

If you're moving theWindow closer to the front (that is, if it's initially even farther behind behindWindow), you must make the following calls after calling SendBehind:

```
wPeek := POINTER(theWindow);
PaintOne(wPeek, wPeek^.strucRgn);
CalcVis(wPeek, wPeek^.strucRgn)
```

PaintOne and CalcVis are described below under "Low-Level Routines".

```
FUNCTION FrontWindow : WindowPtr;
```

FrontWindow returns a pointer to the first visible window in the window list (that is, the active window). If there are no visible windows, it returns NIL.

Assembly-language note: In the global variable ghostWindow, you can store a pointer to a window that's not to be considered frontmost even if it is (for example, if you want to have a special editing window always present and floating above all the others). If the window pointed to by ghostWindow is the first window in the window list, FrontWindow will return a pointer to the next visible window.

```
PROCEDURE DrawGrowIcon (theWindow: WindowPtr);
```

Call DrawGrowIcon in response to an update or activate event involving a window that contains a size box in its content region. If theWindow is active (highlighted), DrawGrowIcon draws the size box; otherwise, it draws whatever is appropriate to show that the window temporarily cannot be sized. The exact appearance and location of what's drawn depend on the window definition function. For an active document window, DrawGrowIcon draws the size box icon in the bottom right corner of the portRect of the window's grafPort, along with the lines delimiting the size box and scroll bar areas (15 pixels in from the right edge and bottom of the portRect). It doesn't erase the scroll bar areas, so if the window doesn't contain scroll bars you should erase those areas yourself after the window's size changes. For an inactive document window, DrawGrowIcon draws only the delimiting lines (again, without erasing anything).

Mouse Location

```
FUNCTION FindWindow (thePt: Point; VAR whichWindow: WindowPtr) :
            INTEGER;
```

When a mouse-down event occurs, the application should call FindWindow with thePt equal to the point where the mouse button was pressed (in global coordinates, as stored in the where field of the event record). FindWindow tells which part of which window, if any, the mouse button was pressed in. If it was pressed in a window, the whichWindow parameter is set to the window pointer; otherwise, it's set to NIL. The integer returned by FindWindow is one of the following predefined constants:

```
CONST inDesk      = Ø;  {none of the following}
      inMenuBar   = 1;  {in menu bar}
      inSysWindow = 2;  {in system window}
      inContent   = 3;  {in content region (except grow, if active)}
      inDrag      = 4;  {in drag region}
      inGrow      = 5;  {in grow region (active window only)}
      inGoAway    = 6;  {in go-away region (active window only)}
```

InDesk usually means that the mouse button was pressed on the desktop, outside the menu bar or any windows; however, it may also mean that the mouse button was pressed inside a window frame but not in the drag region or go-away region of the window. Usually one of the last four values is returned for windows created by the application.

Assembly-language note: If you store a pointer to a procedure in the global variable deskHook, it will be called when the mouse button is pressed on the desktop. The deskHook procedure will be called with -1 in DØ to distinguish this use of it from its use in drawing the desktop (discussed in the description of InitWindows). AØ will contain a pointer to the event record for the mouse-down event. When you use deskHook in this way, FindWindow does not return inDesk when the mouse button is pressed on the desktop; it returns inSysWindow, and the Desk Manager procedure SystemClick calls the deskHook procedure.

If the window is a documentProc type of window that doesn't contain a size box, the application should treat inGrow the same as inContent; if it's a noGrowDocProc type of window, FindWindow will never return inGrow for that window. If the window is a documentProc, noGrowDocProc, or rDocProc type of window with no close box, FindWindow will never return inGoAway for that window.

```
FUNCTION TrackGoAway (theWindow: WindowPtr; thePt: Point) : BOOLEAN;
```

When there's a mouse-down event in the go-away region of theWindow, the application should call TrackGoAway with thePt equal to the point where the mouse button was pressed (in global coordinates, as stored in the where field of the event record). TrackGoAway keeps control until the mouse button is released, highlighting the go-away region as long as the mouse position remains inside it, and unhighlighting it when the mouse moves outside it. The exact way a window's go-away region is highlighted depends on its window definition function; the highlighting of a document window's close box is illustrated in Figure 7. When the mouse button is released, TrackGoAway unhighlights the go-away region and returns TRUE if the mouse is inside the go-away region or FALSE if it's outside the region (in which case the application should do nothing).

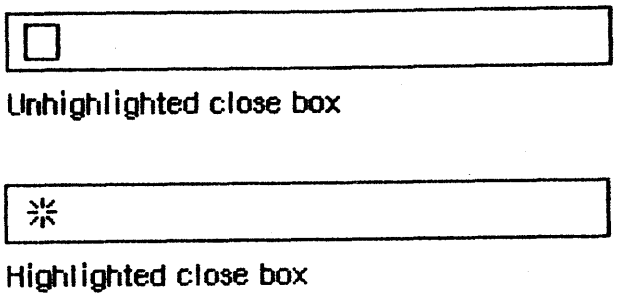


Figure 7. A Document Window's Close Box

Window Movement and Sizing

```
PROCEDURE MoveWindow (theWindow: WindowPtr; hGlobal,vGlobal: INTEGER;
          front: BOOLEAN);
```

MoveWindow moves theWindow to another part of the screen, without affecting its size or plane. The top left corner of the portRect of the window's grafPort is moved to the screen point indicated by the global coordinates hGlobal and vGlobal. The local coordinates of the top left corner remain the same; MoveWindow saves those coordinates before moving the window and calls the QuickDraw procedure SetOrigin to restore them before returning. If the front parameter is TRUE and theWindow isn't the active window, MoveWindow makes it the active window by calling SelectWindow(theWindow).

```
PROCEDURE DragWindow (theWindow: WindowPtr; startPt: Point; boundsRect:
          Rect);
```

When there's a mouse-down event in the drag region of theWindow, the application should call DragWindow with startPt equal to the point where the mouse button was pressed (in global coordinates, as stored in the where field of the event record). DragWindow pulls a gray outline of theWindow around, following the movements of the mouse until the button is released. When the mouse button is released, DragWindow calls MoveWindow to move theWindow to the location to which it was dragged. If theWindow is not the active window and the Command key was not being held down, DragWindow makes it the active window (by passing TRUE for the front parameter when calling MoveWindow).

BoundsRect is also given in global coordinates. If the mouse button is released when the mouse position is outside the limits of boundsRect, DragWindow returns without moving theWindow or making it the active window. For a document window, boundsRect typically will be four pixels in from the menu bar and from the other edges of the screen, to

ensure that there won't be less than a four-pixel-square area of the title bar visible on the screen.

Assembly-language note: By storing a pointer to a procedure in the global variable dragHook, you can specify a procedure to be executed repeatedly for as long as the user holds down the mouse button. (DragWindow calls DragGrayRgn, described under "Miscellaneous Utilities" below, and passes the pointer in dragHook as DragGrayRgn's actionProc parameter.)

```
FUNCTION GrowWindow (theWindow: WindowPtr; startPt: Point; sizeRect:
          Rect) : LongInt;
```

When there's a mouse-down event in the grow region of theWindow, the application should call GrowWindow with startPt equal to the point where the mouse button was pressed (in global coordinates, as stored in the where field of the event record). GrowWindow pulls a grow image of the window around, following the movements of the mouse until the button is released. The grow image for a document window is a gray outline of the entire window and also the lines delimiting the title bar, size box, and scroll bar areas; Figure 8 illustrates this for a document window containing a size box and scroll bars, but the grow image would be the same even if the window contained no size box, one scroll bar, or no scroll bars. In general, the grow image is defined in the window definition function and is whatever is appropriate to show that the window's size will change.

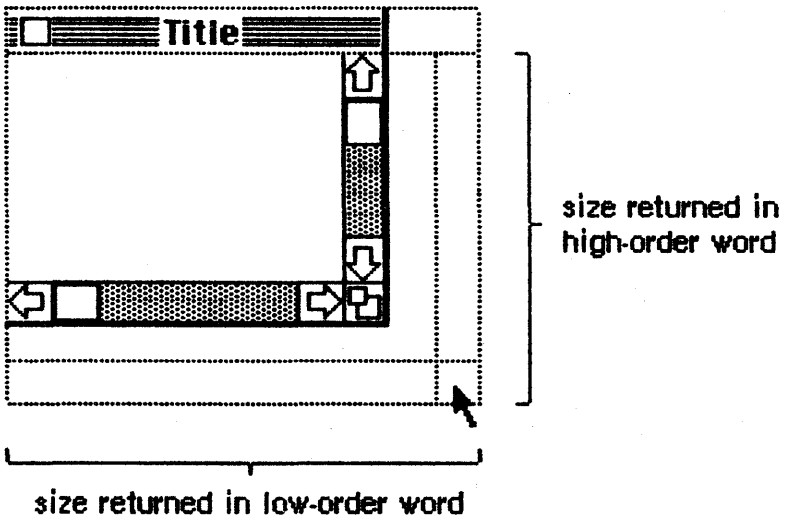


Figure 8. GrowWindow Operation on a Document Window

The application should subsequently call SizeWindow (see below) to change the portRect of the window's grafPort to the new one outlined by

the grow image. The sizeRect parameter specifies limits, in pixels, on the vertical and horizontal measurements of what will be the new portRect. SizeRect.top is the minimum vertical measurement, sizeRect.left is the minimum horizontal measurement, sizeRect.bottom is the maximum vertical measurement, and sizeRect.right is the maximum horizontal measurement.

GrowWindow returns the actual size for the new portRect as outlined by the grow image when the mouse button is released. The high-order word of the LongInt is the vertical measurement in pixels and the low-order word is the horizontal measurement. A return value of Ø indicates that the size is the same as that of the current portRect.

(note)

The Toolbox Utility function HiWord takes a long integer as a parameter and returns an integer equal to its high-order word; the function LoWord returns the low-order word.

```
PROCEDURE SizeWindow (theWindow: WindowPtr; w,h: INTEGER; fUpdate:
          BOOLEAN);
```

SizeWindow enlarges or shrinks the portRect of theWindow's grafPort to the width and height specified by w and h, or does nothing if w and h are Ø. The window's position on the screen does not change. The new window frame is drawn; if the width of a document window changes, the title is again centered in the title bar, or is truncated at its end if it no longer fits. If fUpdate is TRUE, SizeWindow accumulates any newly created area of the content region into the update region (see Figure 9); normally this is what you'll want. If you pass FALSE for fUpdate, you're responsible for the update region maintenance yourself. For more information, see InvalRect and ValidRect below.

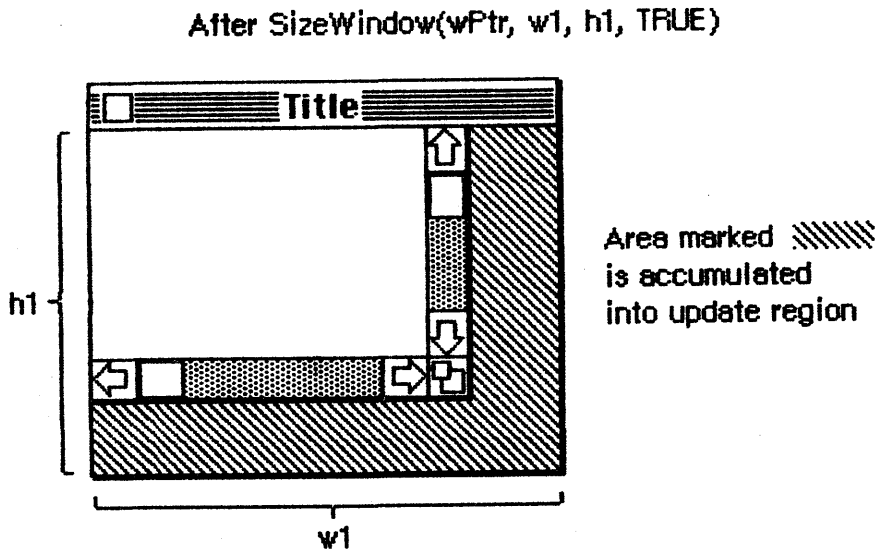


Figure 9. SizeWindow Operation on a Document Window

(note)
You should change the window's size only when the user has done something specific to make it change.

Update Region Maintenance

PROCEDURE InvalRect (badRect: Rect);

InvalRect accumulates the given rectangle into the update region of the window whose grafPort is the current port. This tells the Window Manager that the rectangle has changed and must be updated. The rectangle lies within the window's content region and is given in the local coordinates.

For example, this procedure is useful when you're calling SizeWindow (described above) for a document window that contains a size box or scroll bars. Suppose you're going to call SizeWindow with fUpdate=TRUE. If the window is enlarged as shown in Figure 8 above, you'll want not only the newly created part of the content region to be updated, but also the two rectangular areas containing the (former) size box and scroll bars; before calling SizeWindow, you can call InvalRect twice to accumulate those areas into the update region. In case the window is made smaller, you'll want the new size box and scroll bar areas to be updated, and so can similarly call InvalRect for those areas after calling SizeWindow. See Figure 1Ø for an illustration of this type of update region maintenance.

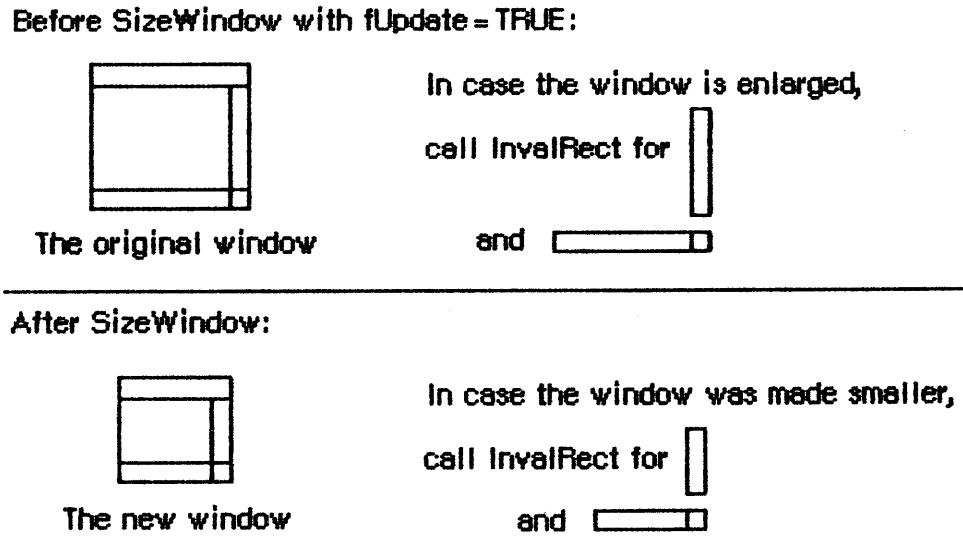


Figure 1Ø. Update Region Maintenance with InvalRect

As another example, suppose your application scrolls up text in a document window and wants to show new text added at the bottom of the

window. You can cause the added text to be redrawn by accumulating that area into the update region with InvalRect.

```
PROCEDURE InvalRgn (badRgn: RgnHandle);
```

InvalRgn is the same as InvalRect (above) but for a region that has changed rather than a rectangle.

```
PROCEDURE ValidRect (goodRect: Rect);
```

ValidRect removes goodRect from the update region of the window whose grafPort is the current port. This tells the Window Manager that the application has already drawn the rectangle and to cancel any updates accumulated for that area. The rectangle lies within the window's content region and is given in local coordinates. Using ValidRect results in better performance and less redundant redrawing in the window.

For example, suppose you've called SizeWindow (described above) with fUpdate=TRUE for a document window that contains a size box or scroll bars. Depending on the dimensions of the newly sized window, the new size box and scroll bar areas may or may not have been accumulated into the window's update region. After calling SizeWindow, you can redraw the size box or scroll bars immediately and then call ValidRect for the areas they occupy in case they were in fact accumulated into the update region; this will avoid redundant drawing.

```
PROCEDURE ValidRgn (goodRgn: RgnHandle);
```

ValidRgn is the same as ValidRect (above) but for a region that has been drawn rather than a rectangle.

```
PROCEDURE BeginUpdate (theWindow: WindowPtr);
```

Call BeginUpdate when an update event occurs for theWindow. BeginUpdate replaces the visRgn of the window's grafPort with the intersection of the visRgn and the update region and then sets the window's update region to the empty region. You would then usually draw the entire content region, though it suffices to draw only the visRgn; in either case, only the parts of the window that require updating will actually be drawn on the screen. Every call to BeginUpdate must be balanced by a call to EndUpdate. (See below, and see "How a Window is Drawn".)

```
PROCEDURE EndUpdate (theWindow: WindowPtr);
```

Call EndUpdate to restore the normal visRgn of theWindow's grafPort, which was changed by BeginUpdate as described above.

Miscellaneous Utilities

```
PROCEDURE SetWRefCon (theWindow: WindowPtr; data: LongInt);
```

SetWRefCon sets theWindow's reference value to the given data.

```
FUNCTION GetWRefCon (theWindow: WindowPtr) : LongInt;
```

GetWRefCon returns theWindow's current reference value.

```
PROCEDURE SetWindowPic (theWindow: WindowPtr; pic: PicHandle);
```

SetWindowPic stores the given picture handle in the window record for theWindow, so that when theWindow's contents are to be drawn, the Window Manager will draw this picture rather than generate an update event.

```
FUNCTION GetWindowPic (theWindow: WindowPtr) : PicHandle;
```

GetWindowPic returns the handle to the picture that draws theWindow's contents, previously stored with SetWindowPic (above).

```
FUNCTION PinRect (theRect: Rect; thePt: Point) : LongInt;
```

PinRect "pins" thePt inside theRect: The high-order word of the function result is the vertical coordinate of thePt or, if thePt lies above or below theRect, the vertical coordinate of the top or bottom of theRect, respectively. The low-order word of the function result is the horizontal coordinate of thePt or, if thePt lies to the left or right of theRect, the horizontal coordinate of the left or right edge of theRect.

```
FUNCTION DragGrayRgn (theRgn: RgnHandle; startPt: Point;
            limitRect,slopRect: Rect; axis: INTEGER; actionProc:
            ProcPtr) : LongInt;
```

Called when the mouse button is down inside theRgn, DragGrayRgn pulls a gray outline of the region around, following the movements of the mouse until the button is released. DragWindow calls this function before actually moving the window, and the Control Manager routine DragControl similarly calls it for controls. You can call it yourself to pull around the outline of any region, and then use the information it returns to determine where to move the region.

The startPt parameter is assumed to be the point where the mouse button was originally pressed, in the local coordinates of the current

grafPort.

LimitRect and slopRect are also in the local coordinates of the current grafPort. To explain these parameters, the concept of "offset point" must be introduced: this is the point whose vertical and horizontal offsets from the top left corner of the region's enclosing rectangle are the same as those of startPt. Initially the offset point is the same as the mouse position, but they may differ, depending on where the user moves the mouse. DragGrayRgn will never move the offset point outside limitRect; this limits the travel of the region's outline (but not the movements of the mouse). SlopRect, which should completely enclose limitRect, allows the user some "slop" in moving the mouse. DragGrayRgn's behavior while tracking the mouse depends on the position of the mouse with respect to these two rectangles:

- When the mouse is inside limitRect, the region's outline follows it normally. If the mouse button is released there, the region should be moved to the mouse position.

- When the mouse is outside limitRect but inside slopRect, DragGrayRgn "pins" the offset point to the edge of limitRect. If the mouse button is released there, the region should be moved to this pinned location.

- When the mouse is outside slopRect, the outline disappears from the screen, but DragGrayRgn continues to follow the mouse; if it moves back into slopRect, the outline reappears. If the mouse button is released outside slopRect, the region should not be moved from its original position.

Figure 11 illustrates what happens when the mouse is moved outside limitRect but inside slopRect, for a rectangular region. The offset point is pinned as the mouse position moves on.

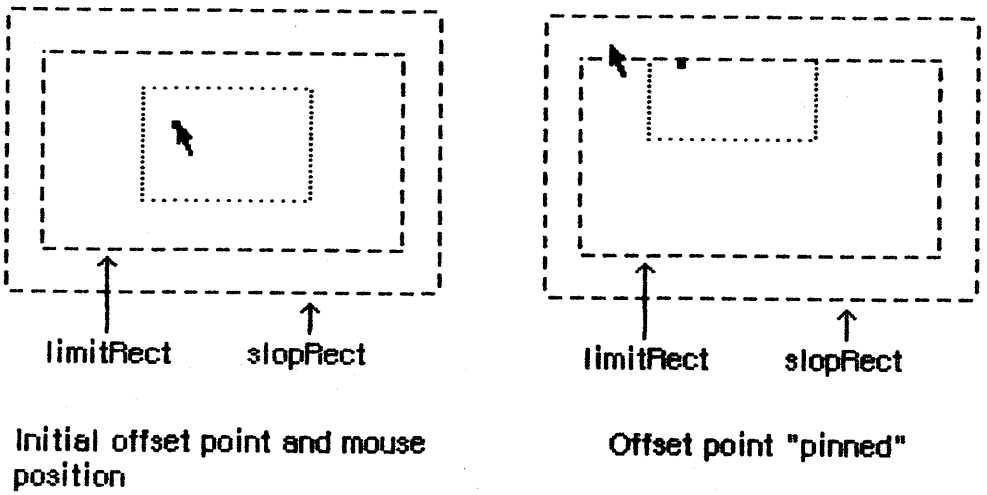


Figure 11. DragGrayRgn Operation on a Rectangular Region

If the mouse button is released outside slopRect, DragGrayRgn returns -32768 ($8000); otherwise, the high-order word of the value returned contains the vertical coordinate of the ending mouse point minus that

of startPt and the low-order word contains the difference between the horizontal coordinates.

The axis parameter allows you to constrain the outline's motion to only one axis. It has one of the following values:

```
CONST noConstraint = Ø;  {no constraint}
      hAxisOnly    = 1;  {horizontal axis only}
      vAxisOnly    = 2;  {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. With or without an axis constraint, the mouse must still be inside the slop rectangle for the outline to appear at all.

The actionProc 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 actionProc is NIL, DragGrayRgn simply retains control until the mouse button is released, performing no action while the mouse button is down.

Assembly-language note: If you want the region's outline to be drawn in a pattern other than gray, you can store the pattern in the global variable dragPattern and call the above function at the entry point _DragTheRgn.

Low-Level Routines

These low-level routines are not normally used by an application but may be of interest to advanced programmers.

```
FUNCTION CheckUpdate (VAR theEvent: EventRecord) : BOOLEAN;
```

CheckUpdate is called by the Toolbox Event Manager. From the front to the back in the window list, it looks for a visible window that needs updating (that is, whose update region is not empty). If it finds one whose window record contains a picture handle, it draws the picture (doing all the necessary region manipulation) and looks for the next visible window that needs updating. If it ever finds one whose window record doesn't contain a picture handle, it stores an update event for that window in theEvent and returns TRUE. If it never finds such a window, it returns FALSE.

```
PROCEDURE ClipAbove (window: WindowPeek);
```

ClipAbove sets the clipRgn of the Window Manager port to be the desktop (global variable grayRgn) intersected with the current clipRgn, minus the structure regions of all the windows above the given window.

```
PROCEDURE SaveOld (window: WindowPeek);
```

SaveOld saves the given window's current structure region and content region for the DrawNew operation (see below). It must be balanced by a subsequent call to DrawNew.

```
PROCEDURE DrawNew (window: WindowPeek; update: BOOLEAN);
```

If the update parameter is TRUE, DrawNew updates the area

(oldStruct XOR newStruct) UNION (oldContent XOR newContent)

where oldStruct and oldContent are the structure and content regions saved by the SaveOld procedure, and newStruct and newContent are the current structure and content regions. It paints the area white and adds it to the window's update region. If update is FALSE, it only paints the area white.

(warning)
SaveOld and DrawNew are **not** nestable.

```
PROCEDURE PaintOne (window: WindowPeek; clobberedRgn: RgnHandle);
```

PaintOne "paints" the given window, clipped to clobberedRgn and all windows above it: it draws the window frame and, if some content is exposed, paints the exposed area white and adds it to the window's update region. If the window parameter is NIL, the window is the desktop and so is painted gray.

```
PROCEDURE PaintBehind (startWindow: WindowPeek; clobberedRgn:
          RgnHandle);
```

PaintBehind calls PaintOne (above) for startWindow and all the windows behind startWindow, clipped to clobberedRgn.

```
PROCEDURE CalcVis (window: WindowPeek);
```

CalcVis calculates the visRgn of the given window by starting with its content region and subtracting the structure region of each window in front of it.

```
PROCEDURE CalcVisBehind (startWindow: WindowPeek; clobberedRgn:
          RgnHandle);
```

CalcVisBehind calculates the visRgns of startWindow and all windows behind startWindow that intersect with clobberedRgn. It's called after PaintBehind (see above).

Assembly-language note: The macro you invoke to call CalcVisBehind from assembly language is named _CalcVBehind.

DEFINING YOUR OWN WINDOWS

Certain types of windows, such as the standard document window, are predefined for you. However, you may want to define your own type of window--maybe a round or hexagon-shaped window, or even a window shaped like an apple. QuickDraw and the Window Manager make it possible for you to do this.

(note)
For the convenience of your application's user, remember to conform to the Macintosh User Interface Guidelines for windows as much as possible.

To define your own type of window, you write a window definition function and (usually) store it in a resource file. When you create a window, you provide a window definition ID, which leads to the window definition function. The window definition ID is an integer that contains the resource ID of the window definition function in its upper 12 bits and a variation code in its lower four bits. Thus, for a given resource ID and variation code, the window definition ID is:

16 * resource ID + variation code

The variation code allows a single window definition function to implement several related types of window as "variations on a theme". For example, the dBoxProc type of window is a variation of the standard document window; both use the window definition function whose resource ID is Ø, but the document window has a variation code of Ø while the dBoxProc window has a variation code of 1.

The Window Manager calls the Resource Manager to access the window definition function with the given resource ID. The Resource Manager reads the window definition function into memory and returns a handle to it. The Window Manager stores this handle in the windowDefProc field of the window record, along with the variation code in the high-order byte of that field. Later, when it needs to perform a type-dependent action on the window, it calls the window definition function and passes it the variation code as a parameter. Figure 12 summarizes

this process.

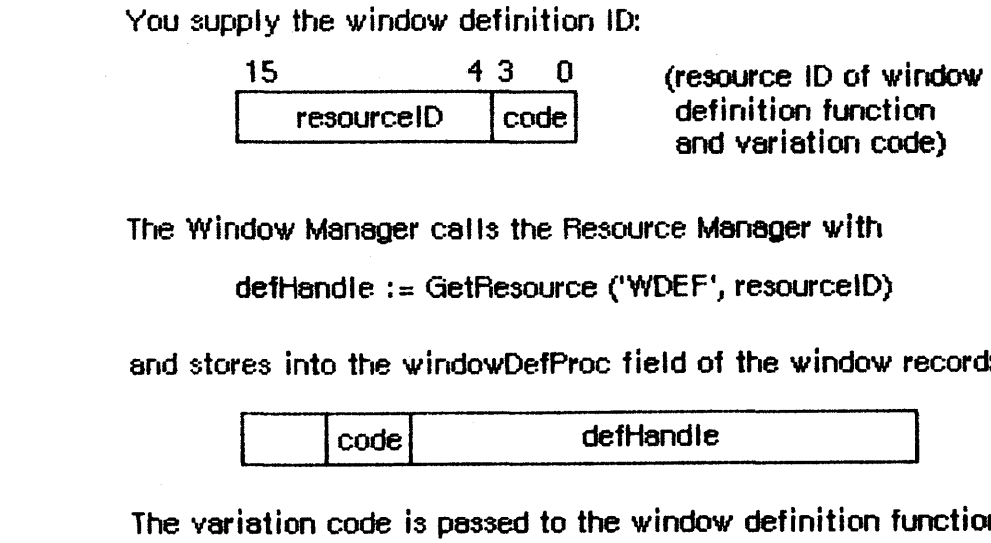


Figure 12. Window Definition Handling

(note)

You may find it more convenient to include the window definition function with the code of your program instead of storing it as a separate resource. If you do this, you should supply the window definition ID of any standard window type when you create the window, and specify that the window initially be invisible. Once the window is created, you can replace the contents of the windowDefProc field with as handle to the actual window definition function (along with a variation code, if needed, in the high-order byte of the field). You can then call ShowWindow to make the window visible.

The Window Definition Function

The window definition function may be written in Pascal or assembly language; the only requirement is that its entry point must be at the beginning. You may choose any name you wish for your window definition function. Here's how you would declare one named MyWindow:

```
FUNCTION MyWindow (varCode: INTEGER; theWindow: WindowPtr;
                   message: INTEGER; param: LongInt) : LongInt;
```

VarCode is the variation code, as described above.

TheWindow indicates the window that the operation will affect. If the window definition function needs to use a WindowPeek type of pointer more than a WindowPtr, you can simply specify WindowPeek instead of WindowPtr in the function declaration.

The message parameter identifies the desired operation. It has one of the following values:

```
CONST wDraw      = Ø;  {draw window frame}
      wHit       = 1;  {tell what region mouse button was pressed in}
      wCalcRgns  = 2;  {calculate strucRgn and contRgn}
      wNew       = 3;  {do any additional window initialization}
      wDispose   = 4;  {take any additional disposal actions}
      wGrow      = 5;  {draw window's grow image}
      wDrawGIcon = 6;  {draw size box in content region}
```

As described below in the discussions of the routines that perform these operations, the value passed for param, the last parameter of the window definition function, depends on the operation. Where it's not mentioned below, this parameter is ignored. Similarly, the window definition function is expected to return a function result only where indicated; in other cases, the function should return Ø.

(note)
"Routine" here does not necessarily mean a procedure or function. While it's a good idea to set these up as subprograms inside the window definition function, you're not required to do so.

The Draw Window Frame Routine

When the window definition function receives a wDraw message, it should draw the window frame in the current grafPort, which will be the Window Manager port. (For details on drawing, see the QuickDraw manual.)

(warning)
Do not change the visRgn or clipRgn of the Window Manager port, or overlapping windows may not be handled properly.

This routine should make certain checks to determine exactly what it should do. If the visible field in the window record is FALSE, the routine should do nothing; otherwise, it should examine the value of param received by the window definition function, as described below.

If param is Ø, the routine should draw the entire window frame. If the hilited field in the window record is TRUE, the window frame should be highlighted in whatever way is appropriate to show that this is the active window. If goAwayFlag in the window record is also TRUE, the highlighted window frame should include a go-away region; this is useful when you want to define a window such that a particular window of that type may or may not have a go-away region, depending on the situation.

Special action should be taken if the value of param is wInGoAway (a predefined constant, equal to 4, which is one of those returned by the hit routine described below). If param is wInGoAway, the routine should do nothing but "toggle" the state of the window's go-away region from unhighlighted to highlighted or vice versa. The highlighting

should be whatever is appropriate to show that the mouse button has been pressed inside the region. Simple inverse highlighting may be used or, as in document windows, the appearance of the region may change considerably. In the latter case, the routine could use a "mask" consisting of the unhighlighted state of the region XORed with its highlighted state (where XOR stands for the logical operation "exclusive or"). When such a mask is itself XORed with either state of the region, the result is the other state; Figure 13 illustrates this.

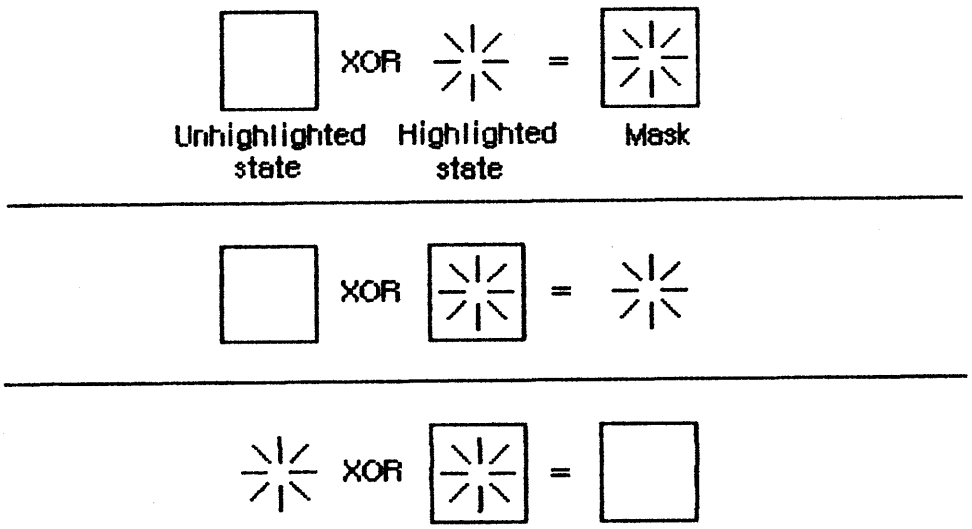


Figure 13. Toggling the Go-Away Region

Typically the window frame will include the window's title, which should be in the system font and system font size for consistency with the Macintosh User Interface Guidelines. The Window Manager port will already be set to use the system font and system font size.

(note)
Nothing drawn outside the window's structure region will be visible.

The Hit Routine

When the window definition function receives a wHit message, it also receives as its param value the point where the mouse button was pressed. This point is given in global coordinates, with the vertical coordinate in the high-order word of the LongInt and the horizontal coordinate in the low-order word. The window definition function should determine where the mouse button "hit" and then return one of these predefined constants:

```
CONST wNoHit     = Ø;  {none of the following}
      wInContent = 1;  {in content region (except grow, if active)}
      wInDrag    = 2;  {in drag region}
      wInGrow    = 3;  {in grow region (active window only)}
      wInGoAway  = 4;  {in go-away region (active window only)}
```

Usually, wNoHit means the given point isn't anywhere within the window, but this is not necessarily so. For example, the document window's hit routine returns wNoHit if the point is in the window frame but not in the title bar.

The constants wInGrow and wInGoAway should be returned only if the window is active, since by convention the size box and go-away region won't be drawn if the window is inactive (or, if drawn, won't be operable). In an inactive document window, if the mouse button is pressed in the title bar where the close box would be if the window were active, the hit routine should return wInDrag.

Of the regions that may have been hit, only the content region necessarily has the structure of a region and is included in the window record. The hit routine can determine in any way it likes whether the drag, grow, or go-away "region" has been hit.

The Routine to Calculate Regions

The routine executed in response to a wCalcRgns message should calculate the window's structure region and content region based on the current grafPort's portRect. These regions, whose handles are in the strucRgn and contRgn fields, are in global coordinates. The Window Manager will request this operation only if the window is visible.

(warning)

> When you calculate regions for your own type of window, do not alter the clipRgn or the visRgn of the window's grafPort. The Window Manager and QuickDraw take care of this for you. Altering the clipRgn or visRgn may result in damage to other windows.

The Initialize Routine

After initializing fields as appropriate when creating a new window, the Window Manager sends the message wNew to the window definition function. This gives the definition function a chance to perform any type-specific initialization it may require. For example, if the content region is unusually shaped, the initialize routine might allocate space for the region and store the region handle in the dataHandle field of the window record. The initialize routine for a document window does nothing.

The Dispose Routine

The Window Manager's CloseWindow and DisposeWindow procedures send the message wDispose to the window definition function, telling it to carry out any additional actions required when disposing of the window. The dispose routine might, for example, release space that was allocated by the initialize routine. The dispose routine for a document window does nothing.

The Grow Routine

When the window definition function receives a wGrow message, it also receives a pointer to a rectangle as its param value. The rectangle is in global coordinates and is usually aligned at its top left corner with the portRect of the window's grafPort. The grow routine should draw a grow image of the window to fit the given rectangle (that is, whatever is appropriate to show that the window's size will change, such as an outline of the content region). The Window Manager requests this operation repeatedly as the user drags inside the grow region. The grow routine should draw in the current grafPort, which will be the Window Manager port, and should use the grafPort's current pen pattern and pen mode, which are set up (as gray and notPatXor) to conform to the Macintosh User Interface Guidelines.

The grow routine for a document window draws a gray outline of the window and also the lines delimiting the title bar, size box, and scroll bar areas.

The Draw Size Box Routine

Thw wDrawGIcon message tells the window definition function to draw the size box in the content region of the window if the window is active (highlighted) or, if the window is inactive, whatever is appropriate to show that it temporarily can't be sized. For active document windows, this routine draws the size box icon in the bottom right corner of the portRect of the window's grafPort, along with the lines delimiting the size box and scroll bar areas; for inactive windows, it draws just the delimiting lines.

(note)
If the size box is located in the window frame rather than the content region, this routine should do nothing.

FORMATS OF RESOURCES FOR WINDOWS

The Window Manager function GetNewWindow takes the resource ID of a window template as a parameter, and gets from the template the same infomation that the NewWindow function gets from six of its parameters. The resource type for a window template is 'WIND', and the resource data has the following format:

Number of bytes	Contents
8 bytes	Same as boundsRect parameter to NewWindow
2 bytes	Same as procID parameter to NewWindow
2 bytes	Same as visible parameter to NewWindow
2 bytes	Same as goAwayFlag parameter to NewWindow
4 bytes	Same as refCon parameter to NewWindow
n bytes	Same as title parameter to NewWindow (1-byte length in bytes, followed by the characters of the title)

The resource type for a window definition function is 'WDEF', and the resource data is simply the compiled or assembled code of the function.

SUMMARY OF THE WINDOW MANAGER

Constants

```
CONST { Window definition IDs }

      documentProc  = Ø;    {standard document window}
      dBoxProc      = 1;    {alert box or modal dialog box}
      plainDBox     = 2;    {plain box}
      altDBoxProc   = 3;    {plain box with shadow}
      noGrowDocProc = 4;    {document window without size box}
      rDocProc      = 16;   {rounded-corner window}

      { Window class, in windowKind field of window record }

      dialogKind  = 2;   {dialog or alert window}
      userKind    = 8;   {window created directly by the application}

      { Values returned by FindWindow }

      inDesk      = Ø;   {none of the following}
      inMenuBar   = 1;   {in menu bar}
      inSysWindow = 2;   {in system window}
      inContent   = 3;   {in content region (except grow, if active)}
      inDrag      = 4;   {in drag region}
      inGrow      = 5;   {in grow region (active window only)}
      inGoAway    = 6;   {in go-away region (active window only)}

      { Axis constraints for DragGrayRgn }

      noConstraint = Ø;  {no constraint}
      hAxisOnly    = 1;  {horizontal axis only}
      vAxisOnly    = 2;  {vertical axis only}

      { Messages to window definition function }

      wDraw      = Ø;   {draw the window frame}
      wHit       = 1;   {tell what region mouse button was pressed in}
      wCalcRgns  = 2;   {calculate strucRgn and contRgn}
      wNew       = 3;   {do any additional window initialization}
      wDispose   = 4;   {take any additional disposal actions}
      wGrow      = 5;   {draw window's grow image}
      wDrawGIcon = 6;   {draw size box in content region}

      { Values returned by window definition function's hit routine }

      wNoHit     = Ø;   {none of the following}
      wInContent = 1;   {in content region (except grow, if active)}
      wInDrag    = 2;   {in drag region}
      wInGrow    = 3;   {in grow region (active window only)}
      wInGoAway  = 4;   {in go-away region (active window only)}
```

Data Types

```
TYPE WindowPtr  =  GrafPtr;
     WindowPeek =  ^WindowRecord;

     WindowRecord =
              RECORD
                port:          GrafPort;    {window's grafPort}
                windowKind:    INTEGER;     {window class}
                visible:       BOOLEAN;     {TRUE if visible}
                hilited:       BOOLEAN;     {TRUE if highlighted}
                goAwayFlag:    BOOLEAN;     {TRUE if has go-away region}
                spareFlag:     BOOLEAN;     {reserved for future use}
                strucRgn:      RgnHandle;   {structure region}
                contRgn:       RgnHandle;   {content region}
                updateRgn:     RgnHandle;   {update region}
                windowDefProc: Handle;      {window definition function}
                dataHandle:    Handle;      {data used by windowDefProc}
                titleHandle:   StringHandle; {window's title}
                titleWidth:    INTEGER;     {width of title in pixels}
                controlList:   Handle;      {window's control list}
                nextWindow:    WindowPeek;  {next window in window list}
                windowPic:     PicHandle;   {picture for drawing window}
                refCon:        LongInt      {window's reference value}
              END;
```

Routines

Initialization and Allocation

```
PROCEDURE InitWindows;
PROCEDURE GetWMgrPort   (VAR wPort: GrafPtr);
FUNCTION  NewWindow     (wStorage: Ptr; boundsRect: Rect; title: Str255;
                         visible: BOOLEAN; procID: INTEGER; behind:
                         WindowPtr; goAwayFlag: BOOLEAN; refCon: LongInt)
                         : WindowPtr;
FUNCTION  GetNewWindow  (windowID: INTEGER; wStorage: Ptr; behind:
                         WindowPtr) : WindowPtr;
PROCEDURE CloseWindow   (theWindow: WindowPtr);
PROCEDURE DisposeWindow (theWindow: WindowPtr);
```

Window Display

```
PROCEDURE SetWTitle    (theWindow: WindowPtr; title: Str255);
PROCEDURE GetWTitle    (theWindow: WindowPtr; VAR title: Str255);
PROCEDURE SelectWindow (theWindow: WindowPtr);
PROCEDURE HideWindow   (theWindow: WindowPtr);
PROCEDURE ShowWindow   (theWindow: WindowPtr);
PROCEDURE ShowHide     (theWindow: WindowPtr; showFlag: BOOLEAN);
```

```
PROCEDURE HiliteWindow (theWindow: WindowPtr; fHilite: BOOLEAN);
PROCEDURE BringToFront (theWindow: WindowPtr);
PROCEDURE SendBehind   (theWindow: WindowPtr; behindWindow: WindowPtr);
FUNCTION  FrontWindow : WindowPtr;
PROCEDURE DrawGrowIcon (theWindow: WindowPtr);
```

Mouse Location

```
FUNCTION FindWindow  (thePt: Point; VAR whichWindow: WindowPtr) :
                      INTEGER;
FUNCTION TrackGoAway (theWindow: WindowPtr; thePt: Point) : BOOLEAN;
```

Window Movement and Sizing

```
PROCEDURE MoveWindow (theWindow: WindowPtr; hGlobal,vGlobal: INTEGER;
                      front: BOOLEAN);
PROCEDURE DragWindow (theWindow: WindowPtr; startPt: Point; boundsRect:
                      Rect);
FUNCTION  GrowWindow (theWindow: WindowPtr; startPt: Point; sizeRect:
                      Rect) : LongInt;
PROCEDURE SizeWindow (theWindow: WindowPtr; w,h: INTEGER; fUpdate:
                      BOOLEAN);
```

Update Region Maintenance

```
PROCEDURE InvalRect   (badRect: Rect);
PROCEDURE InvalRgn    (badRgn: RgnHandle);
PROCEDURE ValidRect   (goodRect: Rect);
PROCEDURE ValidRgn    (goodRgn: RgnHandle);
PROCEDURE BeginUpdate (theWindow: WindowPtr);
PROCEDURE EndUpdate   (theWindow: WindowPtr);
```

Miscellaneous Utilities

```
PROCEDURE SetWRefCon   (theWindow: WindowPtr; data: LongInt);
FUNCTION  GetWRefCon   (theWindow: WindowPtr) : LongInt;
PROCEDURE SetWindowPic (theWindow: WindowPtr; pic: PicHandle);
FUNCTION  GetWindowPic (theWindow: WindowPtr) : PicHandle;
FUNCTION  PinRect      (theRect: Rect; thePt: Point) : LongInt;
FUNCTION  DragGrayRgn  (theRgn: RgnHandle; startPt: Point; limitRect,
                        slopRect: Rect; axis: INTEGER; actionProc:
                        ProcPtr) : LongInt;
```

Low-Level Routines

```
FUNCTION  CheckUpdate   (VAR theEvent: EventRecord) : BOOLEAN;
PROCEDURE ClipAbove     (window: WindowPeek);
PROCEDURE SaveOld       (window: WindowPeek);
PROCEDURE DrawNew       (window: WindowPeek; update: BOOLEAN);
```

```
PROCEDURE PaintOne      (window: WindowPeek; clobberedRgn: RgnHandle);
PROCEDURE PaintBehind   (startWindow: WindowPeek; clobberedRgn:
                         RgnHandle);
PROCEDURE CalcVis       (window: WindowPeek);
PROCEDURE CalcVisBehind (startWindow: WindowPeek; clobberedRgn:
                         RgnHandle);
```

Diameters of Curvature for Rounded-Corner Windows

Window definition ID	Diameters of curvature
rDocProc	16, 16
rDocProc + 1	4, 4
rDocProc + 2	6, 6
rDocProc + 3	8, 8
rDocProc + 4	1Ø, 1Ø
rDocProc + 5	12, 12
rDocProc + 6	2Ø, 2Ø
rDocProc + 7	24, 24

Window Definition Function

```
FUNCTION MyWindow (varCode: INTEGER; theWindow: WindowPtr; message:
                   INTEGER; param: LongInt) : LongInt;
```

Assembly-Language Information

Constants

```
; Window definition IDs

documentProc    .EQU  Ø   ;standard document window
dBoxProc        .EQU  1   ;alert box or modal dialog box
plainDBox       .EQU  2   ;dBoxProc without border
altDBoxProc     .EQU  3   ;dBoxProc with shadow instead of border
noGrowDocProc   .EQU  4   ;document window without size box
rDocProc        .EQU  16  ;rounded-corner window

; Window class, in windowKind field of window record

dialogKind      .EQU  2   ;dialog or alert window
userKind        .EQU  8   ;window created directly by the application

; Values returned by FindWindow

inDesk          .EQU  Ø   ;none of the following
inMenuBar       .EQU  1   ;in menu bar
inSysWindow     .EQU  2   ;in system window
inContent       .EQU  3   ;in content region (except grow, if active)
inDrag          .EQU  4   ;in drag region
```

```
inGrow          .EQU  5    ;in grow region (active window only)
inGoAway        .EQU  6    ;in go-away region (active window only)

; Axis constraints for DragGrayRgn

noConstraint    .EQU  Ø    ;no constraint
hAxisOnly       .EQU  1    ;horizontal axis only
vAxisOnly       .EQU  2    ;vertical axis only

; Messages to window definition function

wDrawMsg        .EQU  Ø    ;draw the window frame
wHitMsg         .EQU  1    ;tell what region mouse button was pressed in
wCalcRgnMsg     .EQU  2    ;calculate strucRgn and contRgn
wInitMsg        .EQU  3    ;do any additional window initialization
wDisposeMsg     .EQU  4    ;take any additional disposal actions
wGrowMsg        .EQU  5    ;draw window's grow image
wGIconMsg       .EQU  6    ;draw size box in content region

; Value returned by window definition function's hit routine

wNoHit          .EQU  Ø    ;none of the following
wInContent      .EQU  1    ;in content region (except grow, if active)
wInDrag         .EQU  2    ;in drag region
wInGrow         .EQU  3    ;in grow region (active window only)
wInGoAway       .EQU  4    ;in go-away region (active window only)
```

Window Record Data Structure

windowPort	Window's grafPort
windowKind	Window class
wVisible	Flag for whether window is visible
wHilited	Flag for whether window is highlighted
wGoAway	Flag for whether window has go-away region
structRgn	Handle to structure region of window
contRgn	Handle to content region of window
updateRgn	Handle to update region of window
windowDef	Handle to window definition function
wDataHandle	Data used by window definition function
wTitleHandle	Handle to window's title
wTitleWidth	Width of title in pixels
wControlList	Handle to window's control list
nextWindow	Pointer to next window in window list
windowPic	Picture handle for drawing window
wRefCon	Window's reference value
windowSize	Length of above structure

Special Macro Names

Routine name	Macro name
CalcVisBehind	_CalcVBehind
DisposeWindow	_DisposWindow

DragGrayRgn	_DragGrayRgn or, after setting the global variable dragPattern, _DragTheRgn

Variables

Name	Size	Contents
windowList	4 bytes	Pointer to first window in window list
saveUpdate	2 bytes	Flag for whether to generate update events
paintWhite	2 bytes	Flag for whether to paint window white before update event
curActivate	4 bytes	Pointer to window to receive activate event
curDeactive	4 bytes	Pointer to window to receive deactivate event
grayRgn	4 bytes	Handle to region to be drawn as desktop
deskPattern	8 bytes	Pattern with which desktop is to be painted
deskHook	4 bytes	Pointer to procedure for painting desktop or responding to clicks on desktop
wMgrPort	4 bytes	Pointer to Window Manager port
ghostWindow	4 bytes	Pointer to window never to be considered frontmost
dragHook	4 bytes	Pointer to procedure to execute during DragWindow
dragPattern	8 bytes	Pattern of dragged region's outline

GLOSSARY

activate event: An event generated by the Window Manager when a window changes from active to inactive or vice versa.

active window: The frontmost window on the desktop.

application window: A window created as the result of something done by the application, either directly or indirectly (as through the Dialog Manager).

content region: The area of a window that the application draws in.

control list: A list of all the controls associated with a given window.

desktop: The screen as a surface for doing work on the Macintosh.

document window: A standard Macintosh window for presenting a document.

drag region: A region in the window frame. Dragging inside this region moves the window to a new location and makes it the active window unless the Command key was down.

go-away region: A region in the window frame. Clicking inside this region of the active window makes the window close or disappear.

grow image: The image pulled around when dragging inside the grow region occurs; whatever is appropriate to show that the window's size will change.

grow region: A window region, usually within the content region, where dragging changes the size of an active window.

highlight: To display an object on the screen in a distinctive visual way, such as inverting it.

inactive window: Any window that isn't the frontmost window on the desktop.

invert: To highlight by changing white pixels to black and vice versa.

invisible window: A window that's not drawn in its plane on the desktop.

modal dialog: A dialog that requires the user to respond before doing any other work on the desktop.

modeless dialog: A dialog that allows the user to work elsewhere on the desktop before responding.

plane: The front-to-back position of a window on the desktop.

reference value: In a window record, a 32-bit field that the application program may store into and access for any purpose.

structure region: An entire window; its complete "structure".

system window: A window in which a desk accessory is displayed.

update event: An event generated by the Window Manager when the update region of a window is to be drawn.

update region: A window region consisting of all areas of the content region that have to be redrawn.

variation code: A number that distinguishes closely related types of windows and is passed as part of a window definition ID when a window is created.

visible window: A window that's drawn in its plane on the desktop (but may be completely overlapped by another window or object on the screen).

window: An object on the desktop that presents information, such as a document or a message.

window class: An indication of whether a window is a system window, a dialog or alert window, or a window created directly by the application.

window definition function: A function called by the Window Manager when it needs to perform certain type-dependent operations on a particular type of window, such as drawing the window frame.

window definition ID: A number passed to window-creation routines to indicate the type of window. It consists of the window definition function's resource ID and a variation code.

window frame: The structure region minus the content region.

window list: A list of all windows ordered according to their front-to-back positions on the desktop.

Window Manager port: A grafPort that has the entire screen as its portRect and is used by the Window Manager to draw window frames.

window record: The internal representation of a window, where the Window Manager stores all the information it needs for its operations on that window.

window template: A resource that contains information from which the Window Manager can create a window.

COMMENTS?

Macintosh User Education encourages your comments on this manual.

- What do you like or dislike about it?

- Were you able to find the information you needed?

- Was it complete and accurate?

- Do you have any suggestions for improvement?

Please send your comments to the author (indicated on the cover page) at 10460 Bandley Drive M/S 3-G, Cupertino CA 95014. Mark up a copy of the manual or note your remarks separately. (We'll return your marked-up copy if you like.)

Thanks for your help!

MACINTOSH USER EDUCATION

The Control Manager: A Programmmer's Guide /CMGR/CONTROLS

See Also: Macintosh User Interface Guidelines
The Memory Manager: A Programmer's Guide
The Resource Manager: A Programmer's Guide
QuickDraw: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Window Manager: A Programmer's Guide
Programming Macintosh Applications in Assembly Language

Modification History:			
	First Draft	Chris Espinosa	8/13/82
	Interim release	Chris Espinosa	9/7/82
	Second Draft (ROM 2.1)	Steve Chernicoff	3/16/83
	Third Draft (ROM 7)	Caroline Rose	5/3Ø/84

ABSTRACT

Controls are special objects on the Macintosh screen with which the user, using the mouse, can cause instant action with graphic results or change settings to modify a future action. The Macintosh Control Manager is the part of the User Interface Toolbox that enables applications to create and manipulate controls in a way that's consistent with the Macintosh User Interface Guidelines. This manual describes the Control Manager.

Summary of significant changes and additions since last draft:

- There's now a way to specify that you want the standard control definition functions to use the font associated with the control's window rather than the system font (page 8).
- You can now detect when the mouse button was pressed in an inactive control as opposed to not in any control; see HiliteControl, TestControl, and FindControl (page 18).
- The control definition function may itself contain an action procedure (pages 2Ø and 3Ø).
- Assembly-language notes were added where appropriate, and the summary was updated to include all assembly-language information.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Control Manager of the Macintosh User Interface Toolbox. *** Eventually it will become a chapter in the comprehensive Inside Macintosh manual. *** The Control Manager is the part of the Toolbox that deals with controls, such as buttons, check boxes, and scroll bars. Using it, your application can create, manipulate, and dispose of controls in a way that's consistent with the Macintosh User Interface Guidelines.

Like all Toolbox documentation, this manual assumes you're familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's Memory Manager. You should also be familiar with the following:

- Resources, as discussed in the Resource Manager manual.

- The basic concepts and structures behind QuickDraw, particularly rectangles, regions, and grafPorts. You don't need a detailed knowledge of QuickDraw, since implementing controls through the Control Manager doesn't require calling QuickDraw directly.

- The Toolbox Event Manager. The essence of a control is to respond to the user's actions with the mouse; your application finds out about those actions by calling the Event Manager.

- The Window Manager. Every control you create with the Control Manager "belongs" to some window. The Window Manager and Control Manager are designed to be used together, and their structure and operation are parallel in many ways.

(note)

Except for scroll bars, most controls appear only in dialog or alert boxes. To learn how to implement dialogs and alerts in your application, you'll have to read the Dialog Manager manual.

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly-language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it.

The manual begins with an introduction to the Control Manager and what you can do with it. It then discusses some basic concepts about controls: the relationship between controls and windows; the relationship between controls and resources; and how controls and their various parts are identified. Following this is a discussion of control records, where the Control Manager keeps all the information it needs about a control.

Next, a section on using the Control Manager introduces its routines and tells how they fit into the flow of your application program. This is followed by detailed descriptions of all Control Manager procedures

and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that will not interest all readers: special information is provided for programmers who want to define their own controls, and the exact formats of resources related to controls are described.

Finally, there's a summary of the Control Manager, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE CONTROL MANAGER

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

- create and 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 size, 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 may be of various types (see Figure 1), 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.

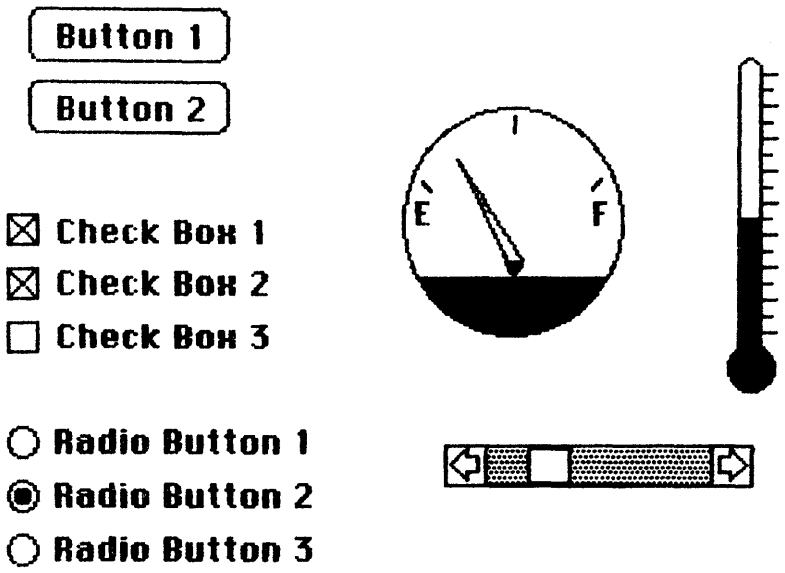


Figure 1. Controls

Certain standard types of controls are predefined for you. Your application can easily create and use controls of these standard types, and can also define its own "custom" control types. Among the standard control types are the following:

- Buttons cause an immediate or continuous action when clicked or pressed with the mouse. They appear on the screen as rounded-corner 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 alongside it; 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.

- Radio buttons also retain and display an on-or-off setting. They're organized into groups, with the property that only one button in the group can be on at a time: clicking any button on turns off all the others in the group, like the buttons on a car radio. Radio buttons are used to offer a choice among several alternatives. On the screen, they look like round check boxes; the radio button that's on is filled with a small black circle instead of an "X".

(note)

The Control Manager doesn't know how radio buttons are grouped, and doesn't automatically turn one off when the user clicks another one on: it's up to your program to handle this.

Another important category of controls is dials. These display a quantitative setting or value, typically in some pseudoanalog 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).

One type of dial is predefined for you: the standard Macintosh scroll bars. Figure 2 shows the five parts of a scroll bar and the terms used by the Control Manager (and this manual) to refer to them. Notice that the part of the scroll bar that Macintosh users know as the "scroll box" is called the "thumb" here. Also, for simplicity, the terms "up" and "down" are used even when referring to horizontal scroll bars (in which case "up" really means "left" and "down" means "right").

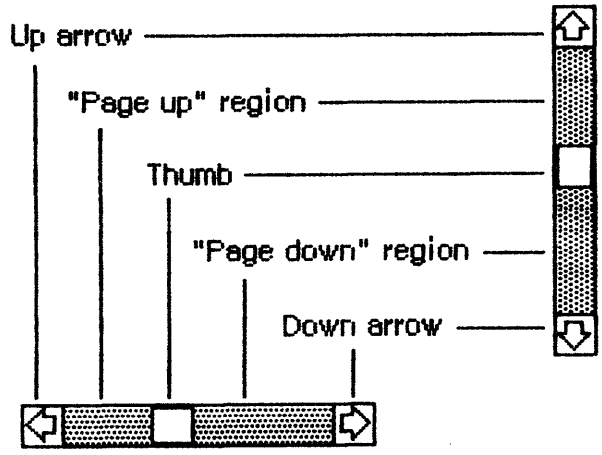


Figure 2. Parts of a Scroll Bar

The up and down arrows scroll the window's contents a line at a time. The two paging regions scroll a "page" (windowful) at a time. The thumb can be dragged to any position in the scroll bar, to scroll to a corresponding position within the document. Although they may seem to behave like individual controls, these are all parts of a single control, the scroll bar type of dial. You can define other dials of any shape or complexity for yourself if your application needs them.

When clicked or pressed, a control is usually highlighted (see Figure 3). Standard button controls are inverted, but some control types may use other forms of highlighting, such as making the outline heavier. It's also possible for just a part of a control to be highlighted: for example, when the user presses the mouse button inside a scroll arrow or the thumb in a scroll bar, the arrow or thumb (not the whole scroll bar) becomes highlighted until the button is released.

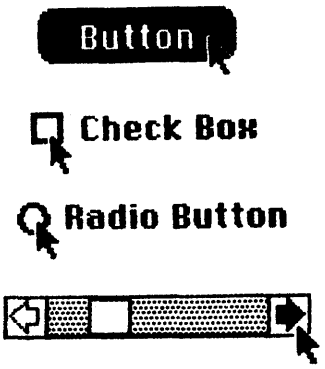


Figure 3. Highlighted Controls

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 remains visible, but is highlighted in some special way, depending on its control type (see Figure 4). For example, the title of an inactive button, check box, or radio button is dimmed (drawn in gray rather than black).

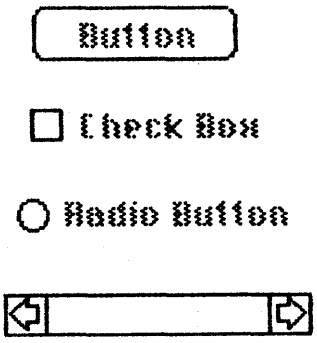


Figure 4. Inactive Controls

CONTROLS AND WINDOWS

Every control "belongs" to a particular window: When displayed, the control appears within that window's content region; when manipulated with the mouse, it acts on that window. All coordinates pertaining to the control (such as those describing its location) are given in its window's local coordinate system.

(warning)
In order for the Control Manager to draw a control properly, the control's window must have the top left corner of its grafPort's portRect at coordinates (Ø,Ø).

> If you change a window's local coordinate system for any reason (with the QuickDraw procedure SetOrigin), be sure to change it back--so that the top left corner is again at (Ø,Ø)--before drawing any of its controls. Since almost all of the Control Manager routines can (at least potentially) redraw a control, the safest policy is simply to change the coordinate system back before calling **any** Control Manager routine.

Normally you'll include buttons and check boxes in dialog or alert windows only. You create such windows with the Dialog Manager, and the Dialog Manager takes care of drawing the controls and letting you know whether the user clicked one of them. See the Dialog Manager manual for details.

CONTROLS AND RESOURCES

The relationship between controls and resources is analogous to the relationship between windows and resources: just as there are window definition functions and window templates, there are control definition functions and control templates.

Each type of control has a control definition function that determines how controls of that type look and behave. The Control Manager calls the control definition function whenever it needs to perform a type-dependent action, such as drawing the control on the screen. Control definition functions are stored as resources and accessed through the Resource Manager. The system resource file includes definition functions for the standard control types (buttons, check boxes, radio buttons, and scroll bars). If you want to define your own, nonstandard control types, you'll have to write your own definition functions for them, as described later in the section "Defining Your Own Controls".

When you create a control, you specify its type with a control definition ID, which tells the Control Manager the resource ID of the definition function for that control type. The Control Manager provides the following predefined constants for the definition IDs of the standard control types:

```
CONST pushButProc   = Ø;   {simple button}
      checkBoxProc  = 1;   {check box}
      radioButProc  = 2;   {radio button}
      scrollBarProc = 16;  {scroll bar}
```

The title of a button, check box, or radio button normally appears in the system font, but you can add the following constant to the definition ID to specify that you instead want to use the font currently associated with the window's grafPort:

```
CONST useWFont = 8;  {use window's font}
```

To create a control, the Control Manager needs to know not only the control definition ID but also other information specific to this control, such as its title (if any), the window it belongs to, and its location within the window. You can supply all the needed information in individual parameters to a Control Manager routine, or you can store it in a control template in a resource file and just pass the template's resource ID. Using templates is highly recommended, since it simplifies the process of creating controls and isolates the control descriptions from your application's code.

(note)

You can create control templates and store them in resource files with the aid of the Resource Editor *** eventually (for now, the Resource Compiler) ***. The Resource Editor relieves you of having to know the exact format of a control template, but if you're interested, you'll find details in the section "Formats of Resources for Controls".

PART CODES

Some controls, such as buttons, are simple and straightforward. Others can be complex objects with many parts: for example, a scroll bar has two scroll arrows, two paging regions, and a thumb (see Figure 2). 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 one as a result.

A part code is an integer between 1 and 253 that stands for a particular part of a control. Each type of control has its own set of part codes, assigned by the control definition function for that type. A simple control such as a button or check box might have just one "part" that encompasses the entire control; a more complex control such as a scroll bar can have as many parts as are needed to define how the control operates. Some of the Control Manager routines need to give special treatment to the indicator of a dial (such as the thumb of a scroll bar). To allow the Control Manager to recognize such indicators, they always have part codes greater than 128.

(note)

The values 254 and 255 are not used for part codes because to some Control Manager routines they represent the entire control in its inactive state.

The part codes for the standard control types are as follows:

```
CONST inButton     = 1Ø;   {simple button}
      inCheckBox   = 11;   {check box or radio button}
      inUpButton   = 2Ø;   {up arrow of a scroll bar}
      inDownButton = 21;   {down arrow of a scroll bar}
      inPageUp     = 22;   {"page up" region of a scroll bar}
      inPageDown   = 23;   {"page down" region of a scroll bar}
      inThumb      = 129;  {thumb of a scroll bar}
```

Notice that inCheckBox applies to both check boxes and radio buttons.

(note)
The part code 128 is reserved for special use by the Control Manager and so should not be used for parts of your controls.

CONTROL RECORDS

Every control is represented internally by a control record containing all pertinent information about that control. The control record contains the following:

- A pointer to the window the control belongs to.
- A handle to the next control in the window's control list.
- A handle to the control definition function.
- The control's title, if any.
- A rectangle that completely encloses the control, which determines the control's size and location within its window. The entire control, including the title of a check box or radio button, is drawn inside this rectangle.
- An indication of whether the control is currently active and how it's to be highlighted.
- The current setting of the control (if this type of control retains a setting) and the minimum and maximum values the setting can assume. For check boxes and radio buttons, a setting of Ø means the control is off and 1 means it's on.

The control record also contains an indication of whether the control is currently visible or invisible. These terms refer only to whether the control is drawn in its window, not to whether you can see it on the screen. A control may be "visible" and still not appear on the screen, because it's obscured by overlapping windows or other objects.

There's a field in the control record for a pointer to the control's default action procedure. An action procedure defines some action to be performed repeatedly for as long as the user holds down the mouse button inside the control. The default action procedure may be used by

the Control Manager function TrackControl if you call it without passing a pointer to an action procedure; this is discussed in detail in the description of TrackControl in the "Control Manager Routines" section.

Finally, the control record includes a 32-bit reference value field, which is reserved for use by your application. You specify an initial reference value when you create a control, and can then read or change the reference value whenever you wish.

The data type for a control record is called ControlRecord. A control record is referred to by a handle:

```
TYPE ControlPtr    = ^ControlRecord;
     ControlHandle = ^ControlPtr;
```

The Control Manager functions for creating a control return a handle to a newly allocated control record; thereafter, your program should normally refer to the control by this handle. Most of the Control Manager routines expect a control handle as their first parameter.

You can store into and access most of a control record's fields with Control Manager routines, so normally you don't have to know the exact field names. However, if you want more information about the exact structure of a control record--if you're defining your own control types, for instance--it's given below.

The ControlRecord Data Type

The type ControlRecord is defined as follows:

```
TYPE ControlRecord =
            RECORD
              nextControl:   ControlHandle; {next control}
              contrlOwner:   WindowPtr;     {control's window}
              contrlRect:    Rect;          {enclosing rectangle}
              contrlVis:     BOOLEAN;       {TRUE if visible}
              contrlHilite:  BOOLEAN;       {highlight state}
              contrlValue:   INTEGER;       {current setting}
              contrlMin:     INTEGER;       {minimum setting}
              contrlMax:     INTEGER;       {maximum setting}
              contrlDefProc: Handle;        {control definition function}
              contrlData:    Handle;        {data used by contrlDefProc}
              contrlAction:  ProcPtr;       {default action procedure}
              contrlRfCon:   LongInt;       {control's reference value}
              contrlTitle:   Str255         {control's title}
            END;
```

NextControl is 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 controlList field of the window record and chained together through the nextControl fields of the individual control records. The end of the list is marked by a NIL

value; as new controls are created, they are added to the beginning of the list.

ContrlOwner is a pointer to the window that this control belongs to.

ContrlRect is the rectangle that completely encloses the control, in the local coordinates of the control's window.

When contrlVis is TRUE, the control is currently visible.

(note)

The Control Manager sets the contrlVis field FALSE by storing 255 in it rather than 1. This may cause problems in Lisa Pascal; to be safe, you should check for the truth or falsity of this flag by comparing ORD of the flag to Ø.

ContrlHilite is an integer between Ø and 255 that specifies whether and how the control is to be highlighted. It's declared as BOOLEAN so that Pascal will put the value in a byte; if declared as Byte, it would be put it in a word because of Pascal's packing conventions. Storing directly into the contrlHilite field limits it to a Boolean value, so you'll probably instead want to use the Control Manager routine that sets it (HiliteControl). See the description of HiliteControl in the "Control Manager Routines" section for information about the meaning of this field's value.

ContrlValue is the control's current setting. For check boxes and radio buttons, Ø means the control is off and 1 means it's on. For dials, the fields contrlMin and contrlMax define the range of possible settings; contrlValue may take on any value within that range. Other (custom) control types can use these three fields as they see fit.

ContrlDefProc is a handle to the control definition function for this type of control. When you create a control, you identify its type with a control definition ID, which is converted into a handle to the control definition function and stored in the contrlDefProc field. Thereafter, the Control Manager uses this handle to access the definition function; you should never need to refer to this field directly.

(note)

The high-order byte of the contrlDefProc field contains some additional information that the Control Manager gets from the control definition ID; for details, see the section "Defining Your Own Controls". Also note that if you write your own control definition function, you can include it as part of your application's code and just store a handle to it in the contrlDefProc field.

ContrlData is reserved for use by the control definition function, typically to hold additional information specific to a particular control type. For example, the standard definition function for scroll bars uses this field for a handle to the region containing the scroll

bar's thumb. If no more than four bytes of additional information are needed, the definition function can store the information directly in the contrlData field rather than use a handle.

ContrlAction is a pointer to the control's default action procedure, if any. The Control Manager function TrackControl may call this procedure to respond to the user's dragging the mouse inside the control.

ContrlRfCon is the control's reference value field, which the application may store into and access for any purpose.

ContrlTitle is the control's title, if any.

Assembly-language note: The global constant contrlSize equals the length in bytes of a control record less its contrlTitle field.

USING THE CONTROL MANAGER

This section discusses how the Control Manager routines fit into the general flow of an application program and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

(note)

For controls in dialogs or alerts, the Dialog Manager makes some of the basic Control Manager calls for you; see the Dialog Manager manual for more information.

To use the Control Manager, you must have previously called InitGraf to initialize QuickDraw, InitFonts to initialize the Font Manager, and InitWindows to initialize the Window Manager.

Where appropriate in your program, use NewControl or GetNewControl to create any controls you need. NewControl takes descriptive information about the new control from its parameters; GetNewControl gets the information from a control template in a resource file. When you no longer need a control, call DisposeControl to remove it from its window's control list and release the memory it occupies. To dispose of all of a given window's controls at once, use KillControls.

(note)

The Window Manager procedures DisposeWindow and CloseWindow automatically dispose of all the controls associated with the given window.

When the Toolbox Event Manager function GetNextEvent reports that an update event has occurred for a window, the application should call

DrawControls to redraw the window's controls as part of the process of updating the window.

After receiving a mouse-down event from GetNextEvent, do the following:

1. First call FindWindow to determine which part of which window the mouse button was pressed in.

2. If it was in the content region of the active window, next call FindControl for that window to find out whether it was in an active control, and if so, in which part of which control.

3. Finally, take whatever action is appropriate when the user presses the mouse button in that part of the control, using routines such as TrackControl (to perform some action repeatedly for as long as the mouse button is down, or to allow the user to drag the control's indicator with the mouse), DragControl (to pull an outline of the control across the screen and move the control to a new location), and HiliteControl (to change the way the control is highlighted).

For the standard control types, step 3 involves calling TrackControl. TrackControl handles the highlighting of the control and determines whether the mouse is still in the control when the mouse button is released. It also handles the dragging of the thumb in a scroll bar and, via your action procedure, the response to presses or clicks in the other parts of a scroll bar. When TrackControl returns the part code for a button, check box, or radio button, the application must do whatever is appropriate as a response to a click of that control. When TrackControl returns the part code for the thumb of a scroll bar, the application must scroll to the corresponding relative position in the document.

The application's exact response to mouse activity in a control that retains a setting will depend on the current setting of the control, which is available from the GetCtlValue function. For controls whose values can be set by the user, the SetCtlValue procedure may be called to change the control's setting and redraw the control accordingly. You'll call SetCtlValue, for example, when a check box or radio button is clicked, to change the setting and draw or clear the mark inside the control.

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

Whenever necessary, you can read various attributes of a control with GetCTitle, GetCtlMin, GetCtlMax, GetCRefCon, or GetCtlAction; you can change them with SetCTitle, SetCtlMin, SetCtlMax, SetCRefCon, or

SetCtlAction.

CONTROL MANAGER ROUTINES

This section describes all the Control Manager procedures and functions. They're presented in their Pascal form; for information on using them from assembly language, see Programming Macintosh Applications in Assembly Language.

Initialization and Allocation

```
FUNCTION NewControl (theWindow: WindowPtr; boundsRect: Rect; title:
            Str255; visible: BOOLEAN; value: INTEGER; min,max: INTEGER;
            procID: INTEGER; refCon: LongInt) : ControlHandle;
```

NewControl creates a control, adds it to the beginning of theWindow's control list, and returns a handle to the new control. The values passed as parameters are stored in the corresponding fields of the control record, as described below. The field that determines highlighting is set to Ø (no highlighting) and the pointer to the default action procedure is set to NIL (none).

(note)
The control definition function may do additional initialization, including changing any of the fields of the control record. The only standard control for which additional initialization is done is the scroll bar; its control definition function allocates space for a region to hold the thumb and stores the region handle in the contrlData field of the control record.

TheWindow is the window the new control will belong to. All coordinates pertaining to the control will be interpreted in this window's local coordinate system.

BoundsRect, given in theWindow's local coordinates, is the rectangle that encloses the control and thus determines its size and location. Note the following about the enclosing rectangle for the standard controls:

- Simple buttons are drawn to fit the rectangle exactly. (The control definition function calls the QuickDraw procedure FrameRoundRect.) To allow for the tallest characters in the system font, there should be at least a 2Ø-point difference between the top and bottom coordinates of the rectangle.

- For check boxes and radio buttons, there should be at least a 16-point difference between the top and bottom coordinates.

- By convention, scroll bars are 16 pixels wide, so there should be a 16-point difference between the left and right (or top and bottom) coordinates. If there isn't, the scroll bar will be scaled to fit the rectangle.

Title is the control's title, if any (if none, you can just pass the empty string as the title). Be sure the title will fit in the control's enclosing rectangle; if it won't, it will be truncated on the right for check boxes and radio buttons, or centered and truncated on both ends for simple buttons.

If the visible parameter is TRUE, NewControl draws the control.

(note)
It does **not** use the standard window updating mechanism, but instead draws the control immediately in the window.

The min and max parameters define the control's range of possible settings; the value parameter gives the initial setting. For controls that don't retain a setting, such as buttons, the values you supply for these parameters will be stored in the control record but will never be used. So it doesn't matter what values you give for those controls--Ø for all three parameters will do. For controls that just retain an on-or-off setting, such as check boxes or radio buttons, min should be Ø (meaning the control is off) and max should be 1 (meaning it's on). For dials, you can specify whatever values are appropriate for min, max, and value.

ProcID is the control definition ID, which leads to the control definition function for this type of control. The control definition IDs for the standard control types are listed above under "Controls and Resources". Control definition IDs for custom control types are discussed later under "Defining Your Own Controls".

RefCon is the control's reference value, set and used only by your application.

```
FUNCTION GetNewControl (controlID: INTEGER; theWindow: WindowPtr) :
            ControlHandle;
```

GetNewControl creates a control from a control template stored in a resource file, adds it to the beginning of theWindow's control list, and returns a handle to the new control. ControlID is the resource ID of the template. GetNewControl works exactly the same as NewControl (above), except that it gets the initial values for the new control's fields from the specified control template instead of accepting them as parameters.

```
PROCEDURE DisposeControl (theControl: ControlHandle);
```

DisposeControl removes theControl from the screen, deletes it from its window's control list, and releases the memory occupied by the control

record and all data structures associated with the control.

Assembly-language note: The macro you invoke to call DisposeControl from assembly language is named _DisposControl.

PROCEDURE KillControls (theWindow: WindowPtr);

KillControls disposes of all controls associated with theWindow by calling DisposeControl (above) for each.

Control Display

These procedures affect the appearance of a control but not its size or location.

PROCEDURE SetCTitle (theControl: ControlHandle; title: Str255);

SetCTitle sets theControl's title to the given string and redraws the control.

PROCEDURE GetCTitle (theControl: ControlHandle; VAR title: Str255);

GetCTitle returns theControl's title as the value of the title parameter.

PROCEDURE HideControl (theControl: ControlHandle);

HideControl makes theControl invisible. It fills the region the control occupies within its window with the background pattern of the window's grafPort. It also adds the control's enclosing rectangle to the window's update region, so that anything else that was previously obscured by the control will reappear on the screen. If the control is already invisible, HideControl has no effect.

PROCEDURE ShowControl (theControl: ControlHandle);

ShowControl makes theControl 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 already visible, ShowControl has no effect.

```
PROCEDURE DrawControls (theWindow: WindowPtr);
```

DrawControls draws all controls currently visible in theWindow. The controls are drawn in reverse order of creation; thus in case of overlap the earliest-created controls appear frontmost in the window.

(note)

Window Manager routines such as SelectWindow, ShowWindow, and BringToFront do not automatically call DrawControls to display the window's controls. They just add the appropriate regions to the window's update region, generating an update event. Your program should always call DrawControls explicitly upon receiving an update event for a window that contains controls.

```
PROCEDURE HiliteControl (theControl: ControlHandle; hiliteState:
            INTEGER);
```

HiliteControl changes the way theControl is highlighted. HiliteState is an integer between Ø and 255:

- A value of Ø means no highlighting.
- A value between 1 and 253 is interpreted as a part code designating the part of the control to be highlighted.
- A value of 254 or 255 means that the control is to be made inactive and highlighted accordingly. Usually you'll want to use 254, because it enables you to detect when the mouse button was pressed in the inactive control as opposed to not in any control; for more information, see FindControl under "Mouse Location" below.

HiliteControl calls the control definition function to redraw the control with its new highlighting.

Mouse Location

```
FUNCTION TestControl (theControl: ControlHandle; thePoint: Point) :
            INTEGER;
```

If theControl is visible and active, TestControl tests which part of the control contains thePoint (in the local coordinates of the control's window); it returns the corresponding part code, or Ø if the point is outside the control. If the control is visible and inactive with 254 highlighting, TestControl returns 254. If the control is invisible, or inactive with 255 highlighting, TestControl returns Ø.

```
FUNCTION FindControl (thePoint: Point; theWindow: WindowPtr; VAR
          whichControl: ControlHandle) : INTEGER;
```

When the Window Manager function FindWindow reports that the mouse button was pressed in the content region of a window, and the window contains controls, the application should call FindControl with theWindow equal to the window pointer and thePoint equal to the point where the mouse button was pressed (in the window's local coordinates). FindControl tells which of the window's controls, if any, the mouse button was pressed in:

- If it was pressed in a visible, active control, FindControl sets the whichControl parameter to the control handle and returns a part code identifying the part of the control that it was pressed in.

- If it was pressed in a visible, inactive control with 254 highlighting, FindControl sets whichControl to the control handle and returns 254 as its result.

- If it was pressed in an invisible control, an inactive control with 255 highlighting, or not in any control, FindControl sets whichControl to NIL and returns Ø as its result.

(warning)
Notice that FindControl expects the mouse point in the window's local coordinates, whereas FindWindow expects it in global coordinates. Always be sure to convert the point to local coordinates with the QuickDraw procedure GlobalToLocal before calling FindControl.

(note)
FindControl also returns NIL for whichControl and Ø as its result if the window is invisible or doesn't contain the given point. In these cases, however, FindWindow wouldn't have returned this window in the first place, so the situation should never arise.

```
FUNCTION TrackControl (theControl: ControlHandle; startPt: Point;
          actionProc: ProcPtr) : INTEGER;
```

When the mouse button is pressed in a visible, active control, the application should call TrackControl with theControl equal to the control handle and startPt equal to the point where the mouse button was pressed (in the local coordinates of the control's window). TrackControl follows the movements of the mouse and responds in whatever way is appropriate 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 does the highlighting, and undoes 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, or with Ø if not (in which case the application

should do nothing).

If the mouse button was pressed in an indicator, TrackControl drags a gray outline of it to follow the mouse (by calling the Window Manager utility function DragGrayRgn). When the mouse button is released, TrackControl calls the control definition function to reposition the control's 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; for example, if the minimum and maximum settings are Ø and 1Ø, and the thumb is in the middle of the scroll bar, 5 is stored as the current setting. The application must then scroll to the corresponding relative position in the document.

TrackControl may take additional actions beyond highlighting the control or dragging the indicator, depending on the value passed in the actionProc parameter, as described below. Here you'll learn what to pass for the standard control types; for a custom control, what you pass will depend on how the control is defined.

- If actionProc is NIL, TrackControl performs no additional actions. This is appropriate for simple buttons, check boxes, radio buttons, and the thumb of a scroll bar.

- ActionProc 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. (See below for details.)

- If actionProc is POINTER(-1), TrackControl looks in the control record for a pointer to the control's default action procedure. If that field of the control record contains a procedure pointer, TrackControl uses the action procedure it points to; if the field contains POINTER(-1), TrackControl calls the control definition function to perform the necessary action. (If the field contains NIL, TrackControl does nothing.)

The action procedure in the control definition function is described in the section "Defining Your Own Controls". The following paragraphs describe only the action procedure whose pointer is passed in the actionProc parameter or stored in the control record.

If the mouse button was pressed in an indicator, the action procedure (if any) should have no parameters. This procedure must allow for the fact that the mouse may not be inside the original control part.

If the mouse button was pressed in a control part other than an indicator, the action procedure should be of the form

```
PROCEDURE MyAction (theControl: ControlHandle; partCode: INTEGER);
```

In this case, TrackControl passes the control handle and the part code to the action procedure. (It passes Ø 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 in 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 SetCtlValue to change the control's setting and redraw the thumb.

(warning)

Since it has a different number of parameters depending on whether the mouse button was pressed in an indicator or elsewhere, the action procedure you pass to TrackControl (or whose pointer you store in the control record) can be set up for only one case or the other. If you store a pointer to a default action procedure in a control record, be sure it will be used only when appropriate for that type of action procedure. The only way to specify actions in response to all mouse-down events in a control, regardless of whether they're in an indicator, is via the control definition function.

Control Movement and Sizing

```
PROCEDURE MoveControl (theControl: ControlHandle; h,v: INTEGER);
```

MoveControl moves theControl to a new location within its window. The top left corner of the control's enclosing rectangle is moved to the horizontal and vertical coordinates h and v (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.

```
PROCEDURE DragControl (theControl: ControlHandle; startPt: Point;
          limitRect,slopRect: Rect; axis: INTEGER);
```

Called with the mouse button down inside theControl, DragControl pulls a gray 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 MoveControl to move the control to the location to which it was dragged.

(note)

Before beginning to follow the mouse, DragControl calls the control definition function to allow it to do its own "custom dragging" if it chooses. If the definition function doesn't choose to do any custom dragging, DragControl uses the default method of dragging described here.

DragControl calls the Window Manager utility function DragGrayRgn and then moves the control accordingly. The startPt, limitRect, slopRect, and axis parameters have the same meaning as for DragGrayRgn. These parameters are reviewed briefly below; see the description of DragGrayRgn in the Window Manager manual for more details.

- StartPt parameter is assumed to be the point where the mouse button was originally pressed, in the local coordinates of the control's window.

- LimitRect limits the travel of the control's outline, and should normally coincide with or be contained within the window's content region.

- SlopRect allows the user some "slop" in moving the mouse; it should completely enclose limitRect.

- The axis parameter allows you to constrain the control's motion to only one axis. It has one of the following values:

```
CONST noConstraint = Ø;  {no constraint}
      hAxisOnly    = 1;  {horizontal axis only}
      vAxisOnly    = 2;  {vertical axis only}
```

```
PROCEDURE SizeControl (theControl: ControlHandle; w,h: INTEGER);
```

SizeControl changes the size of theControl's enclosing rectangle. The bottom right corner of the rectangle is adjusted to set the rectangle's width and height to the number of pixels specified by w and h; the position of the top left corner is not changed. If the control is currently visible, it's hidden and then redrawn in its new size.

Control Setting and Range

```
PROCEDURE SetCtlValue (theControl: ControlHandle; theValue: INTEGER);
```

SetCtlValue sets theControl's current setting to theValue and redraws the control to reflect the new setting. For check boxes and radio buttons, the value 1 fills the control with the appropriate mark, and Ø clears it. For scroll bars, SetCtlValue redraws the thumb where appropriate.

If the specified value is out of range, it's forced to the nearest endpoint of the current range (that is, if theValue is less than the minimum setting, SetCtlValue sets the current setting to the minimum; if theValue is greater than the maximum setting, it sets the current setting to the maximum).

```
FUNCTION GetCtlValue (theControl: ControlHandle) : INTEGER;
```

GetCtlValue returns theControl's current setting.

```
PROCEDURE SetCtlMin (theControl: ControlHandle; minValue: INTEGER);
```

SetCtlMin sets theControl's minimum setting to minValue and redraws the control to reflect the new range. If the control's current setting is less than minValue, the setting is changed to the new minimum.

Assembly-language note: The macro you invoke to call SetCtlMin from assembly language is named _SetMinCtl.

```
FUNCTION GetCtlMin (theControl: ControlHandle) : INTEGER;
```

GetCtlMin returns theControl's minimum setting.

Assembly-language note: The macro you invoke to call GetCtlMin from assembly language is named _GetMinCtl.

```
PROCEDURE SetCtlMax (theControl: ControlHandle; maxValue: INTEGER);
```

SetCtlMax sets theControl's maximum setting to maxValue and redraws the control to reflect the new range. If maxValue is less than the control's current setting, the setting is changed to the new maximum.

Assembly-language note: The macro you invoke to call SetCtlMax from assembly language is named _SetMaxCtl.

```
FUNCTION GetCtlMax (theControl: ControlHandle) : INTEGER;
```

GetCtlMax returns theControl's maximum setting.

Assembly-language note: The macro you invoke to call GetCtlMax from assembly language is named _GetMaxCtl.

Miscellaneous Utilities

```
PROCEDURE SetCRefCon (theControl: ControlHandle; data: LongInt);
```

SetCRefCon sets theControl's reference value to the given data.

```
FUNCTION GetCRefCon (theControl: ControlHandle) : LongInt;
```

GetCRefCon returns theControl's current reference value.

```
PROCEDURE SetCtlAction (theControl: ControlHandle; actionProc:
            ProcPtr);
```

SetCtlAction sets theControl's default action procedure to actionProc.

```
FUNCTION GetCtlAction (theControl: ControlHandle) : ProcPtr;
```

GetCtlAction returns a pointer to theControl's default action procedure, if any. (It returns whatever is in that field of the control record.)

DEFINING YOUR OWN CONTROLS

In addition to the standard, built-in control types (buttons, check boxes, radio buttons, and scroll bars), the Control Manager allows you to define "custom" control types of your own. Maybe you need a three-way selector switch, a memory-space indicator that looks like a thermometer, or a thruster control for a spacecraft simulator--whatever your application calls for. Controls and their indicators may occupy regions of any shape, in the full generality permitted by QuickDraw.

To define your own type of control, you write a control definition function and (usually) store it in a resource file. When you create a control, you provide a control definition ID, which leads to the control definition function. The control definition ID is an integer that contains the resource ID of the control definition function in its upper 12 bits and a variation code in its lower four bits. Thus, for a given resource ID and variation code, the control definition ID is:

```
16 * resource ID + variation code
```

For example, buttons, check boxes, and radio buttons all use the standard definition function whose resource ID is Ø, but they have variation codes of Ø, 1, and 2, respectively.

The Control Manager calls the Resource Manager to access the control definition function with the given resource ID. The Resource Manager reads the control definition function into memory and returns a handle to it. The Control Manager stores this handle in the contrlDefProc field of the control record, along with the variation code in the high-order byte of the field. Later, when it needs to perform a type-dependent action on the control, it calls the control definition function and passes it the variation code as a parameter. Figure 5 illustrates this process.

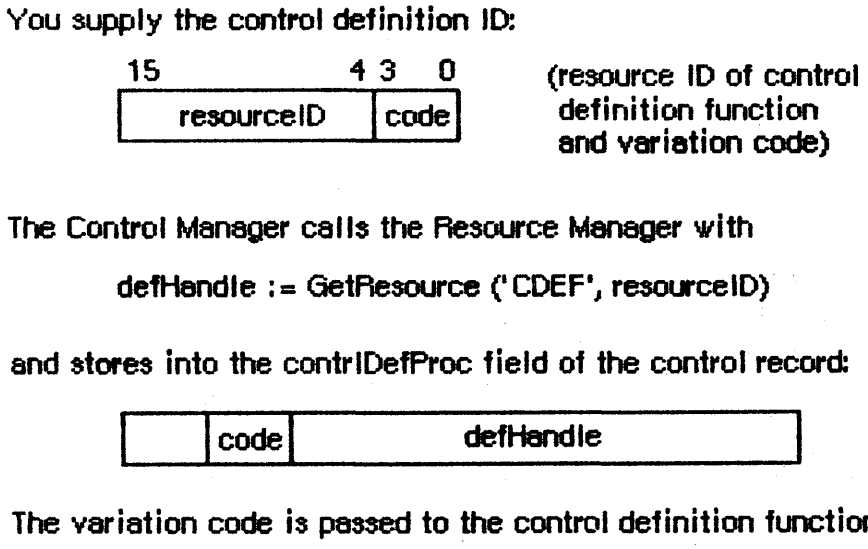


Figure 5. Control Definition Handling

Keep in mind that the calls your application makes to use a control depend heavily on the control definition function. What you pass to the TrackControl function, for example, depends on whether the definition function contains an action procedure for the control. Just as you need to know how to call TrackControl for the standard controls, each custom control type will have a particular calling protocol that must be followed for the control to work properly.

(note)

You may find it more convenient to include the control definition function with the code of your program instead of storing it as a separate resource. If you do this, you should supply the control definition ID of any standard control type when you create the control, and specify that the control initially be invisible. Once the control is created, you can replace the contents of the contrlDefProc field with a handle to the actual control definition function (along with a variation code, if needed, in the high-order byte of the field). You can then call ShowControl to make the control visible.

The Control Definition Function

The control definition function may be written in Pascal or assembly language; the only requirement is that its entry point must be at the beginning. You can give your control definition function any name you like. Here's how you would declare one named MyControl:

```
FUNCTION MyControl (varCode: INTEGER; theControl: ControlHandle;
                    message: INTEGER; param: LongInt) : LongInt;
```

VarCode is the variation code, as described above.

TheControl is a handle to the control that the operation will affect.

The message parameter identifies the desired operation. It has one of the following values:

```
CONST drawCntl  = 0;  {draw the control (or control part)}
      testCntl  = 1;  {test where mouse button was pressed}
      calcCRgns = 2;  {calculate control's region (or indicator's)}
      initCntl  = 3;  {do any additional control initialization}
      dispCntl  = 4;  {take any additional disposal actions}
      posCntl   = 5;  {reposition control's indicator and update it}
      thumbCntl = 6;  {calculate parameters for dragging indicator}
      dragCntl  = 7;  {drag control (or its indicator)}
      autoTrack = 8;  {execute control's action procedure}
```

As described below in the discussions of the routines that perform these operations, the value passed for param, the last parameter of the control definition function, depends on the operation. Where it's not mentioned below, this parameter is ignored. Similarly, the control definition function is expected to return a function result only where indicated; in other cases, the function should return 0.

(note)

"Routine" here does not necessarily mean a procedure or function. While it's a good idea to set these up as subprograms inside the control definition function, you're not required to do so.

The Draw Routine

The message drawCntl asks the control definition function to draw all or part of the control within its enclosing rectangle. The value of param is a part code specifying which part of the control to draw, or 0 for the entire control. If the control is invisible (that is, if its contrlVis field is FALSE), there's nothing to do; if it's visible, the definition function should draw it (or the requested part), taking into account the current values of its contrlHilite and contrlValue fields. The control may be either scaled or clipped to the enclosing rectangle.

If param 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. There's a special case, though, in which the draw routine has to allow for the fact that the indicator may have moved: this happens when the Control Manager procedures SetCtlValue, SetCtlMin, and SetCtlMax call the control definition function to redraw the indicator after changing the control setting. Since they have no way of knowing what part code you chose for your indicator, they all pass 128 (the special reserved part code) to mean the indicator. The draw routine must detect this part code as a special case, and remove the indicator from its former location before drawing it.

(note)

If your control has more than one indicator, 128 should be interpreted to mean all indicators.

The Test Routine

The Control Manager function FindControl sends the message testCntl to the control definition function 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 param, in the local coordinates of the control's window; the vertical coordinate is in the high-order word of the LongInt and the horizontal coordinate is in the low-order word. The control definition function should return the part code for the part of the control that contains the point; it should return 254 if the control is inactive with 254 highlighting, or Ø if the point is outside the control or if the control is inactive with 255 highlighting.

The Routine to Calculate Regions

The control definition function should respond to the message calcCRgns by calculating the region the control occupies within its window. Param is a QuickDraw region handle; the definition function should update this region to the region occupied by the control, expressed in the local coordinate system of its window.

If the high-order bit of param is set, the region requested is that of the control's indicator rather than the control as a whole. The definition function should clear the high **byte** (not just the high bit) of the region handle before attempting to update the region.

The Initialize Routine

After initializing fields as appropriate when creating a new control, the Control Manager sends the message initCntl to the control definition function. This gives the definition function a chance to perform any type-specific initialization it may require. For example, if you implement the control's action procedure in its control definition function, you'll set up the initialize routine to store POINTER(-1) in the contrlAction field; TrackControl calls for this control would pass POINTER(-1) in the actionProc parameter.

The control definition function for scroll bars allocates space for a region to hold the scroll bar's thumb and stores the region handle in the contrlData field of the new control record. The initialize routine for standard buttons, check boxes, and radio buttons does nothing.

The Dispose Routine

The Control Manager's DisposeControl procedure sends the message dispCntl to the control definition function, telling it to carry out any additional actions required when disposing of the control. For example, the standard definition function for scroll bars releases the space occupied by the thumb region, whose handle is kept in the control's contrlData field. The dispose routine for standard buttons, check boxes, and radio buttons does nothing.

The Drag Routine

The message dragCntl asks the control definition function to drag the control or its indicator around on the screen to follow the mouse until the user releases the mouse button. Param specifies whether to drag the indicator or the whole control: Ø means drag the whole control, while a nonzero value means just drag the indicator.

The control definition function need not implement any form of "custom dragging"; if it returns a result of Ø, the Control Manager will use its own default method of dragging (calling DragControl to drag the control or the Window Manager function DragGrayRgn to drag its indicator). Conversely, if the control definition function chooses to do its own custom dragging, it should signal the Control Manager not to use the default method by returning a nonzero result.

If the whole control is being dragged, the definition function should call MoveControl to reposition the control to its new location after the user releases the mouse button. If just the indicator is being dragged, the definition function should execute its own position routine (see below) to update the control's setting and redraw it in its window.

The Position Routine

For controls that don't use the Control Manager's default method of dragging the control's indicator (as performed by DragGrayRgn), the control definition function must include a position routine. When the mouse button is released inside the indicator of such a control, TrackControl calls the control definition function with the message posCntl to reposition the indicator and update the control's setting accordingly. The value of param is a point giving the vertical and horizontal offset, in pixels, by which the indicator is to be moved relative to its current position. (Typically, this is 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 high-order word of the LongInt and the horizontal offset in the low-order word. The definition function should calculate the control's new setting based on the given offset, update the contrlValue field, and redraw the control within its window to reflect the new setting.

(note)

The Control Manager procedures SetCtlValue, SetCtlMin, and SetCtlMax do **not** call the control definition function with this message; instead, they pass the drawCntl message to execute the draw routine (see above).

The Thumb Routine

Like the position routine, the thumb routine is required only for controls that don't use the Control Manager's default method of dragging the control's indicator. The control definition function for such a control should respond to the message thumbCntl by calculating the limiting rectangle, slop rectangle, and axis constraint for dragging the control's indicator. Param is a pointer to the following data structure:

```
RECORD
  limitRect, slopRect: Rect;
  axis: INTEGER
END;
```

On entry, param^.limitRect.topLeft contains the point where the mouse button was first pressed. The definition function should store the appropriate values into the fields of the record pointed to by param; they're analogous to the similarly named parameters to DragGrayRgn.

The Track Routine

You can design a control to have its action procedure in the control definition function. To do this, set up the control's initialize routine to store POINTER(-1) in the contrlAction field of the control record, and pass POINTER(-1) in the actionProc parameter to TrackControl. TrackControl will respond by calling the control definition function with the message autoTrack. The definition function should respond like an action procedure, as discussed in detail in the description of TrackControl. It can tell which part of the control the mouse button was pressed in from param, which contains the part code. The track routine for each of the standard control types does nothing.

FORMATS OF RESOURCES FOR CONTROLS

The GetNewControl function takes the resource ID of a control template as a parameter, and gets from that template the same information that the NewControl function gets from eight of its parameters. The resource type for a control template is 'CNTL', and the resource data has the following format:

Number of bytes	Contents
8 bytes	Same as boundsRect parameter to NewControl
2 bytes	Same as value parameter to NewControl
2 bytes	Same as visible parameter to NewControl
2 bytes	Same as max parameter to NewControl
2 bytes	Same as min parameter to NewControl
4 bytes	Same as procID parameter to NewControl
4 bytes	Same as refCon parameter to NewControl
n bytes	Same as title parameter to NewControl (1-byte length in bytes, followed by the characters of the title)

The resource type for a control definition function is 'CDEF'. The resource data is simply the compiled or assembled code of the function.

SUMMARY OF THE CONTROL MANAGER

Constants

```
CONST { Control definition IDs }

      pushButProc   = Ø;   {simple button}
      checkBoxProc  = 1;   {check box}
      radioButProc  = 2;   {radio button}
      useWFont      = 8;   {add to above to use window's font}
      scrollBarProc = 16;  {scroll bar}

      { Part codes }

      inButton      = 1Ø;  {simple button}
      inCheckBox    = 11;  {check box or radio button}
      inUpButton    = 2Ø;  {up arrow of a scroll bar}
      inDownButton  = 21;  {down arrow of a scroll bar}
      inPageUp      = 22;  {"page up" region of a scroll bar}
      inPageDown    = 23;  {"page down" region of a scroll bar}
      inThumb       = 129; {thumb of a scroll bar}

      { Axis constraints for DragControl }

      noConstraint = Ø;    {no constraint}
      hAxisOnly    = 1;    {horizontal axis only}
      vAxisOnly    = 2;    {vertical axis only}

      { Messages to control definition function }

      drawCntl  = Ø;  {draw the control (or control part)}
      testCntl  = 1;  {test where mouse button was pressed}
      calcCRgns = 2;  {calculate control's region (or indicator's)}
      initCntl  = 3;  {do any additional control initialization}
      dispCntl  = 4;  {take any additional disposal actions}
      posCntl   = 5;  {reposition control's indicator and update it}
      thumbCntl = 6;  {calculate parameters for dragging indicator}
      dragCntl  = 7;  {drag control (or its indicator)}
      autoTrack = 8;  {execute control's action procedure}
```

Data Types

```
TYPE ControlHandle = ^ControlPtr;
     ControlPtr    = ^ControlRecord;
```

```
ControlRecord =
        RECORD
          nextControl:   ControlHandle; {next control}
          contrlOwner:   WindowPtr;     {control's window}
          contrlRect:    Rect;          {enclosing rectangle}
          contrlVis:     BOOLEAN;       {TRUE if visible}
          contrlHilite:  BOOLEAN;       {highlight state}
          contrlValue:   INTEGER;       {current setting}
          contrlMin:     INTEGER;       {minimum setting}
          contrlMax:     INTEGER;       {maximum setting}
          contrlDefProc: Handle;        {control definition function}
          contrlData:    Handle;        {data used by contrlDefProc}
          contrlAction:  ProcPtr;       {default action procedure}
          contrlRfCon:   LongInt;       {control's reference value}
          contrlTitle:   Str255         {control's title}
        END;
```

Routines

Initialization and Allocation

```
FUNCTION  NewControl      (theWindow: WindowPtr; boundsRect: Rect;
                           title: Str255; visible: BOOLEAN; value:
                           INTEGER; min,max: INTEGER; procID: INTEGER;
                           refCon: LongInt) : ControlHandle;
FUNCTION  GetNewControl   (controlID: INTEGER; theWindow: WindowPtr) :
                           ControlHandle;
PROCEDURE DisposeControl  (theControl: ControlHandle);
PROCEDURE KillControls    (theWindow: WindowPtr);
```

Control Display

```
PROCEDURE SetCTitle      (theControl: ControlHandle; title: Str255);
PROCEDURE GetCTitle      (theControl: ControlHandle; VAR title:
                          Str255);
PROCEDURE HideControl    (theControl: ControlHandle);
PROCEDURE ShowControl    (theControl: ControlHandle);
PROCEDURE DrawControls   (theWindow: WindowPtr);
PROCEDURE HiliteControl  (theControl: ControlHandle; hiliteState:
                          INTEGER);
```

Mouse Location

```
FUNCTION TestControl   (theControl: ControlHandle; thePoint: Point) :
                        INTEGER;
FUNCTION FindControl   (thePoint: Point; theWindow: WindowPtr; VAR
                        whichControl: ControlHandle) : INTEGER;
FUNCTION TrackControl  (theControl: ControlHandle; startPt: Point;
                        actionProc: ProcPtr) : INTEGER;
```

Control Movement and Sizing

```
PROCEDURE MoveControl (theControl: ControlHandle; h,v: INTEGER);
PROCEDURE DragControl (theControl: ControlHandle; startPt: Point;
                       limitRect,slopRect: Rect; axis: INTEGER);
PROCEDURE SizeControl (theControl: ControlHandle; w,h: INTEGER);
```

Control Setting and Range

```
PROCEDURE SetCtlValue (theControl: ControlHandle; theValue: INTEGER);
FUNCTION  GetCtlValue (theControl: ControlHandle) : INTEGER;
PROCEDURE SetCtlMin   (theControl: ControlHandle; minValue: INTEGER);
FUNCTION  GetCtlMin   (theControl: ControlHandle) : INTEGER;
PROCEDURE SetCtlMax   (theControl: ControlHandle; maxValue: INTEGER);
FUNCTION  GetCtlMax   (theControl: ControlHandle) : INTEGER;
```

Miscellaneous Utilities

```
PROCEDURE SetCRefCon   (theControl: ControlHandle; data: LongInt);
FUNCTION  GetCRefCon   (theControl: ControlHandle) : LongInt;
PROCEDURE SetCtlAction (theControl: ControlHandle; actionProc: ProcPtr);
FUNCTION  GetCtlAction (theControl: ControlHandle) : ProcPtr;
```

Action Procedure for TrackControl

```
If an indicator:      PROCEDURE MyAction;
If not an indicator:  PROCEDURE MyAction (theControl: ControlHandle;
                                          partCode: INTEGER);
```

Control Definition Function

```
FUNCTION MyControl (varCode: INTEGER; theControl: ControlHandle;
                    message: INTEGER; param: LongInt) : LongInt;
```

Assembly-Language Information

Constants

```
; Control definition IDs

pushButProc     .EQU     Ø      ;simple button
checkBoxProc    .EQU     1      ;check box
radioButProc    .EQU     2      ;radio button
useWFont        .EQU     8      ;add to above to use window's font
scrollBarProc   .EQU    16      ;scroll bar
```

```
; Part codes

inButton          .EQU     1Ø      ;simple button
inCheckBox        .EQU     11      ;check box or radio button
inUpButton        .EQU     2Ø      ;up arrow of scroll bar
inDownButton      .EQU     21      ;down arrow of scroll bar
inPageUp          .EQU     22      ;"page up" region of scroll bar
inPageDown        .EQU     23      ;"page down" region of scroll bar
inThumb           .EQU    129      ;thumb of scroll bar

; Axis constraints for DragControl

noConstraint      .EQU      Ø      ;no constraint
hAxisOnly         .EQU      1      ;horizontal axis only
vAxisOnly         .EQU      2      ;vertical axis only

; Messages to control definition function

drawCtlMsg    .EQU   Ø  ;draw the control (or control part)
hitCtlMsg     .EQU   1  ;test where mouse button was pressed
calcCtlMsg    .EQU   2  ;calculate control's region (or indicator's)
newCtlMsg     .EQU   3  ;do any additional control initialization
dispCtlMsg    .EQU   4  ;take any additional disposal actions
posCtlMsg     .EQU   5  ;reposition control's indicator and update it
thumbCtlMsg   .EQU   6  ;calculate parameters for dragging indicator
dragCtlMsg    .EQU   7  ;drag control (or its indicator)
trackCtlMsg   .EQU   8  ;execute control's action procedure
```

Control Record Data Structure

```
nextControl         Handle to next control in control list
contrlOwner         Pointer to this control's window
contrlRect          Control's enclosing rectangle
contrlVis           Flag for whether control is visible
contrlHilite        Highlight state
contrlValue         Control's current setting
contrlMin           Control's minimum setting
contrlMax           Control's maximum setting
contrlDefHandle     Handle to control definition function
contrlData          Data used by control definition function
contrlAction        Default action procedure
contrlRfCon         Control's reference value
contrlTitle         Control's title
contrlSize          Length of above structure except contrlTitle
```

Special Macro Names

Routine name	Macro name
DisposeControl	_DisposControl
GetCtlMax	_GetMaxCtl
GetCtlMin	_GetMinCtl
SetCtlMax	_SetMaxCtl
SetCtlMin	_SetMinCtl

GLOSSARY

action procedure: A procedure, used by the Control Manager function TrackControl, that defines an action to be performed repeatedly for as long as the mouse button is held down.

active control: A control that will respond to the user's actions with the mouse.

button: A standard Macintosh control that causes some immediate or continuous action when clicked or pressed with the mouse. (See also: radio button)

check box: A standard Macintosh control that displays a setting, either checked (on) or unchecked (off). Clicking inside a check box reverses its setting.

control: An object in a window on the Macintosh screen with which the user, using the mouse, can cause instant action with graphic results or change settings to modify a future action.

control definition function: A function called by the Control Manager when it needs to perform type-dependent operations on a particular type of control, such as drawing the control.

control definition ID: A number passed to control-creation routines to indicate the type of control. It consists of the control definition function's resource ID and a variation code.

control list: A list of all the controls associated with a given window.

control record: The internal representation of a control, where the Control Manager stores all the information it needs for its operations on that control.

control template: A resource that contains information from which the Control Manager can create a control.

dial: A control with a moving indicator that displays a quantitative setting or value. Depending on the type of dial, the user may or may not be able to change the setting by dragging the indicator with the mouse.

dimmed: Drawn in gray rather than black.

inactive control: A control that will not respond to the user's actions with the mouse. An inactive control is highlighted in some special way, such as dimmed.

indicator: The moving part of a dial that displays its current setting. The part code of an indicator is always greater than 128 by convention.

invert: To highlight by changing white pixels to black and vice versa.

invisible control: A control that's not drawn in its window.

part code: An integer between 1 and 253 that stands for a particular part of a control (possibly the entire control). Part codes greater than 128 represent indicators.

radio button: A standard Macintosh control that displays a setting, either on or off, and is part of a group in which only one button can be on at a time. Clicking a radio button on turns off all the others in the group, like the buttons on a car radio.

reference value: In a window record or control record, a 32-bit field that the application program may store into and access for any purpose.

thumb: The Control Manager's term for the scroll box (the indicator of a scroll bar).

variation code: The part of a window or control definition ID that distinguishes closely related types of windows or controls.

visible control: A control that's drawn in its window (but may be completely overlapped by another window or other object on the screen).

COMMENTS?

Macintosh User Education encourages your comments on this manual.

- What do you like or dislike about it?

- Were you able to find the information you needed?

- Was it complete and accurate?

- Do you have any suggestions for improvement?

Please send your comments to the author (indicated on the cover page) at 10460 Bandley Drive M/S 3-G, Cupertino CA 95014. Mark up a copy of the manual or note your remarks separately. (We'll return your marked-up copy if you like.)

Thanks for your help!

MACINTOSH USER EDUCATION

The Menu Manager: A Programmer's Guide /MMGR/MENUS

See Also:
Macintosh User Interface Guidelines
Macintosh Operating System Reference Manual
QuickDraw: A Programmer's Guide
The Window Manager: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Desk Manager: A Programmer's Guide
The Toolbox Utilities: A Programmer's Guide

Modification History:

First Draft	P. Stanton-Wyman	
Second Draft	C. Espinosa	12/23/82
Updated (ROM 2.Ø)	C. Espinosa	1/24/83
Third Draft (ROM 3.Ø)	C. Espinosa & C. Rose	5/17/83
Fourth Draft (ROM 7)	C. Rose	11/1/83

ABSTRACT

The Macintosh User Interface frees the user from having to remember long strings of command words by placing all commands in menus. With the menu bar and pull-down menus, the user can at any time see all available menu choices. This manual describes the nature of pull-down menus and how to implement them with the Macintosh Menu Manager.

Summary of significant changes and additions since last version:

- The symbol for showing keyboard equivalents for menu items has changed from a solid apple to the Command key's symbol on the keyboard (page 6).
- The use of the "!" meta-character to indicate a marked menu item has changed (page 11).
- A new procedure, InsertResMenu, has been added (page 18).
- The predefined constant mCalcSize, for the menu definition procedure's message parameter, has been renamed mSizeMsg (page 27).
- For assembly-language programmers, the unconventional macro names for calling several of the Menu Manager routines are now listed under the descriptions of those routines, and some additional system globals are discussed.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Menu Manager, a major component of the Macintosh User Interface Toolbox. *** Eventually it will become part of a comprehensive manual describing the entire Toolbox and Operating System. *** The Menu Manager allows you to create sets of menus, and allows the user to choose from the commands in those menus in a manner consistent with the Macintosh User Interface guidelines.

(hand)

> This manual describes version 7 of the ROM. If you're using a different version, the Menu Manager may not work as discussed here.

Like all documentation about the Toolbox, this manual assumes you're familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's Memory Manager. You should also be familiar with the following:

- The basic concepts and structures behind QuickDraw, particularly points, rectangles, and character style.
- Resources, as described in the Resource Manager manual.
- The Toolbox Event Manager. Some Menu Manager routines should be called only in response to certain events.

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly-language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it. *** Some of that information refers to the "Toolbox equates" file (ToolEqu.Text), which the reader will have learned about in an earlier chapter of the final comprehensive manual. ***

The manual begins with an introduction to the Menu Manager and the appearance of menus on the Macintosh. It then discusses the basics of menus: the relationship between menus and resources, some internal structures related to menus, and information about how to create menus.

Next, a section on using the Menu Manager introduces its routines and tells how they fit into the flow of your application. This is followed by detailed descriptions of all Menu Manager procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that will not interest all readers: special information is provided for programmers who want to define their own menus, and the exact formats of resources related to menus are described.

Finally, there's a summary of the Menu Manager, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE MENU MANAGER

The Menu Manager supports the use of menus, an integral part of the Macintosh User 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 Macintosh 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 that title (by inverting it) and "pulls down" the menu below it. As long as the mouse button is held down, the menu is displayed. Dragging the mouse through the menu items causes each of the items 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.

After a successful choice, 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 and releases the mouse button, no choice is made: the menu simply disappears and the application takes no action. The user is never forced to choose a command once a menu has been pulled down.

The Menu Bar

The menu bar always appears at the top of the Macintosh screen, 2Ø pixels high and as wide as the screen. It appears in front of all windows; nothing but the cursor ever appears in front of the menu bar. The menu bar is white and has a thin black lower border, and the menu titles in it are always in the system font and the system font size (see Figure 1).

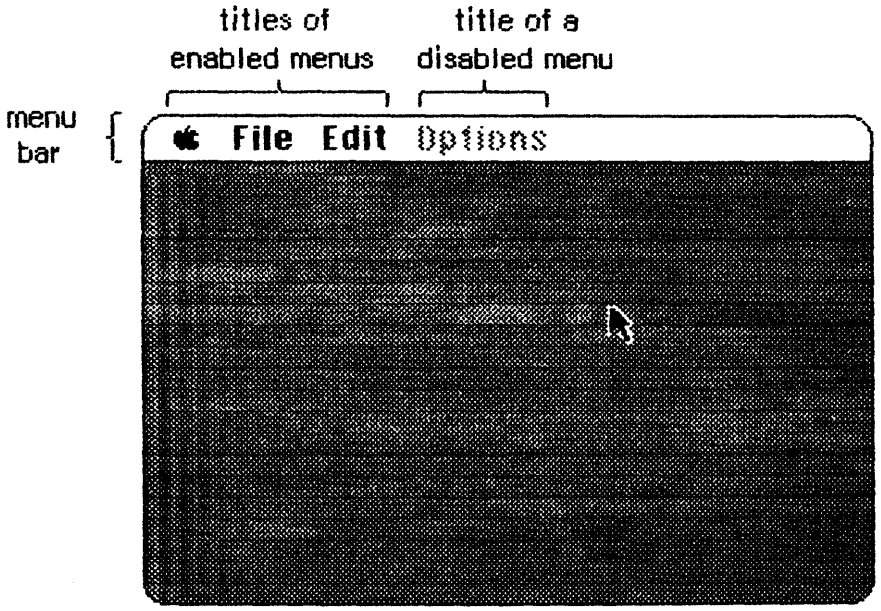


Figure 1. The Menu Bar

In applications that support desk accessories, the first menu should be the standard Apple menu (the menu whose title is an Apple symbol). This menu contains the names of all available desk accessories. When the user chooses a desk accessory, the title of a menu belonging to it may also appear in the menu bar, for as long as the accessory is active, or the entire menu bar may be occupied by menus belonging to the desk accessory. (Desk accessories are discussed in detail in the Desk Manager manual.)

A menu may temporarily be disabled, so that none of the items in the menu can be chosen. The title of a disabled menu and every item in it appear dimmed in the menu bar (that is, drawn in gray rather than black).

The maximum number of menu titles in the menu bar is 16; however, ten to twelve titles is usually all that will fit. If you're having trouble fitting your menus in the menu bar, you should review your menu organization and menu titles.

Appearance of Menus

A standard menu consists of a number of lines of text, listed vertically inside a shadowed rectangle (see Figure 2). Menus always appear in front of everything else (except the cursor); in Figure 2, the menu appears in front of a document window already on the screen.

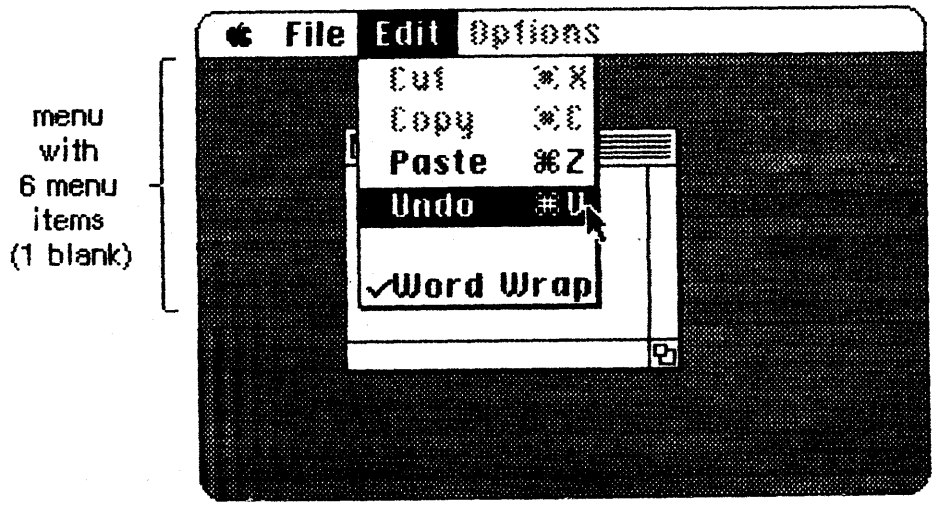


Figure 2. A Standard Menu

Each line of text is one menu item that the user can choose from that menu. The text always appears in the system font and the system font size. Each item can have a few visual variations from the standard appearance:

- An icon to the left of the item's text, to give a symbolic representation of the item's meaning or effect.

- A check mark or other character to the left of the item's text (or icon, if any), to denote the status of the item or of the mode it controls.

- The Command key symbol and another character to the right of the item's text, to show that the item may be invoked from the keyboard (that is, it has a keyboard equivalent).

- A character style other than the standard, such as bold, italic, underline, or a combination of these. (The QuickDraw manual gives a full discussion of character style.)

- A dimmed appearance, to indicate that the item is disabled.

(hand)
Special symbols or icons may have an unusual appearance when dimmed; notice the dimmed Command symbol in the Cut and Copy menu items in Figure 2.

The maximum number of menu items that will fit in a standard menu is 20 (minus 1 for every item that contains an icon). The fewer menu items you have, the simpler and clearer the menu appears to the user. To separate groups of items, you may use blank menu items or items consisting entirely of dashes.

If the standard menu doesn't suit your needs (for example, if you want more graphics or perhaps 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.

MENUS AND RESOURCES

The general definition of how a certain type of menu looks and behaves is determined by a menu definition procedure, which is usually stored as a resource in a resource file. Most applications will use the predefined menu definition procedure in the system resource file; others may write their own menu definition procedures (as described later in the section "Defining Your Own Menus").

One way to define the contents of your application's menus is to have your program create them manually, item by item. When you create a menu this way, the Menu Manager automatically sets it up to use the standard menu definition procedure and gets that procedure from the system resource file. The standard menu definition procedure has the capabilities described above: it lists the menu items vertically, and each one may have an icon, check mark, keyboard equivalent, different character style, or dimmed appearance.

You can also set up your application's menus by reading them in from a resource file. This is strongly recommended, for two reasons: it makes your application smaller, and it allows the menu items to be edited for documentation or translated to foreign languages without affecting the application's source code. The Menu Manager allows you to read not only individual menus but also complete menu bars from a resource file.

(hand)

> You can create menus and menu bars and store them in resource files with the aid of the Resource Editor *** eventually ***. The Resource Editor relieves you of having to know the exact formats of these resources in the file, but for interested programmers this information is given in the section "Formats of Resources for Menus". *** In the absence of the Resource Editor, you can write a small program to create your menus using the Menu Manager procedure AppendMenu, and store them in a resource file using the standard Resource Manager calls. You can also use the interim Resource Compiler; see the manual "Putting Together a Macintosh Application" for more information. ***

Even if you don't store entire menus in resource files, it's a good idea to store the text strings they contain as resources; you can call the Resource Manager directly to read them in. Icons in menus are read from resource files; in this case, the Menu Manager calls the Resource Manager.

There's one other interaction between menus and resources: a Menu Manager procedure that scans all open resource files for resources of a given type and install the names of all available resources of that type into a given menu. This is how you fill a menu with the names of all available desk accessories, for example.

MENU RECORDS

The Menu Manager keeps all the information it needs for its operations on a particular menu in a menu record. The menu record contains:

- The menu ID. For menus stored in resource files, this is the resource ID; for menus created by your application, it's any positive number (less than 32768) that you choose to identify the menu.

- The menu title.

- The contents of the menu; the text and other parts of each item.

- The horizontal and vertical dimensions of the menu, in pixels. The menu items appear inside the rectangle formed by these dimensions; the black border and shadow of the menu appear outside that rectangle.

- A handle to the menu definition procedure.

- Flags telling whether each menu item is enabled or disabled, and whether the menu itself is enabled or disabled.

The data type for a menu record is called MenuInfo. A menu is a dynamic, relocatable data structure and is referred to by a handle.

```
TYPE MenuPtr    = ^MenuInfo;
     MenuHandle = ^MenuPtr;
```

You can store into and access all the necessary fields of a menu record with Menu Manager routines, so normally you don't have to know its exact structure. Advanced users, however--particularly those who define their own types of menus--may need to know some of the field names.

```
TYPE MenuInfo = RECORD
                  menuID:      INTEGER;
                  menuWidth:   INTEGER;
                  menuHeight:  INTEGER;
                  menuProc:    Handle;
                  enableFlags: PACKED ARRAY [Ø..31] OF BOOLEAN;
                  menuData:    Str255
                END;
```

The menuID field contains the menu ID.

The menuWidth and menuHeight fields contain the menu's horizontal and vertical dimensions, respectively.

The menuProc field contains a handle to the menu definition procedure for this type of menu.

The Øth element of the enableFlags array is TRUE if the menu is enabled, or FALSE if it's disabled. The remaining elements similiarly determine whether each item in the menu is enabled or disabled.

The menuData field contains the menu title followed by variable-length data that defines the text and other parts of the menu items. The Str255 data type enables you to access the title from Pascal; there's actually additional data beyond the title that's inaccessible from Pascal and is not reflected in the MenuInfo data structure.

(eye)

You can read the menu title directly from the menuData field, but do not change the title directly, or the data defining the menu items may be destroyed.

Assembly-language note: The Toolbox equates file includes menuBlkSize, the length in bytes of all the fields of a menu record except menuData.

THE MENU LIST

The Menu Manager keeps a list of menu handles for all menus in the menu bar. The user can pull down and choose from any menu whose handle is in this menu list. The menu bar shows the titles, in order, of all menus in the menu list.

You can have menus that aren't in the menu list. These menus' titles don't appear in the menu bar, the menus can't be pulled down, and their items can't be chosen. Such menus are useful as "reserve" menus to hold items not normally available to the user; these items can be exchanged with items in other menus, or entire reserve menus can be added to the menu bar.

The Menu Manager provides all the necessary routines for manipulating the menu list, so there's no need to access it yourself directly. As a general rule, routines that deal specifically with menus in the menu list use the menu ID to refer to menus; those that deal with any menus, whether in the menu list or not, use the menu handle to refer to menus. Some routines refer to the menu list as a whole, with a handle.

Assembly-language note: The system global menuList contains a handle to the current menu list.

CREATING A MENU

For an application to create menus itself, rather than read them from a resource file, it must call the NewMenu and AppendMenu routines of the Menu Manager. NewMenu creates a new menu data structure, returning a handle to it. AppendMenu takes a string and a handle to a menu and adds the items in the string to the end of the menu.

The string passed to AppendMenu consists mainly of the text of the menu items (for a blank item, one or more spaces). Other characters interspersed in the string can have special meaning to the Menu Manager. These characters, called meta-characters, are used in conjunction with text to separate menu items or alter their appearance. The meta-characters do not appear in the menu.

Meta-character	Meaning
; or Return	Separates items
^	Item has an icon
!	Item has a check mark or other mark
<	Item has a special character style
/	Item has a keyboard equivalent
(	Item is disabled

None, any, or all of these meta-characters can appear in the AppendMenu string; they are described in detail below. To add one text-only item to a menu would require a simple string without any meta-characters:

```
AppendMenu(thisMenu,'Just Enough');
```

An extreme example could use many meta-characters:

```
AppendMenu(thisMenu,'(Too Much^1<B/T');
```

This example adds to the menu an item whose text is "Too Much", which is disabled, has icon number 1, is boldfaced, and can be invoked by Command-T. Your menu items should be much simpler than this.

(hand)
If you want any of the meta-characters to appear in the text of a menu item, you can include them by changing the text with the Menu Manager procedure SetItem.

Separating Items

Each call to AppendMenu can add one or many items to the menu. To add multiple items in the same call, use a semicolon (";") or a Return character to separate the items. The call

```
AppendMenu(thisMenu,'Cut;Copy');
```

has exactly the same effect as the calls

```
AppendMenu(thisMenu,'Cut');
AppendMenu(thisMenu,'Copy');
```

Items with Icons

A circumflex ("^") followed by a digit from 1 to 9 indicates that an icon should appear to the left of the menu item's text. The digit, which is called the icon number, yields the resource ID of the icon in the resource file. Resource IDs 257 through 511 are reserved for menu icons; thus the Menu Manager adds 256 to the icon number to get the proper resource ID.

If you need to install more than nine icons, you can use the SetItemIcon procedure.

(hand)
The Menu Manager gets the icon number by subtracting 48 from the ASCII code of the character following the "^" (since, for example, the ASCII code of "1" is 49). You can actually follow the "^" with any character that has an ASCII code greater than 48.

Marked Items

You can use an exclamation point ("!") to cause a check mark or any other character to be placed to the left of the menu item's text (or icon, if any). Follow the exclamation point with the character of your choice; note, however, that you may not be able to type a check mark or certain other special characters (such as the Apple symbol) from the keyboard. To specify one of these characters, you need to take special measures: Declare a string variable to have the length of the desired AppendMenu string, and assign it that string with a space following the exclamation point. Then separately store the special character in the position of the space. The following predefined constants may be useful:

```
CONST checkMark   = 18;    {check mark}
      appleSymbol = 2Ø;    {Apple symbol}
```

For example, suppose you want to use AppendMenu to specify a menu item that has the text "Word Wrap" (nine characters) and a check mark to its left. You can declare the string variable

```
VAR s: STRING[11];
```

and do the following:

```
s := 'Word Wrap! '
s[11] := CHR(checkMark);
AppendMenu(thisMenu,s);
```

Character Style of Items

The system font is the only font available for menus; however, you can vary the character style for clarity and distinction. The meta-character used to specify the character style is the left angle bracket, "<". With AppendMenu, you can assign one and only one of the stylistic variations listed below.

<B	Bold
<I	Italic
<U	Underline
<O	Outline
<S	Shadow

The SetItemStyle procedure allows you to assign any character style to an item. For a further discussion of character style, see the QuickDraw manual.

Items with Keyboard Equivalents

Any menu item that can be chosen from a menu may also be associated with a key on the keyboard. Pressing this key while holding down the Command key invokes the item just as if it had been chosen from the menu.

A slash ("/") followed by a character associates that character with the item. The specified character (preceded by the Command key symbol) appears at the right of the item's text in the menu. For consistency between applications, the character should be uppercase if it's a letter. When invoking the item, the user can type the letter in either uppercase or lowercase. For example, if you specify 'Copy/C', the Copy command can be invoked by holding down the Command key and typing either C or c.

An application that receives a key down event with the Command key held down can call the Menu Manager with the typed character and receive the menu ID and item number of the item associated with that character.

Disabled Items

All items in a menu are usually choosable. There will be times when you don't want an item to be choosable, either initially or for the duration of your program (perhaps due to the program's incomplete state). The meta-character that disables an item is the left parenthesis "(". A disabled item cannot be chosen; it appears dimmed in the menu and is not highlighted when the cursor moves over it.

Blank items in a menu should always be disabled, as should any items used to separate groups of items. For example, the call

```
AppendMenu(thisMenu,'Undo;( ;Word Wrap');
```

adds two enabled menu items, Undo and Word Wrap, with a disabled blank item between them. Note that one or more spaces are required to specify a blank item.

You can change the enabled or disabled state of a menu item with the DisableItem and EnableItem procedures.

USING THE MENU MANAGER

This section discusses how the Menu Manager routines fit into the general flow of an application program and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

To use the Menu Manager, you must have previously called InitGraf to initialize QuickDraw, InitFonts to initialize the Font Manager, and InitWindows to initialize the Window Manager. The first Menu Manager routine to call is the initialization procedure InitMenus.

Your application can set up the menus it needs in any number of ways:

- Allocate the menus with NewMenu, fill them with items using AppendMenu, and place them in the menu bar using InsertMenu.
- Read the menus individually from a resource file using GetMenu, and place them in the menu bar using InsertMenu.
- Read an entire prepared menu list from a resource file with GetNewMBar, and place it in the menu bar with SetMenuBar.
- Allocate a menu with NewMenu, fill it with items using AddResMenu to get the names of all available resources of a given type, and place the menu in the menu bar using InsertMenu.

You can use AddResMenu or InsertResMenu to add items from resource files to any menu, regardless of how you created the menu or whether it already contains any items.

If you call NewMenu to allocate a menu, it will store a handle to the standard menu definition procedure in the window record; so if you want the menu to be one of your own design, you must replace that handle with a handle to your own menu definition procedure. For more information, see "Defining Your Own Menus".

At any time you can change or examine the appearance of an individual menu item with the SetItem and GetItem procedures (and similar procedures to set or get the item's icon, style, check mark, and so on). You can also change the number and order of menus in the menu list with InsertMenu and DeleteMenu, or change the entire menu list with ClearMenuBar, GetNewMBar, GetMenuBar, and SetMenuBar.

When your application receives a mouse down event, and the Window Manager's FindWindow function returns the predefined constant inMenuBar, your application should call the Menu Manager's MenuSelect function, supplying it with the point where the mouse button was pressed. MenuSelect will pull down the appropriate menu, and retain control--tracking the mouse, highlighting menu items, and pulling down other menus--until the user releases the mouse button. MenuSelect returns a long integer to the application: the high-order word contains the menu ID of the menu that was chosen, and the low-order word contains the menu item number of the item that was chosen. The menu item number is the index, starting from 1, of the item in the menu. The entire long integer is Ø if no item was chosen.

- If the long integer is Ø, your application should just continue to poll for further events.

- If the long integer is nonzero, the application should take the appropriate action for when the menu item specified by the low-order word is chosen from the menu whose ID is in the high-order word. Only after the action is completely finished (after all dialogs, alerts, or screen actions have been taken care of) should your application call HiliteMenu(Ø) to remove the highlighting from the menu bar, signaling the completion of the action.

Keyboard equivalents are handled in much the same manner. When your application receives a key down event with the Command key held down, it should call the MenuKey function, supplying it with the character that was typed. MenuKey will return a long integer with the same format as that of MenuSelect, and the application can handle the long integer in the manner described above.

(hand)
You can use the Toolbox Utility routines LoWord and HiWord to extract the high-order and low-order words of a given long integer, as described in the Toolbox Utilities manual.

When you no longer need a menu, call DisposeMenu if you allocated it with NewMenu, or call the Resource Manager procedure ReleaseResource if you used GetMenu.

MENU MANAGER ROUTINES

This section describes all the Menu Manager procedures and functions. They're presented in their Pascal form; for information on using them from assembly language, see "Using the Toolbox from Assembly Language" *** doesn't exist, but see "Using QuickDraw from Assembly Language" in the QuickDraw manual ***.

Initialization and Allocation

```
PROCEDURE InitMenus;
```

InitMenus initializes system globals used by the Menu Manager, sets up its internal data structures, clears the menu list, and draws the (empty) menu bar. Call it once before all other Menu Manager routines. An application should never have to call this procedure more than once; to start afresh with all new menus, use ClearMenuBar.

(hand)
InitWindows, which you previously called to initialize the Window Manager, will already have drawn the menu bar; InitMenus also draws the menu bar just in case it does happen to be called in mid-application.

```
FUNCTION NewMenu (menuID: INTEGER; menuTitle: Str255) : MenuHandle;
```

NewMenu allocates space for a new menu with the given menu ID and title, and returns a handle to it. The new menu (which is created empty) is not installed in the menu list. To use this menu, you must first call AppendMenu or AddResMenu to fill it with items, InsertMenu to place it in the menu list, and DrawMenuBar to update the menu bar to include the new title.

Application menus should always have positive menu IDs. Negative menu IDs are reserved for menus belonging to desk accessories. No menu should ever have a menu ID of Ø.

To set up the title of the Apple menu of desk accessory names, you can use the predefined constant appleSymbol (equal to 2Ø, the ASCII code of the Apple symbol). For example, you can declare the string variable

```
VAR myTitle: STRING[1];
```

and do the following:

```
myTitle := ' ';
myTitle[1] := CHR(appleSymbol);
```

(hand)
Once a menu is created with NewMenu, the only way to deallocate the memory it occupies is by calling DisposeMenu.

```
FUNCTION GetMenu (menuID: INTEGER) : MenuHandle;
```

GetMenu returns a menu handle for the menu having the given resource ID. If the menu isn't already in memory, GetMenu calls the Resource Manager to read it from the resource file into a menu record in memory. It stores the handle to the menu definition procedure in the menu record, reading the procedure from the resource file into memory if necessary. To use this menu, you must call InsertMenu to place it in the menu list and DrawMenuBar to update the menu bar to include the new title.

(hand)
To deallocate the memory occupied by a menu that you read from a resource file with GetMenu, use the Resource Manager procedure ReleaseResource.

Assembly-language note: The macro you invoke to call GetMenu from assembly language is named _GetRMenu.

```
PROCEDURE DisposeMenu (menu: MenuHandle);
```

Call DisposeMenu to deallocate the memory occupied by a menu that you allocated with NewMenu. (For menus read from a resource file with GetMenu, use the Resource Manager procedure ReleaseResource instead.) This is useful if you've created temporary menus that you no longer need.

(eye)
Make sure you remove the menu from the menu list (with DeleteMenu) before disposing of it. Also be careful not to use the menu handle after disposing of the menu.

Assembly-language note: The macro you invoke to call DisposeMenu from assembly language is named _DisposMenu.

```
PROCEDURE AppendMenu (menu: MenuHandle; data: Str255);
```

AppendMenu adds an item or items to the end of the given menu, which must previously have been allocated by NewMenu or read from a resource file by GetMenu. The data string consists of the text of the menu item; it may be blank but should not be the null string. As described in the section "Creating a Menu", the following meta-characters may be embedded in the data string:

Meta-character	Usage
; or Return	Separates multiple items
^	Followed by an icon number, adds that icon to the item
!	Followed by a character, marks the item with that character
<	Followed by B, I, U, O, or S, sets the character style of the item
/	Followed by a character, associates a keyboard equivalent with the item
(	Disables the item

Once items have been appended to a menu, they cannot be removed or rearranged. AppendMenu works properly whether or not the menu is in the menu list.

```
PROCEDURE AddResMenu (menu: MenuHandle; theType: ResType);
```

AddResMenu searches all open resource files for resources of type theType and appends the names of all resources it finds to the given menu. Each resource name appears in the menu as an enabled item, without an icon or mark, and in the normal character style. The standard Menu Manager calls can be used to get the name or change its appearance, as described below under "Controlling Items' Appearance".

(hand)

So that you can have resources of the given type that won't appear in the menu, AddResMenu does not append any resource names that begin with a period (".").

Use this procedure to fill a menu with the names of all available fonts or desk accessories. For example, if you declare a variable as

```
VAR fontMenu: MenuHandle;
```

you can set up a menu containing all font names as follows:

```
fontMenu := NewMenu(5,'Fonts');
AddResMenu(fontMenu,'FONT');
```

```
PROCEDURE InsertResMenu (menu: MenuHandle; theType: ResType; afterItem:
          INTEGER);
```

InsertResMenu is the same as AddResMenu (above) except that it inserts the resource names in the menu where specified by the afterItem parameter: if afterItem is Ø, the names are inserted before the first menu item; if it's the item number of an item in the menu, they're inserted after that item; if it's equal to or greater than the last item number, they're appended to the menu as by AddResMenu.

(hand)

InsertResMenu inserts the names in the reverse of the order that AddResMenu appends them. For consistency in the appearance of menus between applications, use AddResMenu instead of InsertResMenu if possible.

Forming the Menu Bar

```
PROCEDURE InsertMenu (menu: MenuHandle; beforeID: INTEGER);
```

InsertMenu inserts a menu into the menu list before the menu whose menu ID equals beforeID. If beforeID is Ø (or isn't the ID of any menu in the menu list), the new menu is added after all others. If the menu is already in the menu list, InsertMenu does nothing. Be sure to call DrawMenuBar to update the menu bar.

```
PROCEDURE DrawMenuBar;
```

DrawMenuBar redraws the menu bar according to the menu list, incorporating any changes since the last call to DrawMenuBar. Any highlighted menu title remains highlighted when drawn by DrawMenuBar. This procedure should always be called after a sequence of InsertMenu or DeleteMenu calls, and after ClearMenuBar, SetMenuBar, or any other routine that changes the menu list.

```
PROCEDURE DeleteMenu (menuID: INTEGER);
```

DeleteMenu deletes a menu from the menu list. If there's no menu with the given menu ID in the menu list, DeleteMenu has no effect. Be sure to call DrawMenuBar to update the menu bar; the menu titles following the deleted menu will move over to fill the vacancy.

(hand)

DeleteMenu simply removes the menu from the list of currently available menus; it doesn't deallocate the menu data structure.

PROCEDURE ClearMenuBar;

Call ClearMenuBar to remove all menus from the menu list when you want to start afresh with all new menus. Be sure to call DrawMenuBar to update the menu bar.

(hand)
ClearMenuBar, like DeleteMenu, doesn't deallocate the menu data structures; it merely removes them from the menu list.

You don't have to call ClearMenuBar at the beginning of your program, because InitMenus clears the menu list for you.

FUNCTION GetNewMBar (menuBarID: INTEGER) : Handle;

GetNewMBar creates a menu list as defined by the menu bar resource having the given resource ID, and returns a handle to it. If the resource isn't already in memory, GetNewMBar reads it into memory from the resource file. It calls GetMenu to get each of the individual menus.

To make the menu list the current menu list, call SetMenuBar. To dispose of the memory occupied by the menu list, use the Memory Manager procedure DisposHandle.

(eye)
You don't have to know the individual menu IDs to use GetNewMBar, but that doesn't mean you don't have to know them at all: to do anything further with a particular menu, you have to know its ID or its handle (which you can get by passing the ID to GetMHandle, as described below under "Miscellaneous Utilities").

FUNCTION GetMenuBar : Handle;

GetMenuBar creates a copy of the current menu list and returns a handle to the copy. You can then add or remove menus from the menu list (with InsertMenu, DeleteMenu, or ClearMenuBar), and later restore the saved menu list with SetMenuBar. To dispose of the memory occupied by the saved menu list, use the Memory Manager procedure DisposHandle.

(eye)
GetMenuBar doesn't copy the menus themselves, only a list of their handles. Do not dispose of any menus that might be in a saved menu list!

PROCEDURE SetMenuBar (menuBar: Handle);

Given a handle to a menu list, SetMenuBar makes it the current menu list. You can use this procedure to restore a menu list previously

saved by GetMenuBar, or pass it a handle returned by GetNewMBar. Be sure to call DrawMenuBar to update the menu bar.

Choosing From a Menu

```
FUNCTION MenuSelect (startPt: Point) : LongInt;
```

When a mouse down event occurs in the menu bar, you should call MenuSelect with startPt (in global coordinates) equal to the point where the mouse button was pressed. MenuSelect tracks the mouse, pulling down menus as needed and highlighting menu items under the cursor. When the mouse button is released over an enabled item in an application menu, MenuSelect returns a long integer whose high-order word is the menu ID of the menu, and whose low-order word is the menu item number for the item chosen (see Figure 3). It leaves the selected menu title highlighted. After performing the chosen task, your application should call HiliteMenu(Ø) to remove the highlighting from the menu title.

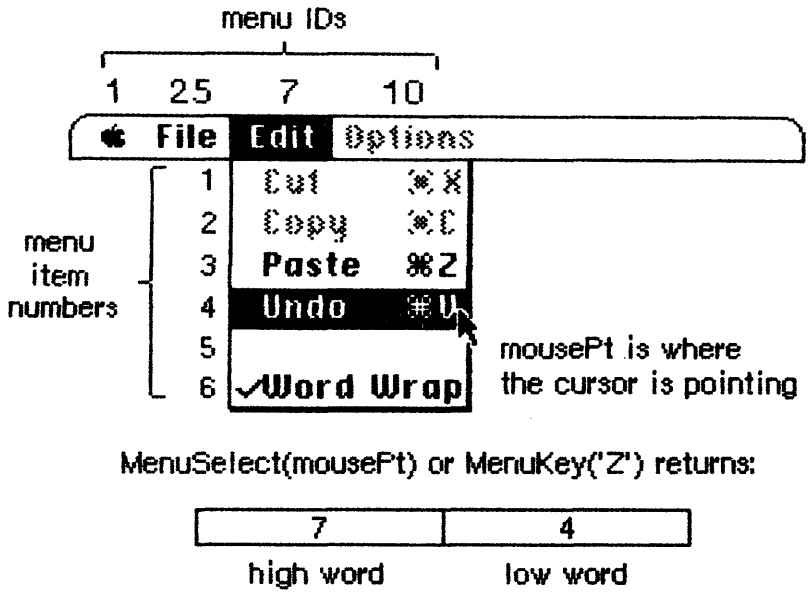


Figure 3. MenuSelect and MenuKey

MenuSelect returns Ø if no choice is made; this includes the case where the mouse button is released over a disabled menu item (such as the blank item in Figure 3) or over any menu title.

If the mouse button is released over an enabled item in a menu belonging to a desk accessory, MenuSelect passes the menu ID and item number to the Desk Manager procedure SystemMenu for processing and returns Ø to your application.

Assembly-language note: If the system global mBarEnable is nonzero, MenuSelect knows that every menu currently in the menu bar belongs to a desk accessory. (See the Desk Manager manual for more information.) The system global menuHook normally contains Ø; you can store in it the address of a routine having no parameters, and MenuSelect will call that routine repeatedly while the mouse button is down.

FUNCTION MenuKey (ch: CHAR) : LongInt;

MenuKey maps the given character to the associated menu and item for that character. When you get a key down event with the Command key held down, call MenuKey with the character that was typed (which can be found in the low-order byte of the event message). MenuKey highlights the appropriate menu title and returns a long integer just as MenuSelect does. This long integer contains the menu ID in its high-order word and the menu item number in its low-order word (see Figure 3 above). After performing the chosen task, your application should call HiliteMenu(Ø) to remove the highlighting from the menu title.

MenuKey returns Ø if the given character isn't associated with any enabled menu item currently in the menu list.

If the given character invokes a menu item in a menu belonging to a desk accessory, MenuKey (like MenuSelect) passes the menu ID and item number to the Desk Manager procedure SystemMenu for processing and returns Ø to your application.

(hand)

There should never be more than one item in the menu list with the same keyboard equivalent, but if there is, MenuKey returns the first such item encountered (scanning the menus from left to right and their items from top to bottom).

PROCEDURE HiliteMenu (menuID: INTEGER);

HiliteMenu highlights the title of the given menu, or does nothing if the title is already highlighted. Since only one menu title can be highlighted at a time, it unhighlights any previously highlighted menu title. If menuID is Ø (or isn't the ID of any menu in the menu list), HiliteMenu simply unhighlights whichever menu title is highlighted.

After MenuSelect or MenuKey, your application should perform the chosen task and then call HiliteMenu(Ø) to unhighlight the chosen menu title.

Assembly-language note: The system global theMenu contains the menu ID of the currently highlighted menu.

Controlling Items' Appearance

```
PROCEDURE SetItem (menu: MenuHandle; item: INTEGER; itemString:
          Str255);
```

SetItem changes the text of the given menu item to itemString. It doesn't recognize the meta-characters used in AppendMenu; if you include them in itemString, they will appear in the text of the menu item. The attributes already in effect for this item--its character style, icon, and so on--remain in effect. ItemString may be blank but should not be the null string.

Use SetItem to flip between two alternative menu items--for example, to change "Show Clipboard" to "Hide Clipboard" when the Clipboard is already showing.

(hand)
We heartily recommend against capricious changing of menu items.

```
PROCEDURE GetItem (menu: MenuHandle; item: INTEGER; VAR itemString:
          Str255);
```

GetItem returns the text of the given menu item in itemString. It doesn't place any meta-characters in the string. This procedure is useful for getting the name of a menu item that was installed with AddResMenu or InsertResMenu.

```
PROCEDURE DisableItem (menu: MenuHandle; item: INTEGER);
```

Given a menu item number in the item parameter, DisableItem disables that menu item; given Ø in the item parameter, it disables the entire menu.

Disabled menu items appear dimmed and are not highlighted when the cursor moves over them. MenuSelect and MenuKey return Ø if the user attempts to invoke a disabled item. Use DisableItem to disable all menu choices that aren't appropriate at a given time (such as a Cut command when there's no text selection).

All menu items are initially enabled unless you specify otherwise (such as by using the "(" meta-character in a call to AppendMenu).

Every menu item in a disabled menu is dimmed. The menu title is also dimmed, but you must call DrawMenuBar to update the menu bar to show the dimmed title.

```
PROCEDURE EnableItem (menu: MenuHandle; item: INTEGER);
```

Given a menu item number in the item parameter, EnableItem enables the item; given Ø in the item parameter, it enables the entire menu. (The item or menu may have been disabled with the DisableItem procedure, or the item may have been disabled with the "(" meta-character in the AppendMenu string.) The item or menu title will no longer appear dimmed and can be chosen like any other enabled item or menu.

```
PROCEDURE CheckItem (menu: MenuHandle; item: INTEGER; checked:
            BOOLEAN);
```

CheckItem places or removes a check mark at the left of the given menu item. After you call CheckItem with checked=TRUE, a check mark will appear each subsequent time the menu is pulled down. Calling CheckItem with checked=FALSE removes the check mark from the menu item (or, if it's marked with a different character, removes that mark).

Menu items are initially unmarked unless you specify otherwise (such as with the "!" meta-character in a call to AppendMenu).

```
PROCEDURE SetItemIcon (menu: MenuHandle; item: INTEGER; icon: INTEGER);
```

SetItemIcon associates the given menu item with an icon. It sets the item's icon number to the given value (an integer from 1 to 255). The Menu Manager adds 256 to the icon number to get the icon's resource ID, which it passes to the Resource Manager to get the corresponding icon.

(eye)

If you deal directly with the Resource Manager to read or store menu icons, be sure to adjust your icon numbers accordingly.

Menu items initially have no icons unless you specify otherwise (such as with the "^" meta-character in a call to AppendMenu).

Assembly-language note: The macro you invoke to call SetItemIcon from assembly language is named _SetItmIcon.

```
PROCEDURE GetItemIcon (menu: MenuHandle; item: INTEGER; VAR icon:
            INTEGER);
```

GetItemIcon returns the icon number associated with the given menu item, as an integer from 1 to 255, or Ø if the item has not been associated with an icon. The icon number is 256 less than the icon's resource ID.

Assembly-language note: The macro you invoke to call GetItemIcon from assembly language is named _GetItmIcon.

```
PROCEDURE SetItemStyle (menu: MenuHandle; item: INTEGER; chStyle:
            Style);
```

SetItemStyle changes the character style of the given menu item to chStyle. For example:

```
SetItemStyle(thisMenu,1,[bold,italic]);      {bold and italic}
```

Menu items are initially in the normal character style unless you specify otherwise (such as with the "<" meta-character in a call to AppendMenu).

Assembly-language note: The macro you invoke to call SetItemStyle from assembly language is named _SetItmStyle.

```
PROCEDURE GetItemStyle (menu: MenuHandle; item: INTEGER; VAR chStyle:
            Style);
```

GetItemStyle returns the character style of the given menu item in chStyle.

Assembly-language note: The macro you invoke to call GetItemStyle from assembly language is named _GetItmStyle.

```
PROCEDURE SetItemMark (menu: MenuHandle; item: INTEGER; markChar:
          CHAR);
```

SetItemMark marks the given menu item in a more general manner than CheckItem. It allows you to place any character in the system font, not just the check mark, to the left of the item. You can specify some useful values for the markChar parameter with the following predefined constants:

```
CONST checkMark   = 18;    {check mark}
      appleSymbol = 2Ø;    {Apple symbol}
      noMark      = Ø;     {nothing, to remove a mark}
```

Assembly-language note: The macro you invoke to call SetItemMark from assembly language is named _SetItmMark.

```
PROCEDURE GetItemMark (menu: MenuHandle; item: INTEGER; VAR markChar:
          CHAR);
```

GetItemMark returns in markChar whatever character the given menu item is marked with, or the NUL character (ASCII code Ø) if no mark is present.

Assembly-language note: The macro you invoke to call GetItemMark from assembly language is named _GetItmMark.

Miscellaneous Utilities

```
PROCEDURE SetMenuFlash (menu: MenuHandle; count: INTEGER);
```

When the mouse button is released over an enabled menu item, the item blinks briefly to confirm the choice. Normally your application need not be concerned about the duration of the blinking, but for special situations SetMenuFlash allows you to control the duration for all items in the given menu. Calling SetMenuFlash with a count of Ø disables blinking; calling it with a count of 2 (the default value) will cause items to blink for about Ø.1 second. A count of 3 is appropriate for naive user applications. Values greater than 3 can be annoyingly slow.

Assembly-language note: The macro you invoke to call SetMenuFlash from assembly language is named _SetMFlash. The current count is stored in the system global menuFlash.

(hand)

Items in both standard and nonstandard menus blink when chosen. The appearance of the blinking for a nonstandard menu depends on the menu definition procedure, as described under "Defining Your Own Menus".

```
PROCEDURE CalcMenuSize (menu: MenuHandle);
```

You can use CalcMenuSize to recalculate the horizontal and vertical dimensions of a menu whose contents have been changed (and store them in the appropriate fields of the menu record). CalcMenuSize is called automatically after every AppendMenu, SetItem, SetItemIcon, and SetItemStyle call.

```
FUNCTION CountMItems (menu: MenuHandle) : INTEGER;
```

CountMItems returns the number of menu items in the given menu.

```
FUNCTION GetMHandle (menuID: INTEGER) : MenuHandle;
```

Given the menu ID of a menu currently installed in the menu list, GetMHandle returns a handle to that menu; given any other menu ID, it returns NIL.

```
PROCEDURE FlashMenuBar (menuID: INTEGER);
```

If menuID is Ø (or isn't the ID of any menu in the menu list), FlashMenuBar inverts the entire menu bar; otherwise, it inverts the title of the given menu.

DEFINING YOUR OWN MENUS

Normally when you create a menu you get the standard type of Macintosh menu, as described in this manual. You may, however, want to define your own type of menu, such as one with more graphics or perhaps 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 must write a menu definition procedure. The menu definition procedure defines the menu by

performing basic operations such as drawing the menu. When the Menu Manager needs to perform one of these operations, it calls the menu definition procedure with a parameter that identifies the operation, and the menu definition procedure in turn takes the appropriate action.

Usually you'll store the menu definition procedure as a resource in a resource file. If you won't be sharing it with other applications, you may want to include it with your application code instead.

When you create a menu with NewMenu, it stores a handle to the standard menu definition procedure in the menu record's menuProc field; you must replace this with a handle to your own menu definition procedure. If your definition procedure is in a resource file, you get the handle by calling the Resource Manager to read it from the resource file into memory.

Instead of creating menus with NewMenu, your application may read the menus from a resource file with GetMenu (or GetNewMBar, which calls GetMenu). A menu in a resource file contains the resource ID of its menu definition procedure. If you store the resource ID of your own menu definition procedure in a menu in a resource file, GetMenu will take care of reading the procedure into memory and storing a handle to it in the menuProc field of the menu record.

The Menu Definition Procedure

The menu definition procedure may be written in Pascal or assembly language; the only requirement is that its entry point be at the beginning. You may choose any name you wish for the procedure. Here's how you would declare one named MyMenu:

```
PROCEDURE MyMenu (message: INTEGER; menu: MenuHandle; menuRect:
                  Rect; hitPt: Point; VAR whichItem: INTEGER);
```

The message parameter identifies the operation to be performed. Its value will be one of the following predefined constants:

```
CONST mDrawMsg   = Ø;   {draw the menu}
      mChooseMsg = 1;   {tell which menu item was chosen and}
                        { highlight it}
      mSizeMsg   = 2;   {calculate the menu's dimensions}
```

The menu parameter indicates the menu that the operation will affect, and menuRect is the rectangle (in global coordinates) in which the menu is located.

The message mDrawMsg tells the menu definition procedure to draw the menu inside menuRect; the grafPort will be set up properly for this. (For details on drawing, see the QuickDraw manual.) The standard menu definition procedure figures out how to draw the menu items by looking in the menu record at the data that defines them; this data is described in detail under "Formats of Resources for Menus" below. For menus of your own definition, you may set up the data defining the menu

items any way you like, or even omit it altogether (in which case all the information necessary to draw the menu would be in the menu definition procedure itself).

(eye)
Be sure that any text in the menu is drawn in the system font.

When the menu definition procedure receives the message mChooseMsg, the hitPt parameter is the point (in global coordinates) where the mouse button was pressed, and the whichItem parameter is the item number of the last item that was chosen from this menu. The procedure should test whether hitPt is inside menuRect and respond accordingly:

- If hitPt is inside menuRect, unhighlight whichItem, highlight the newly chosen item, and return the item number of that item in whichItem.

- If hitPt isn't inside menuRect, unhighlight whichItem and return Ø.

(hand)
When the Menu Manager needs to make a chosen menu item blink, it repeatedly calls the menu definition procedure with the message mChooseMsg, causing the item to be alternately highlighted and unhighlighted.

Finally, the message mSizeMsg tells the menu definition procedure to calculate the horizontal and vertical dimensions of the menu and store them in the menuWidth and menuHeight fields of the menu record.

FORMATS OF RESOURCES FOR MENUS

The resource type for a menu definition procedure is 'MDEF'. The standard menu definition procedure has a resource ID of Ø, so your own such procedures must have resource IDs other than Ø. The resource data is simply the assembled code of the procedure.

Icons in menus must be stored in a resource file under the resource type 'ICON' with resource IDs from 257 to 511. Strings in resource files have the resource type 'STR '--but note that if you follow the recommendation of storing entire menus in resource files, you'll never have to store the strings they contain separately.

The formats of menus and menu bars in resource files are given below.

Menus in a Resource File

The resource type for a menu is 'MENU'. The resource ID must be negative for menus belonging to desk accessories and positive for other menus; it should never be Ø. The resource data for a menu has the format shown below. Once read into memory, this data is stored in a menu record (described earlier in the "Menu Records" section).

Number of bytes	Contents
2 bytes	Menu ID (resource ID of this menu)
2 bytes	Ø; placeholder for menu width
2 bytes	Ø; placeholder for menu height
2 bytes	Resource ID of menu definition procedure
2 bytes	Ø (see comment below)
4 bytes	Same as enableFlags field of menu record
1 byte	Length of following title in bytes
n bytes	Characters of menu title
For each menu item:	
1 byte	Length of following text in bytes
m bytes	Text of menu item
1 byte	Icon number, or Ø if no icon
1 byte	Keyboard equivalent, or Ø if none
1 byte	Character marking menu item, or Ø if none
1 byte	Character style of item's text
1 byte	Ø, indicating end of menu items

The four bytes beginning with the resource ID of the menu definition procedure serve as a placeholder for the handle to the procedure: When GetMenu is called to read the menu from the resource file, it also reads in the menu definition procedure if necessary, and replaces these four bytes with a handle to the procedure. The resource ID of the standard menu definition procedure is:

```
CONST textMenuProc = Ø;
```

The resource data for a nonstandard menu can define menu items in any way whatsoever, or not at all, depending on the requirements of its menu definition procedure. If the appearance of the items is basically the same as the standard, the resource data might be as shown above, but in fact everything following "For each menu item" can have any desired format or can be omitted altogether. Similarly, all bits beyond the first of the enableFlags array may be set and used in any way desired by the menu definition procedure; the first bit applies to the entire menu and must reflect whether it's enabled or disabled.

If your menu definition procedure does use the enableFlags array, menus of that type may contain no more than 31 items (1 per available bit); otherwise, the number of items they may contain is limited only by the amount of room on the screen.

(hand)
See "Using the Toolbox from Assembly Language" for the exact format of the character style byte. *** (Currently

it's in "Using QuickDraw from Assembly Language" in the QuickDraw manual.) ***

(eye)
Menus in resource files must not be purgeable.

Menu Bars in a Resource File

The resource type for the contents of a menu bar is 'MBAR' and the resource data has the following format:

Number of bytes	Contents
2 bytes	Number of menus
For each menu:	
2 bytes	Resource ID of menu

SUMMARY OF THE MENU MANAGER

Constants

```
CONST noMark      = Ø;
      checkMark   = 18;   {check mark}
      applesymbol = 2Ø;   {Apple symbol}

      mDrawMsg    = Ø;    {draw the menu}
      mChooseMsg  = 1;    {tell which item was chosen and highlight it}
      mSizeMsg    = 2;    {calculate the menu's dimensions}

      textMenuProc = Ø;
```

Data Structures

```
TYPE MenuPtr    = ^MenuInfo;
     MenuHandle = ^MenuPtr;
     MenuInfo   = RECORD
                    menuID:      INTEGER;
                    menuWidth:   INTEGER;
                    menuHeight:  INTEGER;
                    menuProc:    Handle;
                    enableFlags: PACKED ARRAY [Ø..31] OF BOOLEAN;
                    menuData:    Str255
                  END;
```

Routines

Initialization and Allocation

```
PROCEDURE InitMenus;
FUNCTION  NewMenu        (menuID: INTEGER; menuTitle: Str255) :
                          MenuHandle;
FUNCTION  GetMenu        (menuID: INTEGER) : MenuHandle;
PROCEDURE DisposeMenu    (menu: MenuHandle);
PROCEDURE AppendMenu     (menu: MenuHandle; data: Str255);
PROCEDURE AddResMenu     (menu: MenuHandle; theType: ResType);
PROCEDURE InsertResMenu  (menu: MenuHandle; theType: ResType; afterItem:
                          INTEGER);
```

Forming the Menu Bar

```
PROCEDURE InsertMenu     (menu: MenuHandle; beforeID: INTEGER);
PROCEDURE DrawMenuBar;
PROCEDURE DeleteMenu     (menuID: INTEGER);
```

```
PROCEDURE ClearMenuBar;
FUNCTION  GetNewMBar    (menuBarID: INTEGER) : Handle;
FUNCTION  GetMenuBar :   Handle;
PROCEDURE SetMenuBar    (menuBar: Handle);
```

Choosing from a Menu

```
FUNCTION  MenuSelect (startPt: Point) : LongInt;
FUNCTION  MenuKey    (ch: CHAR) : LongInt;
PROCEDURE HiliteMenu (menuID: INTEGER);
```

Controlling Items' Appearance

```
PROCEDURE SetItem      (menu: MenuHandle; item: INTEGER; itemString:
                        Str255);
PROCEDURE GetItem      (menu: MenuHandle; item: INTEGER; VAR itemString:
                        Str255);
PROCEDURE DisableItem  (menu: MenuHandle; item: INTEGER);
PROCEDURE EnableItem   (menu: MenuHandle; item: INTEGER);
PROCEDURE CheckItem    (menu: MenuHandle; item: INTEGER; checked:
                        BOOLEAN);
PROCEDURE SetItemIcon  (menu: MenuHandle; item: INTEGER; icon: INTEGER);
PROCEDURE GetItemIcon  (menu: MenuHandle; item: INTEGER; VAR icon:
                        INTEGER);
PROCEDURE SetItemStyle (menu: MenuHandle; item: INTEGER; chStyle: Style);
PROCEDURE GetItemStyle (menu: MenuHandle; item: INTEGER; VAR chStyle:
                        Style);
PROCEDURE SetItemMark  (menu: MenuHandle; item: INTEGER; markChar: CHAR);
PROCEDURE GetItemMark  (menu: MenuHandle; item: INTEGER; VAR markChar:
                        CHAR);
```

Miscellaneous Utilities

```
PROCEDURE SetMenuFlash (menu: MenuHandle; count: INTEGER);
PROCEDURE CalcMenuSize (menu: MenuHandle);
FUNCTION  CountMItems  (menu: MenuHandle) : INTEGER;
FUNCTION  GetMHandle   (menuID: INTEGER) : MenuHandle;
PROCEDURE FlashMenuBar (menuID: INTEGER);
```

Meta-Characters for AppendMenu

Meta-character	Usage
; or Return	Separates multiple items
^	Followed by an icon number, adds that icon to the item
!	Followed by a character, marks the item with that character
<	Followed by B, I, U, O, or S, sets the character style of the item
/	Followed by a character, associates a keyboard equivalent with the item
(	Disables the item

Menu Definition Procedure

```
PROCEDURE MyMenu (message: INTEGER; menu: MenuHandle; menRect: Rect;
                  hitPt: Point; VAR whichItem: INTEGER);
```

Assembly-Language Information

Constants

```
noMark          .EQU    Ø
checkMark       .EQU    18      ;check mark
appleSymbol     .EQU    2Ø      ;Apple symbol

mDrawMsg        .EQU    Ø       ;draw the menu
mChooseMsg      .EQU    1       ;tell which item was chosen and
                                ; highlight it
mSizeMsg        .EQU    2       ;calculate the menu's dimensions
```

Menu Record Data Structure

```
menuID          Menu ID
menuWidth       Menu width
menuHeight      Menu height
menuDefHandle   Handle to menu definition procedure
menuEnable      Enable flags
menuData        Menu title followed by data defining the items
menuBlkSize     Length of all the above fields except menuData
```

Special Macro Names

Routine name	Macro name
DisposeMenu	_DisposMenu
GetItemIcon	_GetItmIcon
GetItemMark	_GetItmMark
GetItemStyle	_GetItmStyle
GetMenu	_GetRMenu
SetItemIcon	_SetItmIcon
SetItemMark	_SetItmMark
SetItemStyle	_SetItmStyle
SetMenuFlash	_SetMFlash

System Globals

Name	Size	Contents
menuList	4 bytes	Handle to current menu list
mBarEnable	2 bytes	Nonzero if menu bar belongs to a desk accessory
menuHook	4 bytes	Hook for routine to be called during MenuSelect
theMenu	2 bytes	Menu ID of currently highlighted menu
menuFlash	2 bytes	Count for duration of menu item blinking

GLOSSARY

character style: A set of stylistic variations, such as bold, italic, and underline. The empty set indicates normal text (no stylistic variations).

dimmed: Drawn in gray rather than black.

disabled: A disabled menu item or menu is one that cannot be chosen; the menu item or menu title appears dimmed.

icon: A 32-by-32 bit image that graphically represents an object, concept, or message.

icon number: A digit from 1 to 9 to which the Menu Manager adds 256 to get the resource ID of an icon associated with a menu item.

keyboard equivalent: A way of invoking a menu item from the keyboard, by holding down the Command key and typing a character.

menu: A list of menu items that appears when the user points to and presses a menu title in the menu bar. Dragging through the menu and releasing over an enabled menu item chooses that item.

menu bar: The horizontal strip at the top of the Macintosh screen that contains the menu titles of all menus in the menu list.

menu definition procedure: A procedure called by the Menu Manager when it needs to perform basic operations on a particular type of menu, such as drawing the menu.

menu ID: For menus defined in resource files, the resource ID of the menu; for application menus, a positive number that you choose to identify the menu.

menu item: A choice in a menu, usually a command to the current application; in a standard Macintosh menu, a line containing text and possibly an icon.

menu item number: The index, starting from 1, of a menu item in a menu.

menu list: A list of menu handles for all menus in the menu bar, kept internally by the Menu Manager.

menu record: The internal representation of a menu, where the Menu Manager stores all the information it needs for its operations on that menu.

menu title: A word or phrase in the menu bar that designates one menu.

meta-character: One of the characters ; ^ ! < / (or Return appearing in the string passed to the Menu Manager routine AppendMenu, to

separate menu items or alter their appearance.

COMMENTS?

Macintosh User Education encourages your comments on this manual.

- What do you like or dislike about it?

- Were you able to find the information you needed?

- Was it complete and accurate?

- Do you have any suggestions for improvement?

Please send your comments to the author (indicated on the cover page) at 10460 Bandley Drive M/S 3-G, Cupertino CA 95014. Mark up a copy of the manual or note your remarks separately. (We'll return your marked-up copy if you like.)

Thanks for your help!

MACINTOSH USER EDUCATION

TextEdit: A Programmer's Guide /TEXT.EDIT/EDIT

See Also: The Macintosh User Interface Guidelines
Macintosh Operating System Manual
QuickDraw: A Programmer's Guide
The Font Manager: A Programmer's Guide
The Window Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
CoreEdit: A Programmer's Guide

Modification History: First Draft (ROM 7) B. Hacker 9/28/83

ABSTRACT

The TextEdit package of the Macintosh User Interface Toolbox is a set of data types and routines for handling basic text formatting and editing capabilities in a Macintosh application. This manual describes TextEdit in detail.

TABLE OF CONTENTS

ABOUT THIS MANUAL

The TextEdit package of the Macintosh User Interface Toolbox is a set of data types and routines for handling basic text formatting and editing capabilities in a Macintosh application. This manual describes TextEdit in detail.

The Toolbox also includes a more sophisticated text editing package, called CoreEdit. You'll need to use CoreEdit instead of TextEdit if you want fully justified text, recognition of word boundaries during editing ("intelligent cut and paste"), or tabbing. Bear in mind, however, that CoreEdit is not in the Macintosh ROM, and occupies over 6K of your application's available memory instead.

(hand)

This manual describes the TextEdit that works with version 7 of the ROM. If you're using a different version, the information presented here may not apply.

Like all documentation about Toolbox units, this manual assumes you're familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's Memory Manager. You should also be familiar with the following:

- The basic concepts and structures behind QuickDraw, particularly points, rectangles, grafPorts, fonts, and character style.

- The ToolBox Event Manager. Some TextEdit routines are called only in response to particular events.

- The Window Manager, particularly update and activate events.

The manual begins with an introduction to TextEdit and what you can do with it. It then discusses the edit record, the primary data structure used by the text editing routines. Learning about this data structure will give you the background you need to understand the routines themselves.

Next, a section on using TextEdit introduces you to its routines and tells how they fit into the flow of your application. This is followed by detailed descriptions of all text editing procedures and functions--their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions is a section containing notes for programmers who will use TextEdit from assembly language.

Finally, there's a summary of the TextEdit data structures and routine calls, for quick reference, and a glossary of terms used in this manual.

ABOUT TEXTEDIT

TextEdit is a group of compact and efficient routines that provide the basic text editing and formatting capabilities needed in an application. These routines perform operations such as:

- Inserting new text
- Deleting characters that are backspaced over
- Translating mouse activity into text selection
- Moving text within a window
- Deleting selected text and possibly inserting it elsewhere, or copying text without deleting it

Because these routines follow the Macintosh User Interface Guidelines, using them ensures that your application presents a consistent, easy-to-learn interface for end users. In particular, TextEdit supports these standard features:

- Selecting text by clicking and dragging with the mouse, double-clicking to select words instead of characters.
- Inverse highlighting of the current text selection, or display of a blinking vertical bar at the insertion point.
- Word wrap, which prevents words from being split between lines when text is drawn. To TextEdit, a word is any series of printing characters, excluding spaces (ASCII code $2Ø) but including nonbreaking spaces (ASCII code $CA).
- Cutting (or copying) and pasting within an application via the Clipboard *** not currently described as "Clipboard" in the User Interface Guidelines, but will be ***. TextEdit puts text you cut or copy into a string of characters called the scrap.

(hand)

Cutting and pasting between applications, or between applications and desk accessories, is done with the aid of the Scrap Manager (see the Scrap Manager manual for details).

THE EDITING ENVIRONMENT: EDIT RECORD

To edit text on the screen, the text editing routines need 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. The data type of an edit record is called TERec.

You prepare to edit text by passing, to a procedure, a destination rectangle in which to draw the text and a view rectangle in which the text will be visible. The procedure incorporates the rectangles and the drawing environment of the current grafPort into an edit record, and returns a handle to the record:

```
TYPE TEPtr    = ^TERec;
     TEHandle = ^TEPtr;
```

Most of the text editing 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
- The current selection range, which determines exactly which characters will be affected by the next editing operation
- The justification of the text, as left, right, or center

The special terms introduced here are described in detail below.

Most programmers won't access any of the fields of an edit record directly, and so don't have to know its exact structure; the necessary access is done with TextEdit routines. Advanced programmers, however, may need to know some of the field names. The structure of an edit record is given below.

The Destination and View Rectangles

The destination rectangle is the rectangle in which the text is drawn. The view rectangle is the rectangle within which the text is actually visible. In other words, the view of the text drawn in the destination rectangle is clipped to the view rectangle (see Figure 1).

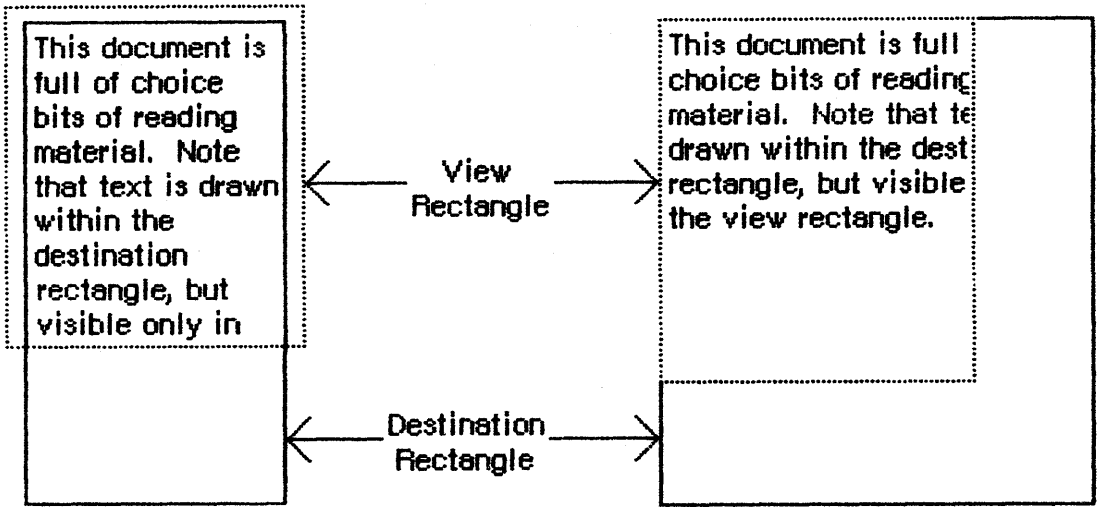


Figure 1. Destination and View Rectangles

You specify both rectangles in the coordinate system of the grafPort. In a document window, the destination rectangle should be inset about four pixels from the left and right edges of the grafPort's portRect (20 pixels from the right edge if there's a scroll bar or size box) to ensure that the first and last character in each line is legible.

Edit operations may of course lengthen or shorten the text. If the text becomes too long to be enclosed by the destination rectangle, it's simply drawn beyond the bottom. In other words, you can think of the destination rectangle as bottomless--its sides determine the beginning and end of each line of text, and its top determines the position of the first line.

Normally, at the right edge of the destination rectangle, the text automatically wraps around to the left edge to begin a new line. A new line also begins where explicitly specified by a Return character in the text. Word wrap ensures that words are never split between lines unless they're too long to fit entirely on one line.

The Selection Range

In the text editing environment, a character position is an index into the text, with position 0 corresponding to the first character. The edit record includes fields for character positions that specify the beginning and end of the current selection range, which is the series of characters at which 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 current selection range.

The selection range, which is always inversely highlighted, extends from the beginning character position up to but NOT including the end position. Figure 2 shows a selection range defined by the beginning

position 2 and the end position 7; it consists of five characters, those at positions 2 through 6. The end position may be 1 greater than the position of the last character of the text, so that the selection range can include the last character.

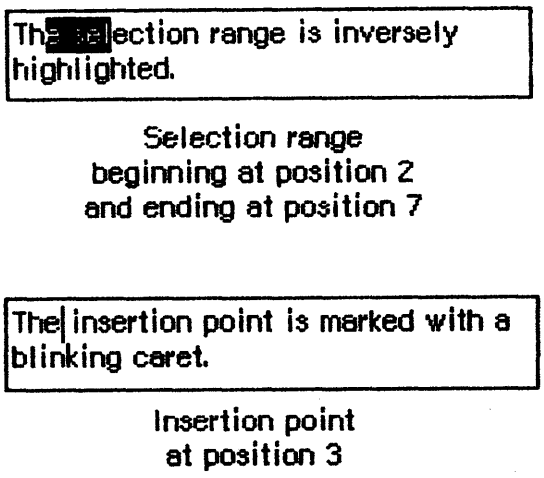


Figure 2. Selection Range and Insertion Point

If the beginning and end of the selection range are the same, that character position is the text's insertion point, the position where characters will be inserted. By convention, it's usually marked with a caret that blinks (is repeatedly inverted). If, for example, the insertion point is as illustrated in Figure 2 and the inserted characters are " edit", the text will read "The edit insertion point...".

(hand)

> We use the word caret here generically, to mean a symbol indicating where something is to be inserted; the specific symbol is a vertical bar. TextEdit does not automatically change the caret to a vertical bar for you. (You must use the QuickDraw procedure SetCursor.)

If you call a procedure to insert characters when there's no insertion point (that is, when there's a selection range of one or more characters), the editing procedure automatically deletes the selection range and replaces it with an insertion point, before inserting the characters.

Justification

TextEdit allows you to specify the justification of the lines of text, that is, their horizontal placement with respect to the left and right edges of the destination rectangle. The different types of justification are illustrated in Figure 3.

- Left justification aligns the text with the left edge of the destination rectangle. This is the default type of justification.
- Center justification centers the text between the left and right edges of the destination rectangle.
- Right justification aligns the text with the right edge of the destination rectangle.

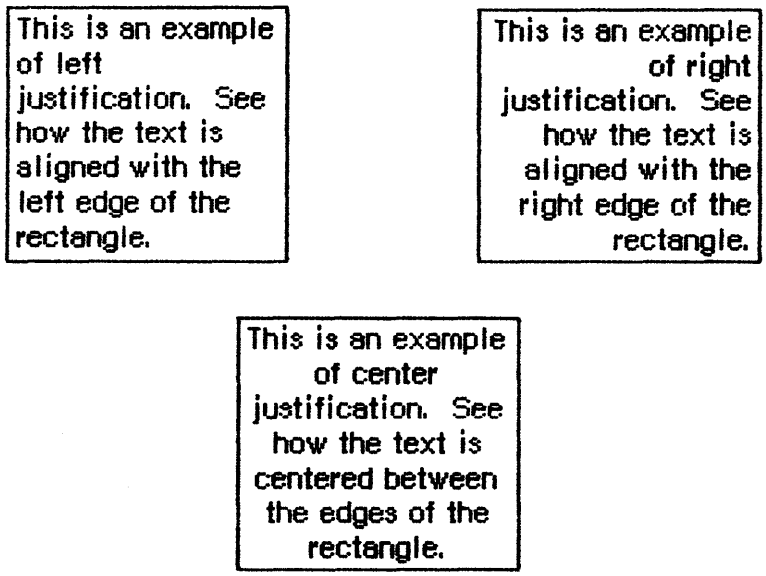


Figure 3. Justification

(hand)
Trailing and leading spaces on a line are ignored for justification. For example, "Fred" and " Fred " will be aligned identically.

TextEdit has three predefined constants for setting the justification:

```
CONST teJustLeft   = 0;
      teJustCenter = 1;
      teJustRight  = -1;
```

The TERec Data Type

For those who want to know more about the structure of an edit record, some (but not all) of the structure is given here. You can skip this section if you want and still use TextEdit as described above, but some TextEdit features are available only if you change fields in the edit record directly.

(eye)
The fields that are not described exist solely for internal use among the text editing routines; their contents cannot be predicted and must not be changed.

```
TYPE TERec = RECORD
               destRect:   Rect;    {destination rectangle}
               viewRect:   Rect;    {view rectangle}
               lineHeight: INTEGER; {line height}
               firstBL:    INTEGER; {location of first base line}
               selStart:   INTEGER; {start of selection range}
               selEnd:     INTEGER; {end of selection range}
               just:       INTEGER; {justification}
               length:     INTEGER; {length of text}
               hText:      Handle;  {text to be edited}
               txFont:     INTEGER; {text font}
               txFace:     INTEGER; {character style}
               txMode:     INTEGER; {pen mode}
               txSize:     INTEGER; {type size}
               inPort:     GrafPtr; {grafPort}
               crOnly:     INTEGER; {new line at Return only, if <Ø}
               nLines:     INTEGER; {number of lines}
               lineStarts: ARRAY [Ø..32ØØØ] OF INTEGER
                                    {positions of line starts}
               {some fields within the record are for internal use
                only, and aren't shown here; see the Pascal interface
                to TextEdit}
             END;
```

Any of the fields in the edit record can be changed, at your discretion. Clearly, some fields (such as length, hText, inPort, and crOnly) might be changed frequently. The lineStarts array should be left unchanged.

The lineHeight field specifies the line height of the text, the number of pixels from the base line of one line to the base line of the next line, as shown in Figure 4. For single-spaced lines, line height is the same as the type size, for double-spaced lines, line height is twice the type size, and so on.

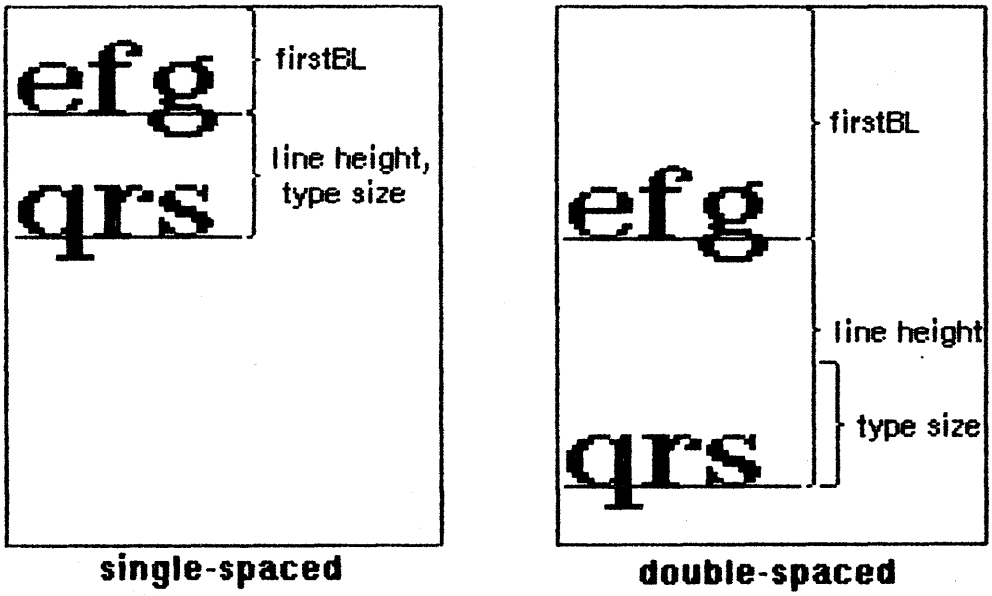


Figure 4. Line Height and FirstBL

The firstBL field specifies the number of pixels from the top of the destination rectangle to the base line of the first line of text. Initially the firstBL field is set for single-spaced lines, but you can change it for any other spacing you want. For example, to change from single to double spacing, use

```
firstBL := firstBL + typeSize
lineHeight := 2 * typeSize
```

where typeSize is the type size of the text.

The hText field is a handle to the text to be edited, and the length field contains the number of characters in the text. You can directly change the text of an edit record by changing these two fields.

The crOnly field specifies whether or not text wraps around at the right edge of the destination rectangle, as shown in Figure 5. If crOnly is zero or positive, text does wrap around. If crOnly is negative, text does not wrap around at the edge of the destination rectangle, and new lines are specified explicitly by Return characters only. This is somewhat faster than wrap around, and is useful in applications such as a programming-language editor, where you don't want a single line of code to be split onto two or more lines.

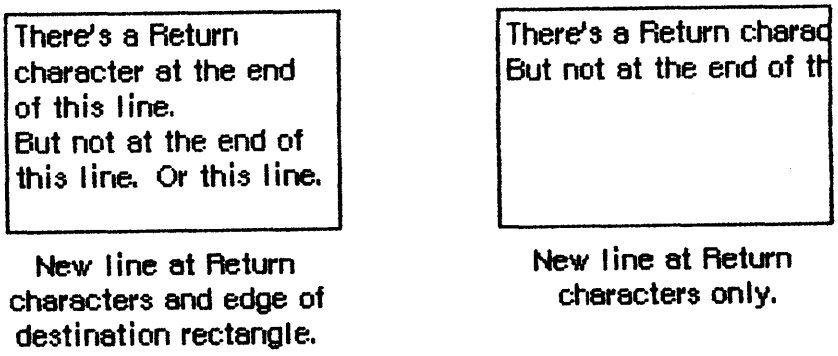


Figure 5. New Lines

The nLines field contains the number of lines in the text. The lineStarts array contains the character position of the first character in each line. It's declared to have 32ØØ1 elements to comply with Pascal range checking; it's actually a dynamic data structure having only as many elements as needed.

(hand)

The values of the lineStarts array, selEnd, and selStart are stored internally as unsigned integers. Be aware that negative values passed from Pascal will be interpreted as greater than 32767.

USING TEXTEDIT

This section discusses how the text editing routines fit into the general flow of an application program and gives you an idea of what routines you'll need to use. The routines themselves are described in detail in the next section.

Before using TextEdit, you should initialize QuickDraw, the Font Manager, and the Window Manager, in that order.

The first TextEdit routine to call is the initialization procedure TEInit. Call TENew to allocate an edit record; it returns a handle to the record. Most of the text editing routines require you to pass this handle as a parameter.

To make a blinking caret appear at the insertion point, call the TEIdle procedure as often as possible; if it's not called often enough, the caret will blink irregularly.

Your application's "main loop" should call the Toolbox Event Manager function GetNextEvent to learn whether any events have occurred. Events that pertain to TextEdit need to be handled by TextEdit routines. Whenever a mouse down event occurs within the view rectangle, call the TEClick procedure. TEClick automatically controls

the placement of the selection range and insertion point (including supporting use of the Shift key to make extended selections).

There are several procedures available for editing text. Usually they are called in response to mouse down events and menu selections. The editing procedures are:

- TEKey inserts characters at the insertion point, and deletes characters backspaced over.

- TECut transfers the selection range to the scrap, removing it from the text, and TEPaste inserts the scrap at the insertion point. By calling TECut, changing the insertion point, and then calling TEPaste, you can perform a "cut and paste" operation, moving text from one place to another.

- TECopy copies the selection range to the scrap. By calling TECopy, changing the insertion point, and then calling TEPaste, you can make multiple copies of text.

- TEDelete removes the selection range (without transferring it to the scrap).

- TEInsert inserts text at the insertion point. You can use this to combine two or more documents. TEDelete and TEInsert do not modify the scrap, and consequently are useful for implementing the Undo command (as described in the Macintosh User Interface Guidelines).

After each editing procedure, the text is redrawn from the insertion point to the end of the destination rectangle. You never have to pass the selection range or insertion point to the editing procedures; the procedures simply access that information from the edit record. The editing procedures and TEClick leave the selection range or insertion point where it should be, according to the Macintosh User Interface Guidelines, so you don't have to set it yourself. But, in case you want to, you can modify the selection range directly by using the TESetSelect procedure.

Every time GetNextEvent reports an update event for the text editing window, call TEUpdate (along with the Window Manager procedures BeginUpdate and EndUpdate), to redraw the text.

(hand)
Advanced programmers: you must call TEUpdate after you change any fields of the edit record if the fields affect the appearance of the text. This ensures that the screen accurately reflects the changed editing environment.

The procedures TEActivate and TEDeactivate must be called each time the Event Manager reports an activate event for the text editing window. TEActivate simply highlights the selection range or displays a caret at the insertion point, and TEDeactivate unhighlights the selection range or removes the caret.

To specify the justification of the text, you can use TESetJust (which requires calling TEUpdate).

If at any time you want to change the text being edited, you can do so by calling TESetText. A common technique (used in dialog boxes, for instance) is to allocate a single edit record for several separate pieces of text where only one may be edited at a time; this saves having to allocate an edit record for each of them.

When you've finished working with the text of an edit record, you can get a handle to the text by calling TEGetText. When you're completely done with an edit record and want to dispose of it, call TEDispose, which removes the text and edit record from the heap.

If you ever want to draw text in any given rectangle (without being able to edit it), use the TextBox procedure.

Advanced programmers may wish to use the TEScroll procedure, to move text within the view rectangle, or TECalText, to recalculate the beginning of each line after changing the text or the destination rectangle.

TEXTEDIT ROUTINES

This section describes all the procedures and functions in TextEdit. They are presented in their Pascal form; for information on using them from assembly language, see "Using the Toolbox from Assembly Language" *** doesn't exist, but see the QuickDraw manual *** and also "Notes for Assembly-Language Programmers" in this manual.

Initialization

```
PROCEDURE TEInit;
```

TEInit initializes TextEdit by allocating a handle for the scrap. The scrap is initially empty. Call this procedure once and only once at the beginning of your program.

```
FUNCTION TENew (destRect,viewRect: Rect) : TEHandle;
```

TENew allocates a handle for the text, builds and initializes an edit record, and returns a handle to the new edit record. DestRect and viewRect are the destination and view rectangles, respectively. Both rectangles are specified in the current grafPort's coordinates. Call this procedure once for every edit record you want allocated. The edit record incorporates the drawing environment of the grafPort, and is initialized for left-justified, single-spaced text with an insertion point at character position Ø.

Manipulating Edit Records

```
PROCEDURE TESetText (text: Ptr; length: LongInt; hTE: TEHandle);
```

TESetText takes the specified text and incorporates it into the edit record specified by hTE. The text parameter points to the text, and the length parameter indicates the number of characters in the text. The selection range is set to an insertion point at the end of the text. TESetText does not affect the text drawn in the destination rectangle, so call TEUpdate (described below) afterwards.

```
FUNCTION TEGetText (hTE: TEHandle) : CharsHandle;
```

TEGetText returns a handle to the text of the edit record specified by hTE. The CharsHandle data type is defined as:

```
TYPE CharsHandle = ^CharsPtr;
     CharsPtr    = ^Chars;
     Chars       = PACKED ARRAY [0..32000] OF CHAR;
```

```
PROCEDURE TEDispose (hTE: TEHandle);
```

TEDispose deallocates the space allocated for the edit record and text specified by hTE, and returns the memory to the free memory pool. Call this procedure when you're completely through with an edit record.

Editing

```
PROCEDURE TEKey (key: CHAR; hTE: TEHandle);
```

TEKey replaces the selection range in the text specified by hTE with the character given by the key parameter, and leaves an insertion point just past the inserted character. If the selection range is an insertion point, TEKey just inserts the character there. If the key parameter contains a Backspace character, the character immediately to the left of the insertion point is deleted. Call TEKey every time the Toolbox Event Manager function GetNextEvent reports a keyboard event that your application decides should be handled by TextEdit.

(eye)

TEKey blindly inserts every character passed in the key parameter, so it's up to your application to filter out all characters that aren't actual text (such as keys typed in conjunction with modifier or special keys).

PROCEDURE TECut (hTE: TEHandle);

TECut removes the selection range from the text specified by hTE and places it in the scrap. Anything previously in the scrap is deleted. (See Figure 6.) If the selection range is an insertion point, the scrap is emptied.

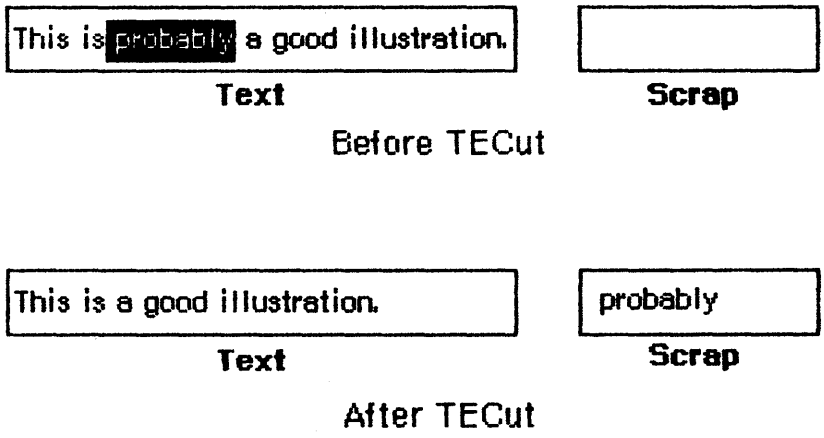


Figure 6. Cutting

PROCEDURE TECopy (hTE: TEHandle);

TECopy copies the selection range from the text specified by hTE into the 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.

PROCEDURE TEPaste (hTE: TEHandle);

TEPaste replaces the selection range in the text specified by hTE with the scrap, and leaves an insertion point just past the inserted text. (See Figure 7.) If the scrap is empty, the selection range is deleted. If the selection range is an insertion point, TEPaste just inserts the scrap there.

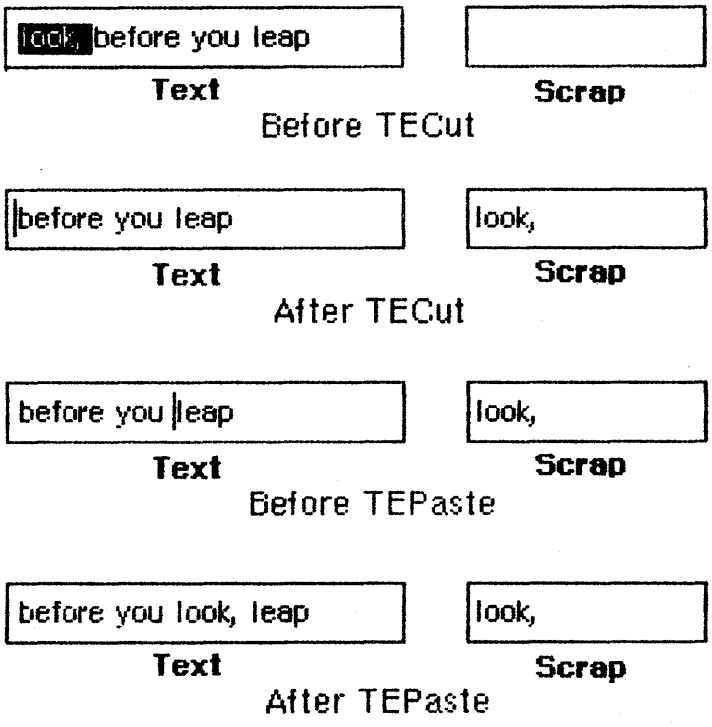


Figure 7. Cutting and Pasting

```
PROCEDURE TEDelete (hTE: TEHandle);
```

TEDelete removes the selection range from the text specified by hTE. It's the same as TECut (above) except that it doesn't transfer the selection range to the scrap. If the selection range is an insertion point, nothing happens.

```
PROCEDURE TEInsert (text: Ptr; length: LongInt; hTE: TEHandle);
```

TEInsert takes the specified text and inserts it, just before the selection range, into the text indicated by hTE. The text parameter points to the inserted text, and the length parameter indicates the number of characters to be inserted.

(eye)

Any current selection range is not deleted. This is different from TEKey and TEPaste, and allows your application to support the Undo command (described in the Macintosh User Interface Guidelines) if you want.

Selection Range and Justification

```
PROCEDURE TESetSelect (selStart,selEnd: LongInt; hTE: TEHandle);
```

TESetSelect unhighlights the current selection range, and changes it to selStart and selEnd in the text specified by hTE. The new selection range is highlighted. If selStart and selEnd are equal, the selection range is an insertion point, and a caret is displayed.

SelEnd and selStart can range from Ø to 65535. If selEnd is anywhere beyond the last character of the text, the position just past the last character is used.

```
PROCEDURE TESetJust (j: INTEGER, hTE: TEHandle);
```

TESetJust changes the justification of the text specified by hTE to j. (See "Justification" under "The Editing Environment: Edit Record".) Call TEUpdate (described below under "Text Display") after TESetJust to cause the text to be redrawn with the new justification.

Mice and Carets

```
PROCEDURE TEClick (pt: Point; extend: BOOLEAN; hTE: TEHandle);
```

TEClick controls the placement and highlighting of the selection range as determined by mouse down events. Call TEClick whenever a mouse down event occurs in the view rectangle of the edit record specified by hTE. Pt is the mouse location (in local coordinates) at the time the button was pressed, obtainable from the event record. Pass TRUE for the extend parameter if the Event Manager indicates that the Shift key was held down at the time of the click (for an extended selection range).

(eye)

> Use the QuickDraw procedure GlobalToLocal to convert the global coordinates of the mouse location given in the event record to the local coordinate system for pt.

If the mouse moves, meaning that a drag is occurring, the selection range expands or shrinks accordingly. The current selection range is unhighlighted. In the case of a double click, meaning that word selection has been chosen, the word under the cursor becomes the selection range.

```
PROCEDURE TEIdle (hTE: TEHandle);
```

Call TEIdle repeatedly to make a blinking caret appear at the insertion point, if any, in the text specified by hTE. TextEdit observes a minimum blink interval: no matter how often you call TEIdle, the time between blinks will never be less than the minimum interval. You should call this procedure as often as possible to provide a constant frequency of blinking.

(hand)
The initial minimum blink interval setting is 4 ticks (sixtieths of a second). The user can adjust this setting to individual preference with the control panel desk accessory.

```
PROCEDURE TEActivate (hTE: TEHandle);
```

TEActivate highlights the selection range in the view rectangle of the edit record specified by hTE. If the selection range is an insertion point, it displays a caret there. This procedure should be called every time the Toolbox Event Manager function GetNextEvent reports that the text editing window has become active.

```
PROCEDURE TEDeactivate (hTE: TEHandle);
```

TEDeactivate unhighlights the selection range in the view rectangle of the edit record specified by hTE. If the selection range is an insertion point, it removes the caret. This procedure should be called every time the Toolbox Event Manager function GetNextEvent reports that the text editing window has become inactive.

Text Display

```
PROCEDURE TEUpdate (rUpdate: Rect; hTE: TEHandle);
```

TEUpdate draws the text specified by hTE within the rectangle specified by rUpdate. The location of the rUpdate rectangle must be given in the coordinates of the grafPort. Call TEUpdate every time the Toolbox Event Manager function GetNextEvent reports an update event--after you call the Window Manager procedure BeginUpdate, and before you call EndUpdate.

Normally you'll use the following when an update event occurs:

```
BeginUpdate(myWindow);
TEUpdate(myWindow^.visRgn^^.rgnBBox, hTE);
EndUpdate(myWindow);
```

Instead of passing rUpdate as shown, you can pass the viewRect, but doing so may result in unnecessary drawing.

```
PROCEDURE TextBox (text: Ptr; length: LongInt; box: Rect; j: INTEGER);
```

TextBox draws the specified text in the rectangle indicated by the box parameter, with justification j. (See "Justification" under "The Editing Environment: Edit Record".) The text parameter points to the text, and the length parameter indicates the number of characters to draw. TextBox does not create an edit record, nor can the text that it draws be edited immediately; it's used solely for drawing text. For example:

```
str := 'Planning Procedures';
SetRect(r, 1ØØ, 1ØØ, 2ØØ, 2ØØ);
TextBox(@str[1], LENGTH(str), r, teFillCenter);
FrameRect(r);
```

Advanced Routines

These routines are useful only if you're directly accessing the fields of an edit record.

```
PROCEDURE TEScroll (dh,dv: INTEGER; hTE: TEHandle);
```

TEScroll moves ("scrolls") the text within the view rectangle of the specified edit record by the number of pixels specified in the dh and dv parameters. The edit record is specified by the hTE parameter. Positive dh and dv values move the text right and down, respectively, and negative values move the text left and up. For example,

```
TEScroll(Ø, -lnHeight, hTE)
```

scrolls the text up one line (where lnHeight is the value of the lineHeight field in the edit record).

```
PROCEDURE TECalText (hTE: TEHandle);
```

TECalText recalculates the beginnings of all lines of text in the edit record specified by hTE, updating elements of the lineStarts array. Call TECalText if you've changed the destination rectangle, the hText field, or anything else that effects the number of characters per line.

(hand)

There really are two ways to specify text to be edited. The easiest, direct method is to use TESetText, which takes an existing edit record, creates a second copy of its text, and makes the edit record point to the copy. An advanced, indirect method is to change the hText field

of the edit record directly, and then call TECalText to recalculate the lineStarts array to match the new text. If you have a lengthy text to edit, use the latter method to save space because it doesn't create a copy.

NOTES FOR ASSEMBLY-LANGUAGE PROGRAMMERS

Information about how to use the User Interface Toolbox from assembly language is given elsewhere *** (currently, in the QuickDraw manual) ***. This section contains special notes of interest to programmers who will be using TextEdit from assembly language.

If you use .INCLUDE to include a file named ToolEqu.Text when you assemble your program, the TextEdit constants and offsets into the fields of structured types in TextEdit will be available in symbolic form.

There are hooks within TextEdit that allow you insert additional procedures for more sophisticated editing. They require some care because they pass arguments in registers and it's the application's responsibility to save and restore the registers.

Two of the hooks are TEHiHook and TECarHook. If you install a nonzero address in either of these hooks, that address (instead of _InverRect) will be jumped to when a selection range is to be highlighted. The routine called can destroy the contents of the registers AØ, A1, DØ, D1, and D2. A3 will be pointing to a locked edit record, and teSelRect(A3) contains the rectangle enclosing the text being highlighted. For example, the following assembly-language fragment draws underlined selection ranges:

```
UnderHigh
        MOVE.L  (SP)+,AØ                ;point to rectangle to be
                                        ;  highlighted
        MOVE    top(AØ),-(SP)           ;save existing top coordinate
        MOVE    bottom(AØ),top(AØ)      ;make the top coordinate equal
        SUBQ    #1,top(SP)              ;  the bottom coordinate - 1
        MOVE.L  AØ,-(SP)                ;invert the resulting
        _InverRect                      ;   rectangle
        MOVE    (SP)+,teSelRect+top(A3) ;restore original top coordinate
        RTS
```

Note that the rectangle must be preserved.

TECarHook acts analogously upon insertion points instead of selection ranges. It must be called with teSelRect containing the insertion point rectangle.

*** The explanation of the other hooks is forthcoming. ***

SUMMARY OF TEXTEDIT

```
CONST teJustLeft   = Ø;
      teJustCenter = 1;
      teJustRight  = -1;

TYPE CharsHandle = ^CharsPtr;
     CharsPtr    = ^Chars;
     Chars       = PACKED ARRAY [Ø..32ØØØ] OF CHAR;

     TEPtr    = ^TERec;
     TEHandle = ^TEPtr;
     TERec    = RECORD
                  destRect:   Rect;    {destination rectangle}
                  viewRect:   Rect;    {view rectangle}
                  lineHeight: INTEGER; {line height}
                  firstBL:    INTEGER; {position of first base line}
                  selStart:   INTEGER; {start of selection range}
                  selEnd:     INTEGER; {end of selection range}
                  just:       INTEGER; {justification}
                  length:     INTEGER; {length of text}
                  hText:      Handle;  {text to be edited}
                  txFont:     INTEGER; {text font}
                  txFace:     INTEGER; {character style}
                  txMode:     INTEGER; {pen mode}
                  txSize:     INTEGER; {type size}
                  inPort:     GrafPtr; {grafPort}
                  crOnly:     INTEGER  {new line at Return only, if <Ø}
                  nLines:     INTEGER; {number of lines}
                  lineStarts: ARRAY [Ø..32ØØØ] OF INTEGER;
                                       {positions of lines starts}
                  {more fields for internal use only}
                END;
```

Initialization

```
PROCEDURE TEInit;
FUNCTION  TENew   (destRect,viewRect: Rect) : TEHandle;
```

Manipulating Edit Records

```
PROCEDURE TESetText (text: Ptr; length: LongInt; hTE: TEHandle);
FUNCTION  TEGetText (hTE: TEHandle) : CharsHandle;
PROCEDURE TEDispose (hTE: TEHandle);
```

Editing

```
PROCEDURE TEKey    (key: CHAR; hTE: TEHandle);
PROCEDURE TECut    (hTE: TEHandle);
PROCEDURE TECopy   (hTE: TEHandle);
PROCEDURE TEPaste  (hTE: TEHandle);
PROCEDURE TEDelete (hTE: TEHandle);
PROCEDURE TEInsert (text: Ptr; length: LongInt; hTE: TEHandle);
```

Selection Range and Justification

```
PROCEDURE TESetSelect (selStart,selEnd: LongInt; hTE: TEHandle);
PROCEDURE TESetJust   (j: INTEGER; hTE: TEHandle);
```

Mice and Carets

```
PROCEDURE TEClick      (pt: Point; extend: BOOLEAN; hTE: TEHandle);
PROCEDURE TEIdle       (hTE: TEHandle);
PROCEDURE TEActivate   (hTE: TEHandle);
PROCEDURE TEDeactivate (hTE: TEHandle);
```

Text Display

```
PROCEDURE TEUpdate (rUpdate: Rect; hTE: TEHandle);
PROCEDURE TextBox  (text: Ptr; length: LongInt; box: Rect; j: INTEGER);
```

Advanced Routines

```
PROCEDURE TEScroll  (dh,dv: INTEGER; hTE: TEHandle);
PROCEDURE TECalText (hTE: TEHandle);
```

GLOSSARY

caret: A generic term meaning a symbol that indicates where something should be inserted in text. The specific symbol used is a vertical bar.

character position: An index into an array containing text, starting at Ø for the first character.

destination rectangle: In TextEdit, the rectangle in which the text is drawn.

edit record: A complete editing environment, including the text to be edited, the grafPort and rectangle in which to display the text, the arrangement of the text within the rectangle, and other editing and display information.

insertion point: An empty selection range; the character position where text will be inserted (marked with a blinking caret by convention).

justification: The horizontal placement of lines of text relative to the edges of the rectangle in which the text is drawn.

line height: The number of pixels from the base line of one line of text to the base line of the next line of text.

nonbreaking space: The character with ASCII code $CA; drawn as a blank, but interpreted as a nonblank character for the purposes of word wrap.

scrap: A string consisting of the characters most recently cut or copied from text by certain TextEdit routines.

selection range: The series of characters (inversely highlighted), or the character position (marked with a blinking caret), at which the next editing operation will occur.

view rectangle: In TextEdit, the rectangle within which the text is visible.

word: In TextEdit, any series of printing characters, excluding spaces (ASCII code $2Ø) but including nonbreaking spaces (ASCII code $CA).

word wrap: Keeping any series of printing characters intact between lines when text is drawn.

COMMENTS?

Macintosh User Education encourages your comments on this manual.

- What do you like or dislike about it?

- Were you able to find the information you needed?

- Was it complete and accurate?

- Do you have any suggestions for improvement?

Please send your comments to the author (indicated on the cover page) at 10460 Bandley Drive M/S 3-G, Cupertino CA 95014. Mark up a copy of the manual or note your remarks separately. (We'll return your marked-up copy if you like.)

Thanks for your help!

MACINTOSH USER EDUCATION

The Dialog Manager: A Programmer's Guide /DMGR/DIALOG

See Also: Macintosh User Interface Guidelines
QuickDraw: A Programmer's Guide
The Font Manager: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Window Manager: A Programmer's Guide
The Control Manager: A Programmer's Guide
The Desk Manager: A Programmer's Guide
TextEdit: A Programmer's Guide
Programming Macintosh Applications in Assembly Language

Modification History:

Preliminary Draft	Caroline Rose	12/8/82
Preliminary Draft	Caroline Rose	1/7/83
First Draft (ROM 2.1)	Caroline Rose	3/22/83
Second Draft (ROM 4)	Caroline Rose	6/13/83
Third Draft (ROM 7)	Caroline Rose	11/16/83
Fourth Draft	Caroline Rose	7/6/84

ABSTRACT

The Dialog Manager is the part of the Macintosh User Interface Toolbox that supports dialog boxes and the alert mechanism. This manual tells you how to manipulate dialogs and alerts with Dialog Manager routines.

Summary of significant changes and additions since last draft:

- EditText and statText items can't be more than 241 characters long.
- A new procedure, SetDAFont, enables Pascal programmers to change the font used in dialogs and alerts (page 19).
- There are two new procedures, CouldDialog and FreeDialog, that are analogous to CouldAlert and FreeAlert (page 23).
- The description of IsDialogEvent now deals with handling keyboard equivalents of commands when a modeless dialog box is up (page 25). For Pascal programmers, there are also four new routines for handling standard editing commands in modeless dialogs (page 26).
- For Pascal programmers, there are now routines for checking the stage of an alert and setting an alert back to its first stage (page 32).

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Dialog Manager of the Macintosh User Interface Toolbox. *** Eventually it will become part of the comprehensive Inside Macintosh manual. *** The Dialog Manager provides Macintosh programmers with routines for implementing dialog boxes and the alert mechanism, two means of communication between the application and the end user.

Like all documentation about Toolbox units, this manual assumes you're familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's Memory Manager. You should also be familiar with the following:

- resources, as discussed in the Resource Manager manual
- the basic concepts and structures behind QuickDraw, particularly rectangles, grafPorts, and pictures
- the Toolbox Event Manager, the Window Manager, and the Control Manager
- TextEdit, to understand editing text in dialog boxes

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly-language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it.

The manual begins with an introduction to the Dialog Manager and what you can do with it. It then discusses the basics of dialogs and alerts: their relationship to windows and resources, and the information stored in memory for the items in a dialog or alert. Following this is a discussion of dialog records, where the Dialog Manager keeps all the information it needs about a dialog, and an overview of how alerts are handled.

Next, a section on using the Dialog Manager introduces its routines and tells how they fit into the flow of your application program. This is followed by detailed descriptions of all Dialog Manager procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that will not interest all readers. There's a discussion of how to modify definitions of dialogs and alerts after they've been read from a resource file, and a section that gives the exact formats of resources related to dialogs and alerts.

Finally, there's a summary of the Dialog Manager, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE DIALOG MANAGER

The Dialog Manager is a tool for handling dialogs and alerts in a way that's consistent with the Macintosh User Interface Guidelines.

A dialog box appears on the screen when a Macintosh application needs more information to carry out a command. As shown in Figure 1, it typically resembles a form on which the user checks boxes and fills in blanks.

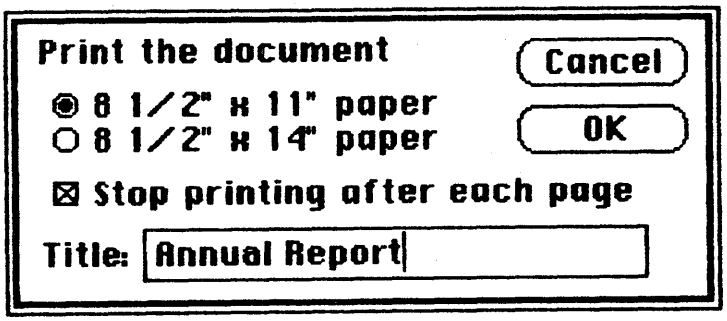


Figure 1. A Typical Dialog Box

By convention, a dialog box comes up slightly below the menu bar, is a bit 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 pictures)
- anything else, as defined by the application

The user provides the necessary information in the dialog box, such as by entering text or clicking a check box. There's usually a button marked "OK" to tell the application to accept the information provided and perform the command, and a button marked "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 manual will still refer to the button as the "OK button". There may even be more than one button that will perform the 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 go away; clicking outside the dialog box only causes a beep from the Macintosh's speaker. This type 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. It usually has the same general appearance as shown in

Figure 1. One of the buttons in the dialog box may be outlined boldly. Pressing the Return key or the Enter key has the same effect as clicking the outlined button or, if none, 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 there's no boldly outlined or OK button, pressing Return or Enter will by convention have no effect.

Other dialog boxes do not require the user to respond before doing anything else; these are called modeless dialog boxes (Figure 2). The user can, for example, do 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 go away: the box will stay around 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 would be useful for long printing or searching operations, for example.

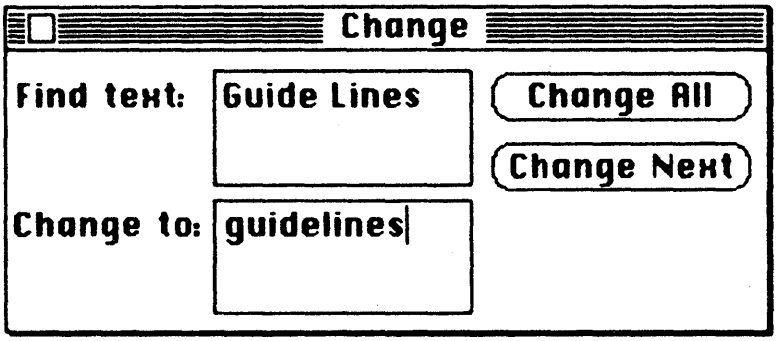


Figure 2. A Modeless Dialog Box

As shown in Figure 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 go away, you can click its close box or choose Close from the File menu when it's the active window.

Dialog boxes may in fact require no response at all. For example, while 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, it 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 it appears only when something has gone wrong or must be brought to the user's attention. Its conventional placement is slightly farther below the menu bar than a dialog box. To assist 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 boldly so it stands out from the other buttons in the alert box (see Figure 3). The outlined button is also the alert's default button; if the user presses the Return key or the Enter key, the effect is the same as

clicking this button.

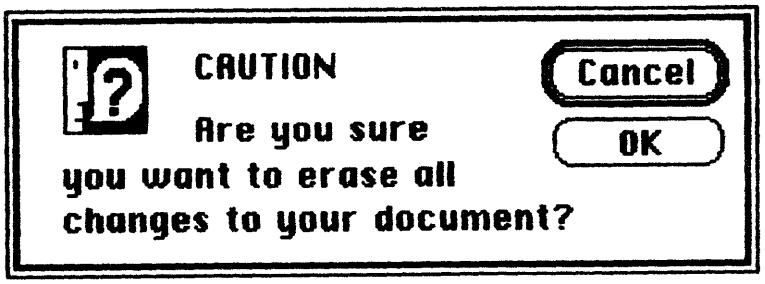


Figure 3. A Typical Alert Box

There are three standard kinds of alerts--Stop, Note, and Caution--each indicated by a particular icon in the top left corner of the alert box. Figure 3 illustrates a Caution alert. The icons identifying Stop and Note alerts are similar; instead of a question mark, they show an exclamation point and an asterisk, respectively. Other alerts can have anything in the the top left corner, including blank space if desired.

The alert mechanism also provides another type of signal: sound from the Macintosh's 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 is not 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, there can also be a change in which button is the default button (perhaps from OK to Cancel). You can specify different responses for up to four occurrences of the same alert.

With Dialog Manager routines, you can create dialog boxes or invoke alerts. The Dialog Manager gets most of the descriptive information about the dialogs and alerts from resources in a resource file. You use a program such as the Resource Editor to store the necessary information in the resource file *** (Resource Editor doesn't exist yet; for now, use the Resource Compiler) ***. The Dialog Manager calls the Resource Manager to read what it needs from the resource file into memory as necessary. In some cases you can modify the information after it's been read into memory.

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 information as when you create a window with a Window Manager routine. For example, you supply the window definition ID, which determines how the window looks and behaves, and a rectangle that becomes the portRect of the window's grafPort. You specify the window's plane (which, by convention, should initially be the frontmost) and whether the window is visible or invisible. The dialog window is created as specified.

You can manipulate a dialog window just like any other window with Window Manager or QuickDraw routines, showing it, hiding it, moving it, changing its size or plane, or whatever--all, of course, in conformance with the Macintosh User Interface Guidelines. The Dialog Manager observes the clipping region of the dialog window's grafPort, so if you want clipping to occur, you can set this region with a QuickDraw routine.

Similarly, 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 ID, so that all alert windows will have the standard appearance and behavior. The size and location of the box are supplied as part of the definition of the alert and are not easily changed. You don't specify the alert window's plane; it always comes up 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 go away, the application doesn't do any manipulation of the alert window.

Figure 4 illustrates a document window, dialog window, and alert window, all overlapping on the desktop.

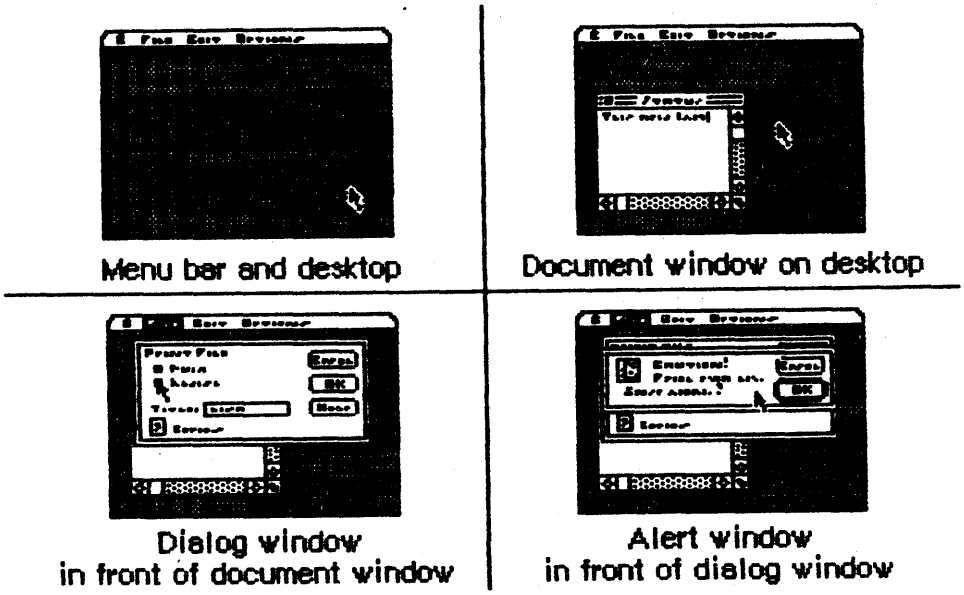


Figure 4. Dialog and Alert Windows

DIALOGS, ALERTS, AND RESOURCES

To create a dialog, the Dialog Manager needs the same information about the dialog window as the Window Manager needs when it creates a new window: the window definition ID along with other information specific to this window. The Dialog Manager also needs to know what items the dialog box contains. You can store the needed information as a resource in a resource file and pass the resource ID to a function that

will create the dialog. This type of resource, which is called a dialog template, is analogous to a window template, and the function, GetNewDialog, is similar to the Window Manager function GetNewWindow. The Dialog Manager calls the Resource Manager to read the dialog template from the resource file. It then incorporates the information in the template into a dialog data structure in memory, called a dialog record.

Similarly, the data that the Dialog Manager needs to create an alert is stored in an alert template in a resource file. The various routines for invoking alerts require the resource ID of the alert template as a parameter.

The information about all the items (text, controls, or graphics) in a dialog or alert box is stored in an item list in a resource file. The resource ID of the item list is included in the dialog or alert template. The item list in turn contains the resource IDs of any icons or QuickDraw pictures in the dialog or alert box, and possibly the resource IDs of control templates for controls in the box. After calling the Resource Manager to read a dialog or alert template into memory, the Dialog Manager calls it again to read in the item list. It then makes a copy of the item list and uses that copy; for this reason, item lists should always be purgeable resources. Finally, the Dialog Manager calls the Resource Manager to read in any individual items as necessary.

(note)

> To create dialog or alert templates and item lists and store them in resource files, you can use the Resource Editor *** (eventually; for now, the Resource Compiler) ***. The Resource Editor relieves you of having to know the exact format of these resources, but for interested programmers this information is given in the section "Formats of Resources for Dialogs and Alerts".

If desired, the application can gain some additional flexibility by calling the Resource Manager directly to read templates, item lists, or items from a resource file. For example, you can read in a dialog or alert template directly and modify some of the information in it before calling the routine to create the dialog or alert. Or, as an alternative to using a dialog template, you can read in a dialog's item list directly and then pass a handle to it along with other information to a function that will create the dialog (NewDialog, analogous to the Window Manager function NewWindow).

(note)

> The use of dialog templates is recommended wherever possible; like window templates, they isolate descriptive information from your application code for ease of modification or translation to foreign languages.

ITEM LISTS IN MEMORY

This section discusses the contents of an item list once it's been read into memory from a resource file and the Dialog Manager has set it up as necessary to be able to work with it.

An item list in memory contains the following information for each item:

- The type of item. This includes not only whether the item is a control, text, or whatever, but also whether the Dialog Manager should return to the application when the item is clicked.
- A handle to the item or, for special application-defined items, a pointer to a procedure that draws the item.
- A display rectangle, which determines the location of the item within the dialog or alert box.

These are discussed below along with item numbers, which identify particular items in the item list.

There's a Dialog Manager procedure that, given a pointer to a dialog record and an item number, sets or returns that item's type, handle (or procedure pointer), and display rectangle.

Item Types

The item type is specified by a predefined constant or combination of constants, as listed below. Figure 5 illustrates some of these item types.

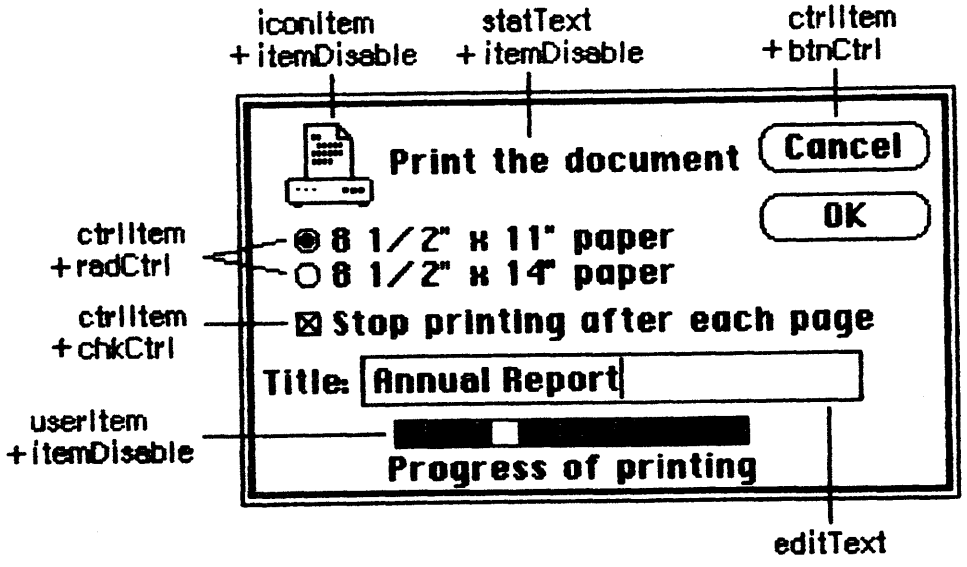


Figure 5. Item Types

Item type	Meaning
ctrlItem+btnCtrl	A standard button control.
ctrlItem+chkCtrl	A standard check box control.
ctrlItem+radCtrl	A standard "radio button" control.
ctrlItem+resCtrl	A control defined in a control template in a resource file.
statText	Static text; text that cannot be edited.
editText	(Dialogs only) Text that can be edited; the Dialog Manager accepts text typed by the user and allows editing.
iconItem	An icon (a 32-by-32 bit image).
picItem	A QuickDraw picture.
userItem	(Dialogs only) An application-defined item, such as a picture whose appearance changes.
itemDisable+<any of the above>	The item is disabled (the Dialog Manager doesn't report events involving this item).

(warning)
StatText and editText items must not be more than 241 characters long.

The text of an editText item may initially be either default text or empty. Text entry and editing is handled in the conventional way, as in TextEdit--in fact, the Dialog Manager calls TextEdit to handle it:

- Clicking in the item displays a blinking vertical bar, indicating an insertion point where text may be entered.
- Dragging over text in the item selects that text, and double-clicking selects a word; the selection is inverted and is replaced by what the user then types.
- Clicking or dragging while holding down the Shift key extends or shortens the current selection.
- The Backspace key deletes the current selection or the character preceding the insertion point.

The Tab key advances to the next editText item in the item list (wrapping around to the first if there aren't any more). In an alert box or a modal dialog box (regardless of whether it contains an editText item), the Return key or Enter key has the same effect as clicking the default button; for alerts, the default button is identified in the alert template, whereas for modal dialogs it's always

the first item in the item list.

If itemDisable is specified for an item, the Dialog Manager doesn't let the application know about events involving that item. For example, you may not have to be informed every time the user types a character or clicks in an editText item, but may only need to look at the text when the OK button is clicked. In this case, the editText item would be disabled. Standard buttons and check boxes should always be enabled, so your application will know when they've been clicked.

(warning)
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.

Item Handle or Procedure Pointer

The item list contains the following information for the various types of items:

Item type	Contents
any ctrlItem	A control handle
statText	A handle to the text
editText	A handle to the current text
iconItem	A handle to the icon
picItem	A picture handle
userItem	A procedure pointer

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. The procedure must have two parameters, a window pointer and an item number. For example, this is how it would be declared if it were named MyItem:

```
PROCEDURE MyItem (theWindow: WindowPtr; itemNo: INTEGER);
```

TheWindow is a pointer to the dialog window; in case the procedure draws in more than one dialog window, this parameter tells it which one to draw in. ItemNo is the item number; in case the procedure draws more than one item, this parameter tells it which one to draw.

Display Rectangle

Each item in the item list is displayed within its display rectangle:

- For controls, the display rectangle becomes the control's enclosing rectangle.

- For an editText item, it becomes TextEdit's destination rectangle and view rectangle. Word wrap occurs, and the text is clipped if

there's more than will fit in the rectangle. In addition, the Dialog Manager uses the QuickDraw procedure FrameRect to draw a rectangle three pixels outside the display rectangle.

- StatText items are displayed in exactly the same way as editText items, except that a rectangle isn't drawn outside the display rectangle.

- Icons and QuickDraw pictures are scaled to fit the display rectangle. For pictures, the Window Manager calls the QuickDraw procedure DrawPicture and passes it 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 of that item.

By giving an item a display rectangle that's off the screen, you can make the item invisible. This might be useful, for example, if your application needs to display a number of dialog boxes that are similar except that one item is 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 remove an item or make one reappear, you just change its display rectangle (and call the Window Manager procedure InvalRect to accumulate the changed area into the dialog window's update region). The QuickDraw procedure OffsetRect is convenient for moving an item off the screen and then on again later. Note the following, however:

- You shouldn't make an editText item invisible, because it may cause strange things to happen. If one of several editText items is invisible, for example, pressing the Tab key may make the insertion point disappear. However, if you do make this type of item invisible, remember that the changed area includes the rectangle that's three pixels outside the item's display rectangle.

- 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 2Ø pixels wide.

- To change text in a statText item, it's easier to use the Dialog Manager procedure ParamText (as described later in the "Dialog Manager Routines" section).

Item Numbers

Each item in an item list is identified by an item number, which is simply the index of the item in the list (starting from 1). By convention, the first item in an alert's item list should be the OK button (or, if none, then one of the buttons that will perform the command) and the second item should be the Cancel button. The Dialog Manager provides predefined constants equal to the item numbers for OK and Cancel:

```
CONST OK     = 1;
      Cancel = 2;
```

In a modal dialog's item list, the first item is assumed to be the dialog's default button; if the user presses the Return key or Enter key, the Dialog Manager normally returns item number 1, just as when that item is actually clicked. To conform to the Macintosh User Interface Guidelines, the application should boldly outline the dialog's default button if it isn't the OK button. The best way to do this is with a userItem. To allow for changes in the default button's size or location, the userItem should identify which button to outline by its item number and then use that number to get the button's display rectangle. The following QuickDraw calls will outline the rectangle in the standard way:

```
PenSize(3,3);
InsetRect(displayRect,-4,-4);
FrameRoundRect(displayRect,16,16)
```

(warning)
If the first item in a modal dialog's item list isn't an OK button and you don't boldly outline it, you should set up the dialog to ignore Return and Enter. To learn how to do this, see ModalDialog under "Handling Dialog Events" in the "Dialog Manager Routines" section.

DIALOG RECORDS

To create a dialog, you pass information to the Dialog Manager in a dialog template and in individual parameters, or only in parameters; in either case, the Dialog Manager incorporates the 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 function NewWindow and then setting the window class in the window record to indicate that it's a dialog window. The routine that creates the dialog returns a pointer to the dialog record, which you use thereafter to refer to the dialog in Dialog Manager routines or even in Window Manager or QuickDraw routines (see "Dialog Pointers" below). The Dialog Manager provides routines for handling events in the dialog window and disposing of the dialog when you're done.

The data type for a dialog record is called DialogRecord. You can do all the necessary operations on a dialog without accessing the fields of the dialog record directly; for advanced programmers, however, the exact structure of a dialog record is given under "The DialogRecord Data Type" below.

Dialog Pointers

There are two types of dialog pointer, DialogPtr and DialogPeek, analogous to the window pointer types WindowPtr and WindowPeek. Most programmers will only need to use DialogPtr.

The Dialog Manager defines the following type of dialog pointer:

```
TYPE DialogPtr = WindowPtr;
```

It can do this because the first field of a dialog record contains the window record for the dialog window. This type of pointer can be used to access fields of the window record or can be passed to Window Manager routines that expect window pointers as parameters. Since the WindowPtr data type is itself defined as GrafPtr, this type of dialog pointer can also be used to access fields of the dialog window's grafPort or passed to QuickDraw routines that expect pointers to grafPorts as parameters.

For programmers who want to access dialog record fields beyond the window record, the Dialog Manager also defines the following type of dialog pointer:

```
TYPE DialogPeek = ^DialogRecord;
```

Assembly-language note: From assembly language, of course, there's no type checking on pointers, and the two types of pointer are equal.

The DialogRecord Data Type

For those who want to know more about the data structure of a dialog record, the exact structure is given here.

```
TYPE DialogRecord = RECORD
                      window:    WindowRecord; {dialog window}
                      items:     Handle;       {item list}
                      textH:     TEHandle; {current editText item}
                      editField: INTEGER;  {editText item number minus 1}
                      editOpen:  INTEGER;  {used internally}
                      aDefItem:  INTEGER   {default button item number}
                    END;
```

The window field contains the window record for the dialog window. The items field contains a handle to the item list used for the dialog. (Remember that after reading an item list from a resource file, the Dialog Manager makes a copy of it and uses that copy.)

(note)

To get or change information about an item in a dialog, you pass the dialog pointer and the item number to a Dialog Manager procedure. You'll never access information directly through the handle to the item list.

The Dialog Manager uses the next three fields when there are one or more editText items in the dialog. If there's more than one such item, these fields apply to the one that currently is selected or displays the insertion point. The textH field contains the handle to the edit record used by TextEdit. EditField is 1 less than the item number of the current editText item, or -1 if there's no editText item in the dialog. The editOpen field is used internally by the Dialog Manager.

(note)

Actually, a single edit record is shared by all editText items; any changes you make to it will apply to all such items. See the TextEdit manual for details about what kinds of changes you can make.

The aDefItem field is used for modal dialogs and alerts, which are treated internally as special modal dialogs. It contains the item number of the default button. The default button for a modal dialog is the first item in the item list, so this field contains 1 for modal dialogs. The default button for an alert is specified in the alert template; see the following section for more information.

Assembly-language note: The global constant dWindLen equals the length of a dialog record in bytes.

ALERTS

When you call a Dialog Manager routine to invoke an alert, you pass it the resource ID of the alert template, which contains the following:

- A rectangle, given in global coordinates, which determines the alert window's size and location. It becomes the portRect of the window's grafPort. To allow for the menu bar and the border around the portRect, the top coordinate of the rectangle should be at least 25 points below the top of the screen.

- The resource ID of the item list for the alert.

- Information about exactly what should happen at each stage of the alert.

Every alert has four stages, corresponding to consecutive occurrences of the alert: the first three stages correspond to the first three occurrences, while the fourth stage includes the fourth occurrence and any beyond the fourth. (The Dialog Manager compares the current alert's resource ID to the last alert's resource ID to determine whether it's the same alert.) The actions for each stage are specified by the following three pieces of information:

- which is the default button--the OK button (or, if none, a button that will perform the command) or the Cancel button
- whether the alert box is to be drawn
- which of four sounds should be emitted at this stage of the alert

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 Ø to 3; it can emit any sound for each of these numbers, which identify the sounds in the alert template. For example, you might declare a sound procedure named MySound as follows:

```
PROCEDURE MySound (soundNo: INTEGER);
```

If you don't write your own sound procedure, the Dialog Manager uses the standard one: sound number Ø represents no sound and sound numbers 1 through 3 represent the corresponding number of short beeps, each of the same pitch and duration. The volume of each beep depends on the current speaker volume setting, which the user can adjust with the Control Panel desk accessory. If the user has set the speaker volume to Ø, the menu bar will blink in place of each beep.

For example, if the second stage of an alert is to cause a beep and no alert box, you can just specify the following for that stage in the alert template: don't draw the alert box, and use sound number 1. If instead you want, say, two successive beeps of different pitch, you need to write a procedure that will emit that sound for a particular sound number, and specify that number in the alert template. The Macintosh Operating System includes routines for emitting sound; see the Sound Driver manual, and also the simple SysBeep procedure in the Operating System Utilities manual *** neither manual currently exists ***. (The standard sound procedure calls SysBeep.)

(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 Macintosh User 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 NewDialog. The Dialog

Manager works from the dialog record created by NewDialog, just as when it operates on a dialog window, but it disposes of the window before returning to the application. Normally your application will not access the dialog record for an alert; however, there is a way that this can happen: for any alert, you can specify a procedure that will be executed repeatedly during the alert, and this procedure may access the dialog record. For details, see the alert routines under "Invoking Alerts" in the "Dialog Manager Routines" section.

USING THE DIALOG MANAGER

This section discusses how the Dialog Manager routines fit into the general flow of an application program and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

Before using the Dialog Manager, you should initialize QuickDraw, the Font Manager, the Window Manager, the Menu Manager, and TextEdit, in that order. The first Dialog Manager routine to call is InitDialogs, which initializes the Dialog Manager. If you want the font in your dialog and alert windows to be other than the system font, call SetDAFont to change the font.

Where appropriate in your program, call NewDialog or GetNewDialog to create any dialogs you need. Usually you'll call GetNewDialog, which takes descriptive information about the dialog from a dialog template in a resource file. You can instead pass the information in individual parameters to NewDialog. In either case, you can supply a pointer to the storage for the dialog record or let it be allocated by the Dialog Manager. When you no longer need a dialog, you'll usually call CloseDialog if you supplied the storage, or DisposDialog if not.

In most cases, you probably won't have to make any changes to the dialogs from the way they're defined in the resource file. However, if you should want to modify an item in a dialog, you can call GetDItem to get the information about the item and SetDItem to change it. In particular, SetDItem is the routine to use for installing a userItem. In some cases it may be appropriate to call some other Toolbox routine to change the item; for example, to change or move a control in a dialog, you would get its handle from GetDItem and then call the appropriate Control Manager routine. There are also two procedures 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 procedure 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, and then usually call DialogSelect if so. 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. 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 editText items normally comes up with the insertion point in the first such item in its item list. You may instead want to bring up a dialog box with text selected in an editText item, or to 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 input when a number is required can be given the opportunity to type the entry again. The SelIText procedure makes this possible.

For alerts, if you want other sounds besides the standard ones (up to three short beeps), write your own sound procedure and call ErrorSound to make it the current sound procedure. 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 defined to have something other than a standard icon (or nothing at all) in its top left corner.

If you're going to invoke a dialog or alert when the resource file might not be accessible, first call CouldDialog or CouldAlert, which will make the dialog or alert template and related resources unable to be purged from memory. You can later make them purgeable again by calling FreeDialog or FreeAlert.

Finally, you can substitute text in statText items with text that you specify in the ParamText procedure. This means, for example, that a document name supplied by the user can appear in an error message.

DIALOG MANAGER ROUTINES

This section describes all the Dialog Manager procedures and functions. They're presented in their Pascal form; for information on using them from assembly language, see the manual Programming Macintosh Applications in Assembly Language.

Initialization

```
PROCEDURE InitDialogs (restartProc: ProcPtr);
```

Call InitDialogs once before all other Dialog Manager routines, to initialize the Dialog Manager.

- It sets a pointer to a fail-safe procedure as specified by restartProc; this pointer will be accessed when a system error (such as running out of memory) occurs. RestartProc should point to a procedure that will restart the application after a system error. If no such procedure is desired, pass NIL as the

parameter.

Assembly-language note: The Dialog Manager stores the address of the fail-safe procedure in a global variable named RestProc.

- It installs the standard sound procedure.
- It passes empty strings to ParamText.

PROCEDURE ErrorSound (soundProc: ProcPtr);

ErrorSound sets the sound procedure for dialogs and alerts to the procedure pointed to by soundProc; if you don't call ErrorSound, the Dialog Manager uses the standard sound procedure. (For details, see the "Alerts" section above.) If you pass NIL for soundProc, there will be no sound (or menu bar blinking) at all.

Assembly-language note: The address of the sound procedure being used is stored in the global variable DABeeper.

PROCEDURE SetDAFont (fontNum: INTEGER); [Pascal only]

For subsequently created dialogs and alerts, SetDAFont sets the font of the dialog or alert window's grafPort to the font having the specified font number. If you don't call this procedure, the system font is used. SetDAFont affects statText and editText items but not titles of controls, which are always in the system font.

Assembly-language note: Assembly-language programmers can simply set the global variable DlgFont to the desired font number.

Creating and Disposing of Dialogs

```
FUNCTION NewDialog (dStorage: Ptr; boundsRect: Rect; title: Str255;
            visible: BOOLEAN; procID: INTEGER; behind: WindowPtr;
            goAwayFlag: BOOLEAN; refCon: LongInt; items: Handle) :
            DialogPtr;
```

NewDialog creates a dialog as specified by its parameters and returns a pointer to the new dialog. The first eight parameters (dStorage through refCon) are passed to the Window Manager function NewWindow, which creates the dialog window; the meanings of these parameters are summarized below. The items parameter is a handle to the dialog's item list. You can get the items handle by calling the Resource Manager to read the item list from the resource file into memory.

(note)

Advanced programmers can create their own item lists in memory rather than have them read from a resource file. The exact format is given later under "Formats of Resources for Dialogs and Alerts".

DStorage is analogous to the wStorage parameter of NewWindow; it's a pointer to the storage to use for the dialog record. If you pass NIL for dStorage, the dialog record will be allocated on the heap (which, in the case of modeless dialogs, may cause the heap to become fragmented).

BoundsRect, a rectangle given in global coordinates, determines the dialog window's size and location. It becomes the portRect of the window's grafPort. Remember that the top coordinate of this rectangle should be at least 25 points below the top of the screen for a modal dialog, to allow for the menu bar and the border around the portRect, and at least 40 points below the top of the screen for a modeless dialog, to allow for the menu bar and the window's title bar.

Title is the title of a modeless dialog box; pass the empty string for modal dialogs.

If the visible 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 with a call to the Window Manager procedure ShowWindow.

(note)

NewDialog generates an update event for the entire window contents, so the items aren't drawn immediately, with the exception of controls. The Dialog Manager calls the Control Manager to draw controls, and the Control Manager draws them immediately rather than via the standard update mechanism. Because of this, the Dialog Manager calls the Window Manager procedure ValidRect for the enclosing rectangle of each control, so the controls

won't be drawn twice. If you find that the other items aren't being drawn soon enough after the controls, try making the window invisible initially and then calling ShowWindow to show it.

ProcID is the window definition ID, which leads to the window definition function for this type of window. The window definition IDs for the standard types of dialog window are dBoxProc for the modal type and documentProc for the modeless type.

The behind parameter specifies the window behind which the dialog window is to be placed on the desktop. Pass POINTER(-1) to bring up the dialog window in front of all other windows.

GoAwayFlag applies to modeless dialog boxes; if it's TRUE, the dialog window has a close box in its title bar when the window is active.

RefCon is the dialog window's reference value, which the application may store into and access for any purpose.

NewDialog 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 window class in the window record to dialogKind.

```
FUNCTION GetNewDialog (dialogID: INTEGER; dStorage: Ptr; behind:
            WindowPtr) : DialogPtr;
```

Like NewDialog (above), GetNewDialog creates a dialog as specified by its parameters and returns a pointer to the new dialog. Instead of having the parameters boundsRect, title, visible, procID, goAwayFlag, and refCon, GetNewDialog has a single dialogID parameter, where dialogID is the resource ID of a dialog template that supplies the same information as those parameters. The dialog template also contains the resource ID of the dialog's item list. After calling the Resource Manager to read the item list into memory (if it's not already in memory), GetNewDialog makes a copy of the item list and uses that copy; thus you may have multiple independent dialogs whose items have the same types, locations, and initial contents. The dStorage and behind parameters of GetNewDialog have the same meaning as in NewDialog.

```
PROCEDURE CloseDialog (theDialog: DialogPtr);
```

CloseDialog removes theDialog's window from the screen and deletes it from the window list, just as when the Window Manager procedure CloseWindow is called. It 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 the items in the dialog (except for pictures and icons, which might be shared resources), and any data structures associated

with them. For example, it would dispose of the region occupied by the thumb of a scroll bar, or a similar region for some other control in the dialog.

CloseDialog does **not** dispose of the dialog record or the item list. Figure 6 illustrates the effect of CloseDialog (and DisposDialog, described below).

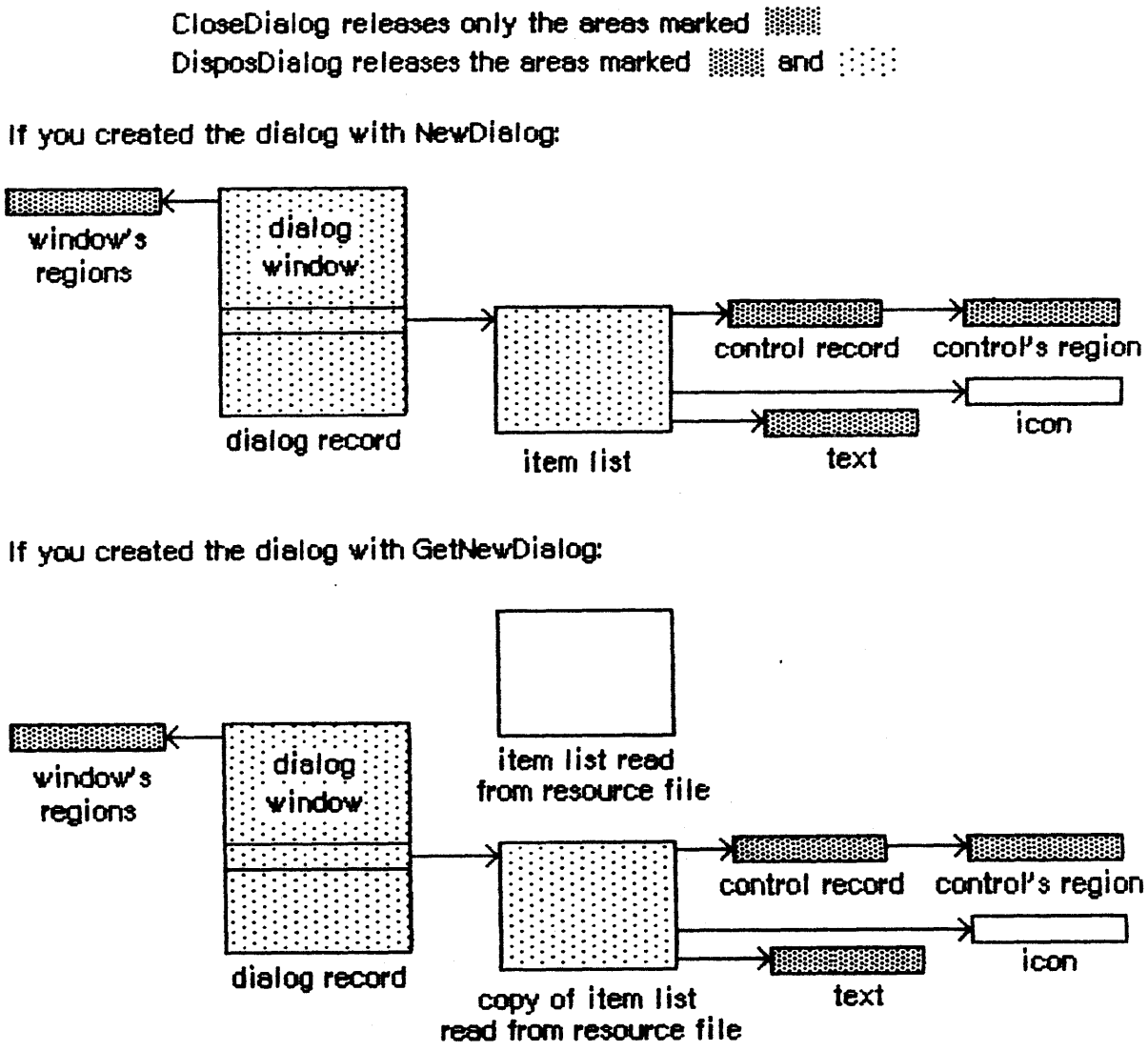


Figure 6. CloseDialog and DisposDialog

Call CloseDialog when you're done with a dialog if you supplied NewDialog or GetNewDialog with a pointer to the dialog storage (in the dStorage parameter) when you created the dialog.

(note)

Even if you didn't supply a pointer to the dialog storage, you may want to call CloseDialog if you created the dialog with NewDialog. You would call CloseDialog if you wanted to keep the item list around (since, unlike GetNewDialog, NewDialog does not use a copy of the item

list).

```
PROCEDURE DisposDialog (theDialog: DialogPtr);
```

DisposDialog calls CloseDialog (above) and then releases the memory occupied by the dialog's item list and dialog record. Call DisposDialog when you're done with a dialog if you let the dialog record be allocated on the heap when you created the dialog (by passing NIL as the dStorage parameter to NewDialog or GetNewDialog).

```
PROCEDURE CouldDialog (dialogID: INTEGER);
```

CouldDialog ensures that the dialog template having the given resource ID is in memory and makes it unable to be purged. It does the same for the dialog window's definition function, the dialog's item list resource, and any items defined as resources. This is useful if the dialog box may come up when the resource file isn't accessible, such as during a disk copy.

```
PROCEDURE FreeDialog (dialogID: INTEGER);
```

Given the resource ID of a dialog template previously specified in a call to CouldDialog (above), FreeDialog undoes the effect of CouldDialog. It should be called when there's no longer a need to keep the resources in memory.

Handling Dialog Events

```
PROCEDURE ModalDialog (filterProc: ProcPtr; VAR itemHit: INTEGER);
```

Call ModalDialog after creating a modal dialog and bringing up its window in the frontmost plane. ModalDialog repeatedly gets and handles events in the dialog's window; after handling an event involving an enabled dialog item, it returns with the item number in itemHit. Normally you'll then do whatever is appropriate as a response to an event in that item.

ModalDialog gets each event by calling the Toolbox Event Manager function GetNextEvent. If the event is a mouse-down event outside the content region of the dialog window, ModalDialog emits sound number 1 (which should be a single beep) and gets the next event; otherwise, it filters and handles the event as described below.

(note)

Once before getting each event, ModalDialog calls SystemTask, a Desk Manager procedure that needs to be called regularly if the application is to support the use of desk accessories.

The filterProc parameter determines how events are filtered. If it's NIL, the standard filterProc function is executed; this causes ModalDialog to return 1 in itemHit if the Return key or Enter key is pressed. If filterProc isn't NIL, ModalDialog filters events by executing the function it points to. Your filterProc function should have three parameters and return a Boolean value. For example, this is how it would be declared if it were named MyFilter:

```
FUNCTION MyFilter (theDialog: DialogPtr; VAR theEvent:
            EventRecord; VAR itemHit: INTEGER) : BOOLEAN;
```

A function result of FALSE tells ModalDialog to go ahead and handle the event, which either can be sent through unchanged or can be changed to simulate a different event. A function result of TRUE tells ModalDialog to return immediately rather than handle the event; in this case, the filterProc function sets itemHit to the item number that ModalDialog should return.

(note)

If you want it to be consistent with the standard filterProc function, your function should at least check whether the Return key or Enter key was pressed and, if so, return 1 in itemHit and a function result of TRUE.

You can use the filterProc function, for example, to treat a typed character in a special way (such as 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-down 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 function can call that procedure and return FALSE without altering the current event.

(note)

ModalDialog calls GetNextEvent with a mask that excludes disk-inserted events. To receive disk-inserted events, your filterProc function can call GetNextEvent (or EventAvail) with a mask that accepts only that type of event.

ModalDialog handles the events for which the filterProc function returns FALSE as follows:

- In response to an activate or update event for the dialog window, ModalDialog activates or updates the window.

- If the mouse button is pressed in an editText item, ModalDialog responds to the mouse activity as appropriate (displaying an insertion point or selecting text). If a key-down event occurs and there's an editText item, text entry and editing are handled in the standard way for such items (except that if the Command key is down, ModalDialog responds as though it isn't). In either case, ModalDialog returns if the editText item is enabled or does nothing if it's disabled. If a key-down event occurs when there's

no editText item, ModalDialog does nothing.

- If the mouse button is pressed in a control, ModalDialog calls the Control Manager function TrackControl. If the mouse button is released inside the control and the control is enabled, ModalDialog returns; otherwise, it does nothing.

- If the mouse button is pressed in any other enabled item in the dialog box, ModalDialog returns. If the mouse button is pressed in any other disabled item or in no item, or if any other event occurs, ModalDialog does nothing.

```
FUNCTION IsDialogEvent (theEvent: EventRecord) : BOOLEAN;
```

If your application includes any modeless dialogs, call IsDialogEvent after calling the Toolbox Event Manager function GetNextEvent. Pass the current event in theEvent. IsDialogEvent determines whether theEvent needs to be handled as part of a dialog. If theEvent 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, just handle the event yourself like any other event that's not dialog-related. When TRUE is returned, you'll generally end up passing the event to DialogSelect for it to handle (as described below), but first you should do some additional checking:

- DialogSelect doesn't handle keyboard equivalents for commands. Check whether the event is a key-down event with the Command key held down and, if so, carry out the command if it's one that applies when a dialog window is active. (If the command doesn't so apply, do nothing.)

- In special cases, you may want to bypass DialogSelect or do some preprocessing before calling it. If so, check for those events and respond accordingly. You would need to do this, for example, if the dialog is to respond to disk-inserted events.

For cases other than these, pass the event to DialogSelect for it to handle.

```
FUNCTION DialogSelect (theEvent: EventRecord; VAR theDialog: DialogPtr;
          VAR itemHit: INTEGER) : BOOLEAN;
```

You'll normally call DialogSelect after IsDialogEvent, passing in theEvent an event that needs to be handled as part of a modeless dialog. DialogSelect handles the event as described below. If the event involves an enabled dialog item, DialogSelect returns a function result of TRUE with the dialog pointer in theDialog and the item number in itemHit; otherwise, it returns FALSE with theDialog and itemHit 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.

If the event is an activate or update event for a dialog window, DialogSelect activates or updates the window and returns FALSE.

If the event is a mouse-down event in an editText item, DialogSelect responds as appropriate (displaying an insertion point or selecting text). If it's a key-down event and there's an editText item, text entry and editing are handled in the standard way. In either case, DialogSelect returns TRUE if the editText item is enabled or FALSE if it's disabled. If a key-down event is passed when there's no editText item, DialogSelect returns FALSE.

(note)

For a key-down event, DialogSelect doesn't check to see whether the Command key is held down; to handle keyboard equivalents of commands, you have to check for them before calling DialogSelect. Similarly, to treat a typed character in a special way (such as ignore it, or make it have 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.

If the event is a mouse-down event in a control, DialogSelect calls the Control Manager function TrackControl. If the mouse button is released inside the control and the control is enabled, DialogSelect returns TRUE; otherwise, it returns FALSE.

If the event is a mouse-down event in any other enabled item, DialogSelect returns TRUE. If it's a mouse-down event in any other disabled item or in no item, or if it's any other event, DialogSelect returns FALSE.

PROCEDURE DlgCut (theDialog: DialogPtr); [Pascal only]

DlgCut checks whether theDialog has any editText items and, if so, applies the TextEdit procedure TECut to the currently selected editText item. (If the dialog record's editField is Ø or greater, DlgCut passes the contents of the textH field to TECut.) You can call DlgCut to handle the editing command Cut when a modeless dialog window is active.

Assembly-language note: Assembly-language programmers can just read the dialog record's fields and call TextEdit directly.

PROCEDURE DlgCopy (theDialog: DialogPtr); [Pascal only]

DlgCopy is the same as DlgCut (above) except that it calls TECopy, for handling the Copy command.

PROCEDURE DlgPaste (theDialog: DialogPtr); [Pascal only]

DlgPaste is the same as DlgCut (above) except that it calls TEPaste, for handling the Paste command.

PROCEDURE DlgDelete (theDialog: DialogPtr); [Pascal only]

DlgDelete is the same as DlgCut (above) except that it calls TEDelete, for handling the Clear command.

PROCEDURE DrawDialog (theDialog: DialogPtr);

DrawDialog draws the contents of the given dialog box. Since DialogSelect and ModalDialog handle 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.

Invoking Alerts

FUNCTION Alert (alertID: INTEGER; filterProc: ProcPtr) : INTEGER;

This function invokes the alert defined by the alert template that has the given resource ID. It 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 creates and displays the alert window for this alert and draws the alert box.

(note)
It creates the alert window by calling NewDialog, and does the rest of its processing by calling ModalDialog.

Alert repeatedly gets and handles events in the alert window until an enabled item is clicked, at which time it returns the item number. Normally you'll then do whatever is appropriate in response to a click of that item.

Alert gets each event by calling the Toolbox Event Manager function 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 below.

The filterProc parameter has the same meaning as in ModalDialog (see above). If it's NIL, the standard filterProc function is executed, which makes the Return key or the Enter key have the same effect as clicking the default button. If you specify your own filterProc function and want to retain this feature, you must include it in your function. You can find out what the current default button is by looking at the aDefItem field of the dialog record for the alert (via the dialog pointer passed to the function).

Alert handles the events for which the filterProc function returns FALSE as follows:

- If the mouse button is pressed in a control, Alert calls the Control Manager procedure 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. (It disposes of the alert window and its associated data structures, the item list, and the items.)

(note)

The Alert function's removal of the alert box would not be the desired result if the user clicked a check box or radio button; however, normally alerts contain only static text, icons, pictures, and buttons that are supposed to make the alert box go away. If your alert contains other items besides these, consider whether it might be more appropriate as a dialog.

```
FUNCTION StopAlert (alertID: INTEGER; filterProc: ProcPtr) : INTEGER;
```

StopAlert is the same as the Alert function (above) except that before drawing the items of the alert in the alert box, it draws the Stop icon in the top left corner of the box (within the rectangle (1Ø,2Ø,42,52)). The Stop icon has the following resource ID:

```
CONST stopIcon = Ø;
```

If the application's resource file doesn't include an icon with that ID number, the Dialog Manager uses the standard Stop icon in the system resource file (see Figure 7).

Figure 7. Standard Alert Icons

```
FUNCTION NoteAlert (alertID: INTEGER; filterProc: ProcPtr) : INTEGER;
```

NoteAlert is like StopAlert except that it draws the Note icon, which has the following resource ID:

```
CONST noteIcon = 1;
```

```
FUNCTION CautionAlert (alertID: INTEGER; filterProc: ProcPtr) :
        INTEGER;
```

CautionAlert is like StopAlert except that it draws the Caution icon, which has the following resource ID:

```
CONST ctnIcon = 2;
```

```
PROCEDURE CouldAlert (alertID: INTEGER);
```

CouldAlert ensures that the alert template having the given resource ID is in memory and makes it unable to be purged. It does the same for the alert window's definition function, the alert's item list resource, and any items defined as resources. This is useful if the alert may occur when the resource file isn't accessible, such as during a disk copy.

```
PROCEDURE FreeAlert (alertID: INTEGER);
```

Given the resource ID of an alert template previously specified in a call to CouldAlert (above), FreeAlert undoes the effect of CouldAlert. It should be called when there's no longer a need to keep the resources in memory.

Manipulating Items in Dialogs and Alerts

```
PROCEDURE ParamText (param∅,param1,param2,param3: Str255);
```

ParamText provides a means of substituting text in statText items: param∅ through param3 will replace the special strings '^∅' through '^3' in all statText items in all subsequent dialog or alert boxes. Pass empty strings for parameters not used.

Assembly-language note: Assembly-language programmers may pass NIL for parameters not used or for strings that are not to be changed.

For example, if the text is defined as 'Cannot open document ^∅' and docName is a string variable containing a document name that the user typed, you can call ParamText(docName,'','','').

(warning)
All strings that will need to be translated to foreign languages should be stored in resource files.

Assembly-language note: The Dialog Manager stores handles to the four ParamText parameters in a global array named DAStrings.

```
PROCEDURE GetDItem (theDialog: DialogPtr; itemNo: INTEGER; VAR type:
          INTEGER; VAR item: Handle; VAR box: Rect);
```

GetDItem returns in its VAR parameters the following information about the item numbered itemNo in the given dialog's item list: in the type parameter, the item type; in the item parameter, a handle to the item (or, for item type userItem, the procedure pointer); and in the box parameter, the display rectangle for the item.

Suppose, for example, that you want to change the title of a control in a dialog box. You can get the item handle with GetDItem, convert it to type ControlHandle, and call the Control Manager procedure SetCTitle to change the title. Similarly, to move the control or change its size, you would call MoveControl or SizeControl.

(note)
To access the text of a statText or editText item, pass the handle returned by GetDItem to GetIText or SetIText (see below).

```
PROCEDURE SetDItem (theDialog: DialogPtr; itemNo: INTEGER; type:
          INTEGER; item: Handle; box: Rect);
```

SetDItem sets the item numbered itemNo in the given dialog's item list, as specified by the parameters (without drawing the item). The type parameter is the item type; the item parameter is a handle to the item (or, for item type userItem, the procedure pointer); and the box parameter is the display rectangle for the item.

Consider, for example, how to install an item of type userItem in a dialog: In the item list in the resource file, define an item in which the type is set to userItem and the display rectangle to (Ø,Ø,Ø,Ø). Specify that the dialog window be invisible (in either the dialog template or the NewDialog call). After creating the dialog, convert the item's procedure pointer to type Handle; then call SetDItem, passing that handle and the display rectangle for the item. Finally, call the Window Manager procedure ShowWindow to display the dialog window.

(note)

Do not use SetDItem to change the text of a statText or editText item or to change or move a control. See the description of GetDItem above for more information.

```
PROCEDURE GetIText (item: Handle; VAR text: Str255);
```

Given a handle to a statText or editText item in a dialog box, as returned by GetDItem, GetIText returns the text of the item in the text parameter.

```
PROCEDURE SetIText (item: Handle; text: Str255);
```

Given a handle to a statText or editText item in a dialog box, as returned by GetDItem, SetIText sets the text of the item to the specified text and draws the item. For example, suppose the exact content of a dialog's text item cannot be determined until the application is running, but the display rectangle is defined in the resource file: Call GetDItem to get a handle to the item, and call SetIText with the desired text.

```
PROCEDURE SelIText (theDialog: DialogPtr; itemNo: INTEGER;
          strtSel,endSel: INTEGER);
```

Given a pointer to a dialog and the item number of an editText item in the dialog box, SelIText does the following:

- If the item contains text, SelIText sets the selection range to extend from character position strtSel up to but not including character position endSel. The selection range is inverted unless strtSel 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, SelIText simply displays the insertion point.

For example, if the user makes an unacceptable entry in the editText item, the application can put up 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 Ø for strtSel and a very large number for endSel. For details about selection range and character position, see the TextEdit manual.

FUNCTION GetAlrtStage : INTEGER; [Pascal only]

GetAlrtStage returns the stage of the last occurrence of an alert, as a number from Ø to 3.

Assembly-language note: Assembly-language programmers can get this number by accessing the global variable ACount. In addition, the global variable ANumber contains the resource ID of the alert template of the last alert that occurred.

PROCEDURE ResetAlrtStage; [Pascal only]

ResetAlrtStage 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 ParamText to change the text of an alert such that from the user's point of view it's a different alert.

Assembly-language note: Assembly-language programmers can set the global variable ACount to -1 for the same effect.

MODIFYING TEMPLATES IN MEMORY

When you call GetNewDialog or one of the routines that invokes an alert, the Dialog Manager calls the Resource Manager to read the dialog or alert template from the resource file and return a handle to it. If the template is already in memory, the Resource Manager just returns a

handle to it. If you want, you can call the Resource Manager yourself to read the template into memory (and make it unpurgeable), and then make changes to it before calling the dialog or alert routine. When called by the Dialog Manager, the Resource Manager will return a handle to the template as you modified it.

To modify a template in memory, you need to know its exact structure and the data type of the handle through which it may be accessed. These are discussed below for dialogs and alerts.

Dialog Templates in Memory

The data structure of a dialog template is as follows:

```
TYPE DialogTemplate = RECORD
                        boundsRect: Rect;     {becomes window's portRect}
                        procID:     INTEGER;  {window definition ID}
                        visible:    BOOLEAN;  {TRUE if visible}
                        filler1:    BOOLEAN;  {not used}
                        goAwayFlag: BOOLEAN;  {TRUE if has go-away region}
                        filler2:    BOOLEAN;  {not used}
                        refCon:     LongInt;  {window's reference value}
                        itemsID:    INTEGER;  {resource ID of item list}
                        title:      Str255    {window's title}
                      END;
```

The filler1 and filler2 fields are there only to ensure that the goAwayFlag and refCon fields begin on a word boundary. The itemsID field contains the resource ID of the dialog's item list. The other fields are the same as the parameters of the same name in the NewDialog function; they provide information about the dialog window.

You access the dialog template by converting the handle returned by the Resource Manager to a template handle:

```
TYPE DialogTHndl = ^DialogTPtr;
     DialogTPtr  = ^DialogTemplate;
```

Alert Templates in Memory

The data structure of an alert template is as follows:

```
TYPE AlertTemplate = RECORD
                       boundsRect: Rect;      {becomes window's portRect}
                       itemsID:    INTEGER;   {resource ID of item list}
                       stages:     StageList  {alert stage information}
                     END;
```

BoundsRect is the rectangle that becomes the portRect of the window's grafPort. The itemsID field contains the resource ID of the item list for the alert.

The information in the stages field determines exactly what should happen at each stage of the alert. It's packed into a word that has the following structure:

```
TYPE StageList = PACKED ARRAY [1..4] OF
                   RECORD
                     boldItem:  Ø..1; {default button item number minus 1}
                     boxDrawn:  BOOLEAN; {TRUE if alert box to be drawn}
                     sound:     Ø..3     {sound number}
                   END;
```

The elements of the StageList array are stored in reverse order of the stages: element 1 is for the fourth stage, and element 4 is for the first stage.

BoldItem indicates which button should be the default button (and therefore boldly outlined in the alert box). If the first two items in the alert's item list are the OK button and the Cancel button, respectively, Ø will refer to the OK button and 1 to the Cancel button. The reason for this is that the value of boldItem plus 1 is interpreted as an item number, and normally items 1 and 2 are the OK and Cancel buttons, respectively. Whatever the item having the corresponding item number happens to be, a bold rounded-corner rectangle will be drawn around its display rectangle.

(warning)
When deciding where to place items in an alert box, be sure to allow room for any bold outlines that may be drawn.

BoxDrawn is TRUE if the alert box is to be drawn.

The sound field specifies which sound should be emitted at this stage of the alert, with a number from Ø to 3 that's passed to the current sound procedure. You can call ErrorSound to specify your own sound procedure; if you don't, the standard sound procedure will be used (as described earlier in the "Alerts" section).

You access the alert template by converting the handle returned by the Resource Manager to a template handle:

```
TYPE AlertTHndl = ^AlertTPtr;
     AlertTPtr  = ^AlertTemplate;
```

Assembly-language note: Rather than offsets into the fields of the StageList data structure, there are masks for accessing the information stored for an alert stage in a stages word; they're listed in the summary at the end of this manual.

FORMATS OF RESOURCES FOR DIALOGS AND ALERTS

Every dialog template, alert template, and item list must be stored in a resource file, as must any icons or QuickDraw pictures in item lists and any control templates for items of type ctrlItem+resCtrl. The exact formats of a dialog template, alert template, and item list in a resource file are given below. For icons and pictures, the resource type is 'ICON' or 'PICT' and the resource data is simply the icon or the picture. The format of a control template is discussed in the Control Manager manual.

Dialog Templates in a Resource File

The resource type for a dialog template is 'DLOG', and the resource data has the same format as a dialog template in memory.

Number of bytes	Contents
8 bytes	Same as boundsRect parameter to NewDialog
2 bytes	Same as procID parameter to NewDialog
1 byte	Same as visible parameter to NewDialog
1 byte	Ignored
1 byte	Same as goAwayFlag parameter to NewDialog
1 byte	Ignored
4 bytes	Same as refCon parameter to NewDialog
2 bytes	Resource ID of item list
n bytes	Same as title parameter to NewDialog (1-byte length in bytes, followed by the characters of the title)

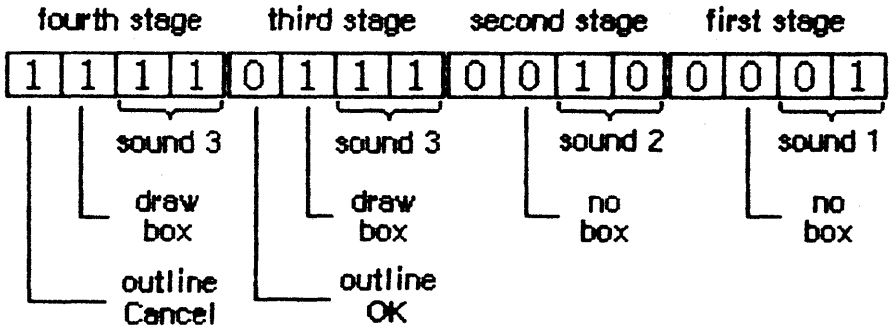
Alert Templates in a Resource File

The resource type for an alert template is 'ALRT', and the resource data has the same format as an alert template in memory.

Number of bytes	Contents
8 bytes	Rectangle enclosing alert window
2 bytes	Resource ID of item list
2 bytes	Stages

The resource data ends with a word of information about stages. As shown in the example in Figure 8, there are four bits of stage information for each of the four stages, from the four low-order bits for the first stage to the four high-order bits for the fourth stage. Each set of four bits is as follows:

Number of bits	Contents
1 bit	Item number minus 1 of default button; normally Ø is OK and 1 is Cancel
1 bit	1 if alert box is to be drawn, Ø if not
2 bits	Sound number (Ø through 3)



(value: hexadecimal F721)

Figure 8. Sample Stages Word

(note)

So that the disk won't be accessed just for an alert that beeps, you may want to set the resPreload attribute of the alert's template in the resource file. For more information, see the Resource Manager manual.

Item Lists in a Resource File

The resource type for an item list is 'DITL'. The resource data begins with a word containing the number of items in the list minus 1. This is what follows for each item:

Number of bytes	Contents	
4 bytes	Ø (placeholder for handle or procedure pointer)	
8 bytes	Display rectangle (local coordinates)	
1 byte	Item type	
1 byte	Length of following data in bytes	
n bytes (n is even)	If item type is:	Content is:
	ctrlItem+resCtrl	Resource ID (length 2)
	any other ctrlItem	Title of the control
	statText, editText	The text
	iconItem, picItem	Resource ID (length 2)
	userItem	Empty (length Ø)

As shown here, the first four bytes serve as a placeholder for the item's handle or, for item type userItem, its procedure pointer; the handle or pointer is stored after the item list is read into memory. The next eight bytes define the display rectangle for the item, and the next byte gives the length of the data that follows: for a text item, it's the text itself; for an icon, picture, or control of type ctrlItem+resCtrl, it's the two-byte resource ID for the item; and for any other type of control, it's the title of the control. For userItems, no data follows the item type. When the data is text or a control title, the number of bytes it occupies must be even to ensure word alignment of the next item.

Assembly-language note: Offsets into the fields of an item list are available as global constants; they're listed in the summary.

SUMMARY OF THE DIALOG MANAGER

Constants

```
CONST  { Item types }

       ctrlItem    = 4;    {add to following four constants}
       btnCtrl     = Ø;    {standard button control}
       chkCtrl     = 1;    {standard check box control}
       radCtrl     = 2;    {standard "radio button" control}
       resCtrl     = 3;    {control defined in control template}
       statText    = 8;    {static text}
       editText    = 16;   {editable text (dialog only)}
       iconItem    = 32;   {icon}
       picItem     = 64;   {QuickDraw picture}
       userItem    = Ø;    {application-defined item (dialog only)}
       itemDisable = 128;  {add to any of above to disable}

       { Item numbers of OK and Cancel buttons }

       OK     = 1;
       Cancel = 2;

       { Resource IDs of alert icons }

       stopIcon = Ø;
       noteIcon = 1;
       ctnIcon  = 2;
```

Data Types

```
TYPE DialogPtr    = WindowPtr;
     DialogPeek   = ^DialogRecord;

     DialogRecord = RECORD
                      window:    WindowRecord; {dialog window}
                      items:     Handle;       {item list}
                      textH:     TEHandle; {current editText item}
                      editField: INTEGER;  {editText item number minus 1}
                      editOpen:  INTEGER;  {used internally}
                      aDefItem:  INTEGER   {default button item number}
                    END;

     DialogTHndl   = ^DialogTPtr;
     DialogTPtr    = ^DialogTemplate;
```

```
DialogTemplate = RECORD
                   boundsRect: Rect;    {becomes window's portRect}
                   procID:     INTEGER; {window definition ID}
                   visible:    BOOLEAN; {TRUE if visible}
                   filler1:    BOOLEAN; {not used}
                   goAwayFlag: BOOLEAN; {TRUE if has go-away region}
                   filler2:    BOOLEAN; {not used}
                   refCon:     LongInt; {window's reference value}
                   itemsID:    INTEGER; {resource ID of item list}
                   title:      Str255   {window's title}
                 END;

AlertTHndl    = ^AlertTPtr;
AlertTPtr     = ^AlertTemplate;

AlertTemplate = RECORD
                  boundsRect: Rect;     {becomes window's portRect}
                  itemsID:    INTEGER;  {resource ID of item list}
                  stages:     StageList {alert stage information}
                END:

StageList = PACKED ARRAY [1..4] OF
              RECORD
                boldItem:  Ø..1; {default button item number minus 1}
                boxDrawn:  BOOLEAN; {TRUE if alert box to be drawn}
                sound:     Ø..3     {sound number}
              END;
```

Routines

Initialization

```
PROCEDURE InitDialogs (restartProc: ProcPtr);
PROCEDURE ErrorSound  (soundProc: ProcPtr);
PROCEDURE SetDAFont   (fontNum: INTEGER);  [Pascal only]
```

Creating and Disposing of Dialogs

```
FUNCTION  NewDialog    (dStorage: Ptr; boundsRect: Rect; title: Str255;
                        visible: BOOLEAN; procID: INTEGER; behind:
                        WindowPtr; goAwayFlag: BOOLEAN; refCon: LongInt;
                        items: Handle) : DialogPtr;
FUNCTION  GetNewDialog (dialogID: INTEGER; dStorage: Ptr; behind:
                        WindowPtr) : DialogPtr;
PROCEDURE CloseDialog  (theDialog: DialogPtr);
PROCEDURE DisposDialog (theDialog: DialogPtr);
PROCEDURE CouldDialog  (dialogID: INTEGER);
PROCEDURE FreeDialog   (dialogID: INTEGER);
```

Handling Dialog Events

```
PROCEDURE ModalDialog   (filterProc: ProcPtr; VAR itemHit: INTEGER);
FUNCTION  IsDialogEvent (theEvent: EventRecord) : BOOLEAN;
FUNCTION  DialogSelect  (theEvent: EventRecord; VAR theDialog: DialogPtr;
                         VAR itemHit: INTEGER) : BOOLEAN;
PROCEDURE DlgCut        (theDialog: DialogPtr);  [Pascal only]
PROCEDURE DlgCopy       (theDialog: DialogPtr);  [Pascal only]
PROCEDURE DlgPaste      (theDialog: DialogPtr);  [Pascal only]
PROCEDURE DlgDelete     (theDialog: DialogPtr);  [Pascal only]
PROCEDURE DrawDialog    (theDialog: DialogPtr);
```

Invoking Alerts

```
FUNCTION  Alert        (alertID: INTEGER; filterProc: ProcPtr) : INTEGER;
FUNCTION  StopAlert    (alertID: INTEGER; filterProc: ProcPtr) : INTEGER;
FUNCTION  NoteAlert    (alertID: INTEGER; filterProc: ProcPtr) : INTEGER;
FUNCTION  CautionAlert (alertID: INTEGER; filterProc: ProcPtr) : INTEGER;
PROCEDURE CouldAlert   (alertID: INTEGER);
PROCEDURE FreeAlert    (alertID: INTEGER);
```

Manipulating Items in Dialogs and Alerts

```
PROCEDURE ParamText      (paramØ,param1,param2,param3: Str255);
PROCEDURE GetDItem       (theDialog: DialogPtr; itemNo: INTEGER; VAR type:
                          INTEGER; VAR item: Handle; VAR box: Rect);
PROCEDURE SetDItem       (theDialog: DialogPtr; itemNo: INTEGER; type:
                          INTEGER; item: Handle; box: Rect);
PROCEDURE GetIText       (item: Handle; VAR text: Str255);
PROCEDURE SetIText       (item: Handle; text: Str255);
PROCEDURE SelIText       (theDialog: DialogPtr; itemNo: INTEGER; strtSel,
                          endSel: INTEGER);
FUNCTION  GetAlrtStage :  INTEGER;  [Pascal only]
PROCEDURE ResetAlrtStage;  [Pascal only]
```

UserItem Procedure

```
PROCEDURE MyItem (theWindow: WindowPtr; itemNo: INTEGER);
```

Sound Procedure

```
PROCEDURE MySound (soundNo: INTEGER);
```

FilterProc Function for Modal Dialogs and Alerts

```
FUNCTION MyFilter (theDialog: DialogPtr; VAR theEvent: EventRecord;
                   VAR itemHit: INTEGER) : BOOLEAN;
```

Assembly-Language Information

Constants

```
; Item types

ctrlItem     .EQU   4     ;add to following four constants
btnCtrl      .EQU   Ø     ;standard button control
chkCtrl      .EQU   1     ;standard check box control
radCtrl      .EQU   2     ;standard "radio button" control
resCtrl      .EQU   3     ;control defined in control template
statText     .EQU   8     ;static text
editText     .EQU   16    ;editable text (dialog only)
iconItem     .EQU   32    ;icon
picItem      .EQU   64    ;QuickDraw picture
userItem     .EQU   Ø     ;application-defined item (dialog only)
itemDisabl   .EQU   128   ;add to any of above to disable}

; Item numbers of OK and Cancel buttons

okButton       .EQU   1
cancelButton   .EQU   2

; Resource IDs of alert icons

stopIcon       .EQU   Ø
noteIcon       .EQU   1
ctnIcon        .EQU   2

; Masks for stages word in alert template

volBits        .EQU    3     ;sound number
alBit          .EQU    4     ;whether to draw box
okDismissal    .EQU    8     ;item number of default button minus 1
```

Dialog Record Data Structure

```
dWindow        Dialog window
items          Handle to dialog's item list
teHandle       Handle to current editText item
editField      Item number of editText item minus 1
editOpen       Used internally
aDefItem       Item number of default button
dWindLen       Length of dialog record
```

Dialog Template Data Structure

dBounds	Rectangle that becomes portRect of dialog window's grafPort
dWindProc	Window definition ID
dVisible	Flag for whether dialog window is visible
dGoAway	Flag for whether dialog window has a go-away region
dRefCon	Dialog window's reference value
dItems	Resource ID of dialog's item list
dTitle	Dialog window's title

Alert Template Data Structure

aBounds	Rectangle that becomes portRect of alert window's grafPort
aItems	Resource ID of alert's item list
aStages	Stages word; information for alert stages

Item List Data Structure

dlgMaxIndex	Number of items minus 1
itmHndl	Handle or procedure pointer for this item
itmRect	Display rectangle for this item
itmType	Item type for this item
itmData	Length byte followed by that many bytes of data for this item (must be even length)

Variables

Name	Size	Contents
RestProc	4 bytes	Address of restart fail-safe procedure
DAStrings	16 bytes	Handles to ParamText strings
DABeeper	4 bytes	Address of current sound procedure
DlgFont	2 bytes	Font number for dialogs and alerts
ACount	2 bytes	Stage number of last alert (Ø through 3)
ANumber	2 bytes	Resource ID of last alert

GLOSSARY

alert: A warning or report of an error, in the form of an alert box, sound from the Macintosh's speaker, or both.

alert box: A box that appears on the screen to give a warning or report an error during a Macintosh application.

alert template: A resource that contains information from which the Dialog Manager can create an alert.

alert window: The window in which an alert box is displayed.

default button: In an alert box or modal dialog, the button whose effect will occur if the user presses Return or Enter. In an alert box, it's boldly outlined; in a modal dialog, it's boldly outlined or the OK button.

dialog: Same as dialog box.

dialog box: A box that a Macintosh application displays to request information it needs to complete a command, or to report that it's waiting for a process to complete.

dialog record: The internal representation of a dialog, where the Dialog Manager stores all the information it needs for its operations on that dialog.

dialog template: A resource that contains information from which the Dialog Manager can create a dialog.

dialog window: The window in which a dialog box is displayed.

disabled: A disabled item in a dialog or alert box has no effect when clicked.

display rectangle: A rectangle that determines where an item is displayed within a dialog or alert box.

icon: A 32-by-32 bit image that graphically represents an object, concept, or message.

item: In dialog and alert boxes, a control, icon, picture, or piece of text, each displayed inside its own display rectangle.

item list: A list of information about all the items in a dialog or alert box.

item number: The index, starting from 1, of an item in an item list.

modal dialog: A dialog that requires the user to respond before doing any other work on the desktop.

modeless dialog: A dialog that allows the user to work elsewhere on the desktop before responding.

sound procedure: A procedure that will emit one of up to four sounds from the Macintosh's speaker. Its integer parameter ranges from Ø to 3 and specifies which sound.

stage: Every alert has four stages, corresponding to consecutive occurrences of the alert, and a different response may be specified for each stage.

COMMENTS?

Macintosh User Education encourages your comments on this manual.

- What do you like or dislike about it?

- Were you able to find the information you needed?

- Was it complete and accurate?

- Do you have any suggestions for improvement?

Please send your comments to the author (indicated on the cover page) at 10460 Bandley Drive M/S 3-G, Cupertino CA 95014. Mark up a copy of the manual or note your remarks separately. (We'll return your marked-up copy if you like.)

Thanks for your help!

MACINTOSH USER EDUCATION

The Desk Manager: A Programmer's Guide /DSKMGR/DESK

See Also: Macintosh User Interface Guidelines
Macintosh Operating System Reference Manual
QuickDraw: A Programmer's Guide
The Window Manager: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Dialog Manager: A Programmer's Guide
The Menu Manager: A Programmer's Guide

Modification History:			
	First Draft (ROM 2.Ø)	C. Rose	2/3/83
	Erratum Added	C. Rose	2/28/83
	Second Draft (ROM 4)	C. Rose	6/14/83
	Third Draft (ROM 7)	C. Rose	9/26/83

This manual introduces you to the Desk Manager, the part of the Macintosh User Interface Toolbox that handles desk accessories such as the Calculator. It describes the simple programmatic interface to the Desk Manager and tells you how to define your own desk accessories.

Summary of significant changes and additions since last version:

- OpenDeskAcc is now a Desk Manager routine, as is the new procedure CloseDeskAcc (page 7).
- A new function, SystemEdit, processes standard editing commands in desk accessories (page 8). Four new messages are passed to a desk accessory's control routine to handle this (page 13).
- Storing the window pointer in the Device Control Entry is now optional for a desk accessory's open routine, and setting the windowKind field to the driver's reference number is required (page 13).
- A desk accessory may be displayed in a window created by the Dialog Manager; if so, its control routine must respond to the "cursor" message in a special way (page 14). Applications allowing access to desk accessories must initialize TextEdit and the Dialog Manager.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Desk Manager, the part of the Macintosh User Interface Toolbox that supports the use of desk accessories from an application; the Calculator, for example, is a standard desk accessory available to any application. *** Eventually this will become part of a large manual describing the entire Toolbox. *** You'll learn how to use the Desk Manager routines and how to define your own accessories.

(hand)

> This manual describes version 7 of the ROM. If you're using a different version, the Desk Manager may not work as discussed here.

Like all documentation about Toolbox units, this manual assumes you're familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's Memory Manager. You should also be familiar with the following:

- The Toolbox Event Manager, the Window Manager, the Menu Manager, and the Dialog Manager.
- The basic concepts behind the Resource Manager.
- I/O drivers, as discussed in the Macintosh Operating System Reference Manual.

This manual begins with an introduction to the Desk Manager and desk accessories. Next, a section on using the Desk Manager introduces you to its routines and tells how they fit into the flow of your application. This is followed by the detailed descriptions of all Desk Manager procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions is a section for programmers who want to define their own desk accessories.

Finally, there's a summary of the Desk Manager routine calls, for quick reference, and a glossary of terms used in this manual. *** The glossary will eventually be merged with the glossaries from the other Toolbox documentation. The many Operating System terms have not been included in the glossary in this manual. ***

ABOUT THE DESK MANAGER

The Desk Manager enables your application to support desk accessories, which are "mini-applications" that can be run at the same time as a Macintosh application. The standard Calculator desk accessory is shown in Figure 1. *** The method of highlighting an active desk accessory is currently different from what's shown here and will probably change. ***

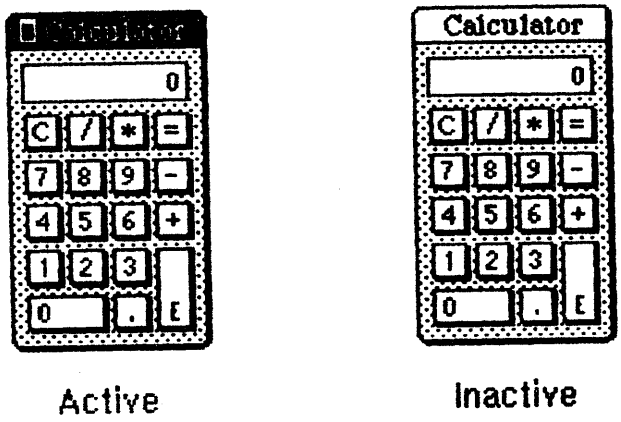


Figure 1. The Calculator Desk Accessory

The Macintosh user opens desk accessories by choosing them from the standard Apple menu (the menu whose title is an Apple symbol), which by convention is the first menu in the menu bar. When a desk accessory is chosen from this menu, it's usually displayed in a window on the desktop, and that window becomes the active window. (See Figure 2.)

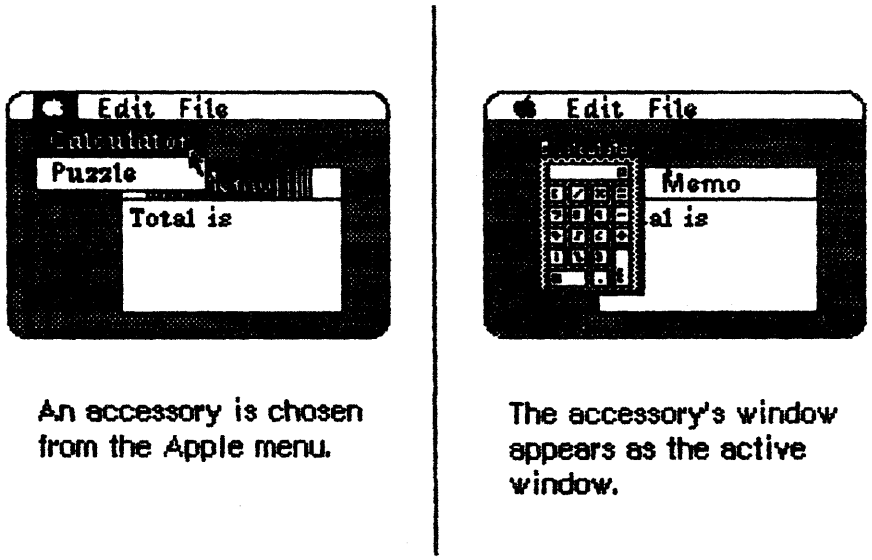


Figure 2. Opening a Desk Accessory

After being selected, the accessory may be used as long as it's active. The user can activate other windows and then reactivate the desk accessory by clicking inside it. Whenever a standard desk accessory is active, it has a close box in its title bar. Clicking the close box makes the accessory disappear, and the window that's then the frontmost becomes active.

The window associated with a desk accessory usually resembles a rounded-corner document window, as shown above. It also may look and behave like a dialog window; the accessory can call on the Dialog Manager to create the window and then use Dialog Manager routines to

operate on it. In either case, the window will be a system window, as indicated by its window class.

Many applications will have an Edit menu that includes the standard commands Cut, Copy, Paste, and Undo, which may be useful in desk accessories as well as in the application's windows. The Desk Manager provides a mechanism that lets those commands be applied to a desk accessory when it's active. Even if the commands aren't particularly useful for editing within the accessory, they may be useful for cutting and pasting between the accessory and the application or even another accessory. For example, the result of a calculation made with the Calculator desk accessory can be copied into a document prepared in MacWrite *** eventually ***.

A desk accessory may also have its own menu. When the accessory becomes active, the title of its menu is added to the menu bar and menu items may be chosen from it. Any of the application's menus or menu items that no longer apply are disabled. A desk accessory can even have an entire menu bar full of its own menus, which will completely replace the menus already in the menu bar. When an accessory that has its own menu or menus becomes inactive, the menu bar is restored to normal.

Although desk accessories are usually displayed in windows (one per accessory), this is not necessarily so. It's possible for an accessory to have only a menu (or menus) and not a window. The menu includes a command to close the accessory. Also, a desk accessory that's displayed in a window may create any number of additional windows while it's open.

You can define your own desk accessories. A desk accessory is actually a special type of I/O driver--special in that it may have its own windows and menus for interacting with the user. Desk accessories and other I/O drivers used by Macintosh applications are stored in resource files.

USING THE DESK MANAGER

This section introduces you to the Desk Manager routines and how they fit into the general flow of an application program. The routines themselves are described in detail in the next section.

To allow access to desk accessories, your application must do the following:

- Initialize TextEdit and the Dialog Manager, in case any desk accessories are displayed in windows created by the Dialog Manager (which uses TextEdit).

- Set up the Apple menu as the first menu in the menu bar. You can put the names of all currently available desk accessories in a

menu by using the Menu Manager routine AddResMenu (see the Menu Manager manual for details).

When the user chooses a menu item from the Apple menu, you should call the Menu Manager procedure GetItem to get the name of the corresponding desk accessory, and then the Desk Manager function OpenDeskAcc to open and display the accessory. You can close the desk accessory with the CloseDeskAcc procedure.

When the Toolbox Event Manager function GetNextEvent reports that a mouse down event has occurred, the application calls the Window Manager function FindWindow to find out where the mouse button was pressed. If FindWindow returns the predefined constant inSysWindow, which means that the mouse button was pressed in a system window, you should call the Desk Manager procedure SystemClick. SystemClick handles mouse down events in system windows, routing them to desk accessories where appropriate.

(hand)
The application need not be concerned with exactly which desk accessories are currently open, except when it wants to use the accessory directly itself (such as the Mini-Finder accessory).

When the active window changes from an application window to a system window, the application should disable any of its menus or menu items that don't apply while an accessory is active. It should enable them again when one of its own windows becomes active.

When a mouse down event occurs in the menu bar, or a key down event occurs when the Command key is held down, and the application determines that one of the four standard editing commands Cut, Copy, Paste, and Undo has been invoked, it should call SystemEdit. Only if SystemEdit returns FALSE should the application process the editing command itself; if the active window belongs to a desk accessory, SystemEdit passes the editing command on to that accessory and returns TRUE.

Certain periodic actions may be defined for desk accessories. To see that they're performed, you need to call the SystemTask procedure at least once every time through your main event loop.

The two remaining Desk Manager routines--SystemEvent and SystemMenu--are never called by the application, but are described in this manual because they reveal inner mechanisms of the Toolbox that may be of interest to advanced Macintosh programmers.

DESK MANAGER ROUTINES

This section describes all the Desk Manager procedures and functions. They're presented in their Pascal form; for information on using them from assembly language, see "Using the Toolbox from Assembly Language"

*** doesn't exist, but see "Using QuickDraw from Assembly Language" in the QuickDraw manual ***.

Opening and Closing Desk Accessories

```
FUNCTION OpenDeskAcc (theAcc: Str255) : INTEGER;
```

OpenDeskAcc opens the desk accessory having the given name, displays its window (if any) as the active window, and returns its reference number (or 0 if the accessory can't be opened). The name is the accessory's resource name, which you get from the Apple menu by calling the Menu Manager procedure GetItem. OpenDeskAcc calls the Resource Manager to read the desk accessory from the resource file.

```
PROCEDURE CloseDeskAcc (refNum: INTEGER);
```

CloseDeskAcc closes the desk accessory having the given reference number. Usually, though, the application won't close the desk accessory; instead, it will be closed when the user clicks its close box (or, if there's a menu instead of a window, when the user chooses the command to close the accessory). Also, since the application heap is deallocated when the application terminates, every desk accessory goes away at that time.

Handling Events in Desk Accessories

```
PROCEDURE SystemClick (theEvent: EventRecord; theWindow: WindowPtr);
```

When a mouse down event occurs and the Window Manager routine FindWindow reports that the mouse button was pressed in a system window, the application should call SystemClick with the event record and the window pointer. If the given window belongs to a desk accessory, SystemClick sees that the event gets handled properly.

SystemClick determines which part of the desk accessory's window the mouse button was pressed in, and responds accordingly (similar to the way your application responds to mouse activities in its own windows).

- If the mouse button was pressed in the content region of the window and the window was active, SystemClick sends the mouse down event to the desk accessory, which processes it as appropriate.

- If the mouse button was pressed in the content region and the window was inactive, SystemClick makes it the active window.

- If the mouse button was pressed in the drag region, SystemClick calls the Window Manager routine DragWindow to pull an outline of

the window across the screen and move the window to a new location. If the window was inactive, DragWindow also makes it the active window (unless the Command key was pressed along with the mouse button).

- If the mouse button was pressed in the go-away region, SystemClick calls the Window Manager routine TrackGoAway to determine whether the mouse is still inside the go-away region when the click is completed: if so, it tells the desk accessory to close itself; otherwise, it does nothing.

```
FUNCTION SystemEdit (editCmd: INTEGER) : BOOLEAN;
```

Call SystemEdit when the user invokes the editing command specified by editCmd, which may be one of the following predefined constants:

```
CONST cutCmd   = Ø;   {Cut command}
      copyCmd  = 1;   {Copy command}
      pasteCmd = 2;   {Paste command}
      undoCmd  = 3;   {Undo command}
```

If the active window doesn't belong to a desk accessory, SystemEdit returns FALSE; the application should then process the editing command as usual. If the active window does belong to a desk accessory, SystemEdit asks that accessory to process the command and returns TRUE; in this case, the application should ignore the command.

(hand)
It's up to the application to make sure desk accessories get their editing commands. In particular, make sure your application doesn't disable the Edit menu or any of the four commands when a desk accessory is activated.

Performing Periodic Actions

```
PROCEDURE SystemTask;
```

For each open desk accessory, SystemTask causes the accessory to perform the periodic action defined for it, if any such action has been defined and if the proper time period has passed since the action was last performed. For example, a clock accessory can be defined such that the second hand is to move once every second; the periodic action for the accessory will be to move the second hand to the next position, and SystemTask will alert the accessory every second to perform that action.

You should call SystemTask as often as possible, usually once every time through your main event loop. Call it more than once if your application does an unusually large amount of processing each time through the loop.

(hand)
Preferably SystemTask would be called at least every 60th of a second.

Advanced Routines

FUNCTION SystemEvent (theEvent: EventRecord) : BOOLEAN;

SystemEvent is called only by the Toolbox Event Manager routine GetNextEvent when it receives an event, to determine whether the event should be handled by the application or by the system. If the given event should be handled by the application, SystemEvent returns FALSE; otherwise, it calls the appropriate system code to handle the event and returns TRUE.

In the case of a null, abort, or mouse down event, SystemEvent does nothing but return FALSE. Notice that it responds this way to a mouse down event even though the event may in fact have occurred in a system window (and therefore may have to be handled by the system). The reason for this is that the check for exactly where the event occurred (via the Window Manager routine FindWindow) is made later by the application and so would be made twice if SystemEvent were also to do it. To avoid this duplication, SystemEvent passes the event on to the application and lets it make the sole call to FindWindow. Should FindWindow reveal that the mouse down event did occur in a system window, the application can then call SystemClick, as described above, to get the system to handle it.

If the given event is a mouse up, key down, key up, or auto-key event, SystemEvent checks whether the active window belongs to a desk accessory and whether that accessory can handle this type of event. If so, it sends the event to the desk accessory and returns TRUE; otherwise, it returns FALSE.

If SystemEvent is passed an activate or update event, it checks whether the window it occurred in is a system window belonging to a desk accessory and whether that accessory can handle this type of event. If so, it sends the event to the desk accessory and returns TRUE; otherwise, it returns FALSE.

(hand)
It's unlikely that a desk accessory would not be set up to handle activate and update events.

Finally, if the given event is a disk inserted event, SystemEvent does some low-level processing (by calling the Operating System routine MountVolume) but passes the event on to the application by returning FALSE, in case the application wants to do further processing.

```
PROCEDURE SystemMenu (menuResult: LongInt);
```

SystemMenu is called only by the Menu Manager routines MenuSelect and MenuKey, when an item in a menu belonging to a desk accessory has been chosen. The menuResult parameter has the same format as the value returned by MenuSelect and MenuKey: the menu ID in the high-order word and the menu item number in the low-order word. (The menu ID will be negative.) SystemMenu directs the desk accessory to perform the appropriate action for the given menu item.

DEFINING YOUR OWN DESK ACCESSORIES

To define your own desk accessories, you must create the corresponding I/O driver and include it in a resource file. Standard or shared desk accessories are stored in the system resource file. Accessories specific to an application are rare; if there are any, they're stored in the application's resource file.

The resource type for I/O drivers is 'DRVR'. The resource ID for a desk accessory is the driver's unit number and should be between 12 and 31 inclusive. The resource name should be whatever you want to appear in the Apple menu, but should also include a nonprinting character; by convention, the name should begin with a NUL character (ASCII code Ø). The nonprinting character is needed to avoid conflict with file names that are the same as the names of desk accessories.

The structure of an I/O driver is described in the Macintosh Operating System Reference Manual. The rest of this section reviews some of that information and presents additional details pertaining specifically to I/O drivers that are desk accessories.

(hand)

> Usually drivers are created entirely from assembly language, but you can use an assembly language-to-Pascal interface that will enable you to write the body of the driver routines in Pascal. An interface named ProtoOrn has been created for this purpose at Apple; for more information, see your Macintosh software coordinator.

As illustrated in Figure 3, the I/O driver begins with a few words of flags and other data for the driver, followed by offsets to the routines that do the work of the driver, an optional title, and finally the routines themselves.

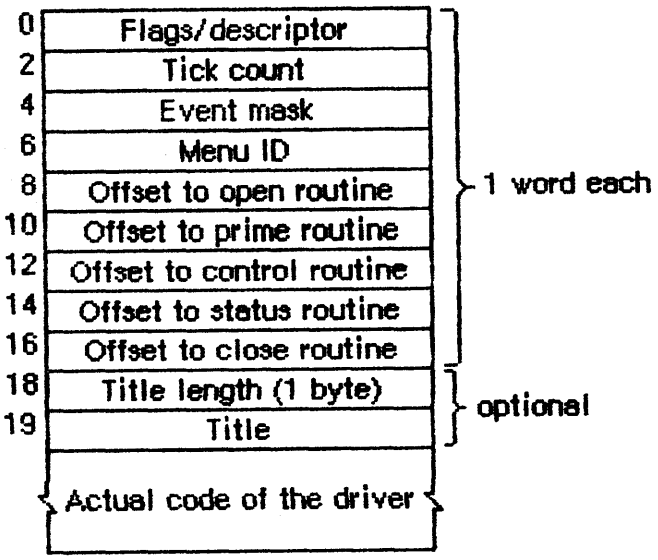


Figure 3. Desk Accessory I/O Driver

The first four words of the driver for a desk accessory contain the following:

1. A flags/descriptor word. Bits Ø through 7 and bit 12 are relevant only to ROM-based drivers; they're ignored for desk accessories. Bits 8 through 11 are the enable flags for the driver routines. The following flags are especially for desk accessories:

Flag	Name	Meaning if set
bit 13	dNeedTime	Driver needs time for performing a periodic action for the desk accessory
bit 14	dNeedLock	Driver will be locked in memory as soon as it's opened

   If you want to test one of these flags with the assembly-language instruction BTST, remember that when the destination of BTST is a memory location, the operation is performed on a **byte** read from that location.

2. If the dNeedTime flag is set, a tick count indicating how often the periodic action should occur. A tick count of Ø means it should happen as often as possible, 1 means it should happen every 6Øth of a second, 2 means every 3Øth of a second, and so on. The action itself is performed by the control routine in the driver when it's called by the SystemTask procedure.

3. An event mask specifying which events the desk accessory can handle. This should especially include update and activate events and usually will include mouse down events.

4. If the desk accessory has its own menu (or menus), the ID of the menu (or of any of the menus); otherwise, Ø. The menu ID will be negative. For menus defined in resource files, it's the resource

ID; for menus created by the desk accessory, it's any negative number (between -1 and -32767) that you choose to identify this accessory's menu. It must be different from the menu ID stored here for other desk accessories.

Following these four words are the offsets to the driver routines and, optionally, a title for the desk accessory (preceded by its length in bytes). You can use the title in the driver as the title of the accessory's window, or just as a way of identifying the driver in memory.

The Device Control Entry

When any of the routines in the I/O driver is called, a pointer to the driver's Device Control Entry is passed in A1. Most of the data in the Device Control Entry is stored and accessed only by the Operating System, but in some cases the driver routines themselves must store into it. The structure of the Device Control Entry, which is discussed in detail in the Operating System manual, is illustrated in Figure 4. Notice that some of the data is taken from the first four words of the I/O driver.

0	Pointer to start of driver	long
4	Flags (from driver, plus some dynamic flags)	word
6	Driver input queue header: flags	word
8	Driver input queue header: QHead	long
12	Driver input queue header: QTail	long
16	Position pointer (position in file)	long
20	Handle to driver's private storage (optional)	long
24	Reference number for this driver	word
26	Counter for SystemTask timing	long
30	Pointer to driver's window (optional)	long
34	Tick count (from driver)	word
36	Event mask (from driver)	word
38	Menu ID (from driver)	word

Figure 4. Device Control Entry

The Driver Routines

Of the five possible driver routines, only three need to exist for desk accessories: the open, close, and control routines. The other routines (prime and status) may be used if desired for a particular accessory.

The open routine opens the desk accessory.

- It creates the window to be displayed when the accessory is opened, if any, specifying that it be invisible (since OpenDeskAcc will display it). The window can be created with the Dialog Manager routine NewDialog (or GetNewDialog) if desired; the accessory will look and respond like a dialog box, and subsequent operations may be performed on it with Dialog Manager routines. In any case, the open routine sets the windowKind field in the window record to the reference number for the driver, which it gets from the Device Control Entry. (The reference number will be negative.) It also may store the window pointer in the Device Control Entry if desired.

- If the driver has any private storage, it allocates the storage, stores a handle to it in the Device Control Entry, and initializes any local variables. It might, for example, create a menu or menus for the accessory.

The close routine closes the desk accessory, disposing of its window (if any) and replacing the window pointer in the Device Control Entry with NIL (if one was stored there by the open routine). If the driver has any private storage, the close routine also disposes of that storage.

The action taken by the control routine depends on information passed in the parameter block pointed to by A0. A message is passed in the "op code" field (a word located at 26(A0)); this message is simply a number that tells the routine what action to take. There are eight such messages:

Message	Name	Action to be taken by control routine
64	accEvent	Handle a given event
65	accRun	Take the periodic action, if any, for this desk accessory
66	accCursor	Change the cursor shape if appropriate; generate a null event if the window was created by the Dialog Manager
67	accMenu	Handle a given menu item
68	accCut	Handle the Cut command
69	accCopy	Handle the Copy command
70	accPaste	Handle the Paste command
71	accUndo	Handle the Undo command

Along with the accEvent message, the control routine receives as a parameter a pointer to an event record (a long integer located at 28(A0)). It responds by handling the given event in whatever way is appropriate for this desk accessory. SystemClick and SystemEvent call the control routine with this message to send the driver an event that it should handle--for example, an activate event that makes the desk accessory active or inactive. When a desk accessory becomes active, its control routine might install a menu in the menu bar. If the accessory becoming active has more than one menu, the control routine should respond as follows:

- Store the accessory's unique menu ID in the system global mBarEnable. (This is the negative menu ID in the I/O driver and the Device Control Entry.)

- Call the Menu Manager routines GetMenuBar to save the current menu list and ClearMenuBar to clear the menu bar.

- Install the accessory's own menus in the menu bar.

Then, when the desk accessory becomes inactive, the control routine should call SetMenuBar to restore the former menu list, call DrawMenuBar to draw the menu bar, and set mBarEnable to Ø.

The accRun message tells the control routine to perform the periodic action for this desk accessory. For every open driver that has the dNeedTime flag set, the SystemTask procedure calls the control routine with this message if the proper time period has passed since the action was last performed.

The accCursor message makes it possible for the cursor to have a special shape when it's inside an active desk accessory. The control routine is called repeatedly with this message as long as the desk accessory is active. If desired, the control routine may respond by checking whether the mouse position is in the desk accessory's window and then changing the shape of the cursor if so. Furthermore, if the desk accessory is displayed in window created by the Dialog Manager, the control routine should respond to the accCursor message by generating a null event (storing the event code for a null event in an event record) and passing it to DialogSelect. This enables the Dialog Manager to blink the vertical bar in editText items.

(hand)

In assembly language, the code might look like this:

```
CLR.L     -SP          ; event code for null event is Ø
PEA       2(SP)        ; pass null event
CLR.L     -SP          ; pass NIL dialog pointer
CLR.L     -SP          ; pass NIL pointer
_DialogSelect          ; invoke DialogSelect
ADDQ.L    #4,SP        ; pop off result and null event
```

When the accMenu message is sent to the control routine, the following information is passed in the parameter block: the menu ID of the desk accessory's menu in a word at 28(AØ), and a menu item number in a word at 3Ø(AØ). The control routine takes the appropriate action for when the given menu item is chosen from the menu, and then makes the Menu Manager call HiliteMenu(Ø) to remove the highlighting from the menu bar.

Finally, the control routine should respond to one of the last four messages--accCut through accUndo--by processing the corresponding editing command in the desk accessory window if appropriate. SystemEdit calls the control routine with these messages. For information on cutting and pasting between a desk accessory and the

application, or between two desk accessories, see the *** forthcoming *** Scrap Manager manual.

(hand)

If you use .INCLUDE to include a file named SysEqu.Text when you assemble your program, the messages sent to the driver's control routine will be available in symbolic form, as will offsets into the fields of the I/O driver and Device Control Entry.

A Sample Desk Accessory

*** to be supplied; meanwhile, see your Macintosh software coordinator ***

SUMMARY OF THE DESK MANAGER

```
CONST cutCmd   = Ø;   {Cut command}
      copyCmd  = 1;   {Copy command}
      pasteCmd = 2;   {Paste command}
      undoCmd  = 3;   {Undo command}
```

Opening and Closing Desk Accessories

```
FUNCTION  OpenDeskAcc  (theAcc: Str255) : INTEGER;
PROCEDURE CloseDeskAcc (refNum: INTEGER);
```

Handling Events in Desk Accessories

```
PROCEDURE SystemClick (theEvent: EventRecord; theWindow: WindowPtr);
FUNCTION  SystemEdit  (editCmd: INTEGER) : BOOLEAN;
```

Performing Periodic Actions

```
PROCEDURE SystemTask;
```

Advanced Routines

```
FUNCTION  SystemEvent (theEvent: EventRecord) : BOOLEAN;
PROCEDURE SystemMenu  (menuResult: LongInt);
```

GLOSSARY

desk accessory: A "mini-application", implemented as an I/O driver, that can be run at the same time as a Macintosh application.

tick: A 60th of a second.

COMMENTS?

Macintosh User Education encourages your comments on this manual.

- What do you like or dislike about it?

- Were you able to find the information you needed?

- Was it complete and accurate?

- Do you have any suggestions for improvement?

Please send your comments to the author (indicated on the cover page) at 10460 Bandley Drive M/S 3-G, Cupertino CA 95014. Mark up a copy of the manual or note your remarks separately. (We'll return your marked-up copy if you like.)

Thanks for your help!

MACINTOSH USER EDUCATION

The Scrap Manager: A Programmer's Guide /SMGR/SCRAP

See Also: Macintosh User Interface Guidelines
Macintosh Operating System Reference Manual
QuickDraw: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Segment Loader: A Programmer's Guide
The Desk Manager: A Programmer's Guide
Putting Together a Macintosh Application

Modification History:			
	First Draft (ROM 7)	C. Rose	1Ø/21/83
	Erratum Added	C. Rose	11/16/83

ABSTRACT

The Scrap Manager is a set of simple routines and data types that help Macintosh applications manipulate the Clipboard for cutting and pasting between applications, desk accessories, or an application and a desk accessory. This manual describes the Scrap Manager in detail.

Erratum:

The 'TEXT' type of data in the desk scrap is simply a series of ASCII characters, without a character count or an optional comment. If you want to know the count, you can get it by passing a NIL handle to the GetScrap function.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Scrap Manager, a new part of the Macintosh User Interface Toolbox in ROM version 7. *** Eventually it will become part of a comprehensive manual describing the entire Toolbox. *** The Scrap Manager supports cutting and pasting between applications, desk accessories, or an application and a desk accessory.

Like all documentation about Toolbox units, this manual assumes you're familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's Memory Manager. You should also be familiar with the following:

- QuickDraw pictures
- Resources, as discussed in the Resource Manager manual
- The Toolbox Event Manager

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest only to assembly-language programmers is isolated and labeled so that Pascal programmers can conveniently skip it.

The manual begins with an introduction to the Scrap Manager, an overview of the scrap that you manipulate with it, and a discussion of the types of data that the scrap may contain.

Next, a section on using the Scrap Manager introduces its routines and tells how they fit into the flow of your application. This is followed by detailed descriptions of all Scrap Manager routines, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions is a section that gives the exact format of the scrap, for those programmers who are interested; you don't have to read this section to be able to use the Scrap Manager routines.

Finally, there's a summary of the Scrap Manager, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE SCRAP MANAGER

The Scrap Manager is a set of simple routines and data types that help Macintosh applications manipulate the desk scrap, which is where data that's cut (or copied) and pasted between applications is stored. An application can also use the desk scrap for storing data that's cut and pasted within the application, but usually it will have its own private scrap for this purpose. The format of the private scrap may be whatever the application likes, since no other application will use it.

From the user's point of view, there's a single place where all cut or copied data resides, and it's called the Clipboard. The Cut command deletes data from a document and places it in the Clipboard; the Copy command copies data into the Clipboard without deleting it from the document. The next Paste command--whether applied to the same document or another, in the same application or another--inserts the contents of the Clipboard at a specified place. An application that offers these editing commands will usually also have a special window for displaying the current Clipboard contents; it may show the Clipboard window at all times or only when requested (via the Show Clipboard and Hide Clipboard commands).

The desk scrap is the vehicle for transferring data not only between two applications but also between an application and a desk accessory, or even between two desk accessories. Desk accessories that display text will commonly allow the text to be cut or copied. The user might, for example, use the Calculator accessory to do a calculation and then copy the result into a document. It's also possible for a desk accessory to allow something to be pasted into it.

(hand)

The Scrap Manager is optimized for transferring **small** amounts of data; attempts to transfer very large amounts of data may fail due to lack of memory.

The nature of the data to be transferred varies according to the application. For example, for the Calculator or a word processor the data is text, and for a graphics application it's a picture. The amount of information retained about the data that's transferred also varies. Between two text applications, text can be cut and pasted without any loss of information; however, if the user of a graphics application cuts a picture consisting of text and then pastes it into a document created with a word processor, the text in the picture may not be editable in the word processor, or it may be editable but not look exactly the same as in the graphics application. The Scrap Manager allows for a variety of data types and provides a mechanism whereby applications have control over how much information is retained when data is transferred.

Like any scrap, the desk scrap can be kept on the disk (in the scrap file) if there's not enough room for it in memory. It may remain on the disk throughout the use of the application but must be read back into memory when the application terminates, since the user may then remove that disk and insert another. The Scrap Manager provides routines for writing the desk scrap to the disk and for reading it back into memory.

OVERVIEW OF THE DESK SCRAP

The desk scrap is initially located in the application heap, with a handle to it in low memory. When starting up an application, the Segment Loader temporarily moves the scrap out of the heap into the

stack, reinitializes the heap, and puts the scrap back in the heap. (See Figure 1.) For a short time while it does this, two copies of the scrap exist in the memory allocated for the stack and the heap; for this reason, the desk scrap cannot be bigger than half that amount of memory.

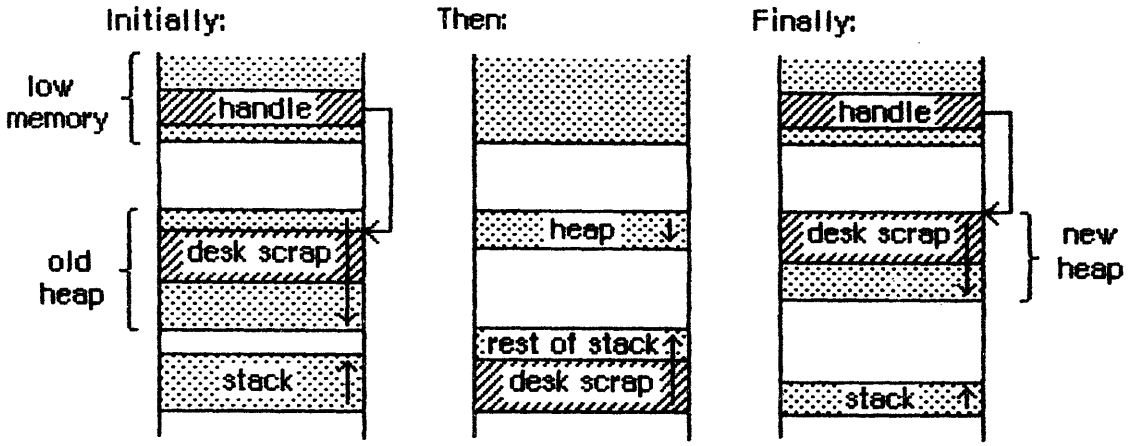


Figure 1. The Desk Scrap at Application Start-up

The application can get the size of the desk scrap by calling a Scrap Manager function named InfoScrap. An application concerned about whether there's room for the desk scrap in memory might be set up so that a small initial segment of the application is loaded in just to check out the scrap size. After a decision is made about whether to keep the scrap in memory or on the disk, the remaining segments can be loaded in as needed.

There are certain disadvantages to keeping the desk scrap on the disk. The disk may be write-protected, may not have enough room for the scrap, or may be removed during use of the application. If the application can't write the scrap to the disk, it should put up an alert box informing the user, who may want to abort the application at that point.

The application must use the desk scrap for any Paste command given before the first Cut or Copy command (that is, the first since the application started up or since a desk accessory was deactivated); this requires copying the desk scrap to the private scrap, if any. Clearly the application must keep the contents of the desk scrap intact until the first Cut or Copy command is given. Thereafter it can ignore the desk scrap until a desk accessory is activated or the application is terminated; in either of these cases, it must copy its private scrap to the desk scrap. Thus whatever was last cut or copied within the application will be pasted if a Paste command is then given in a desk accessory or in the next application.

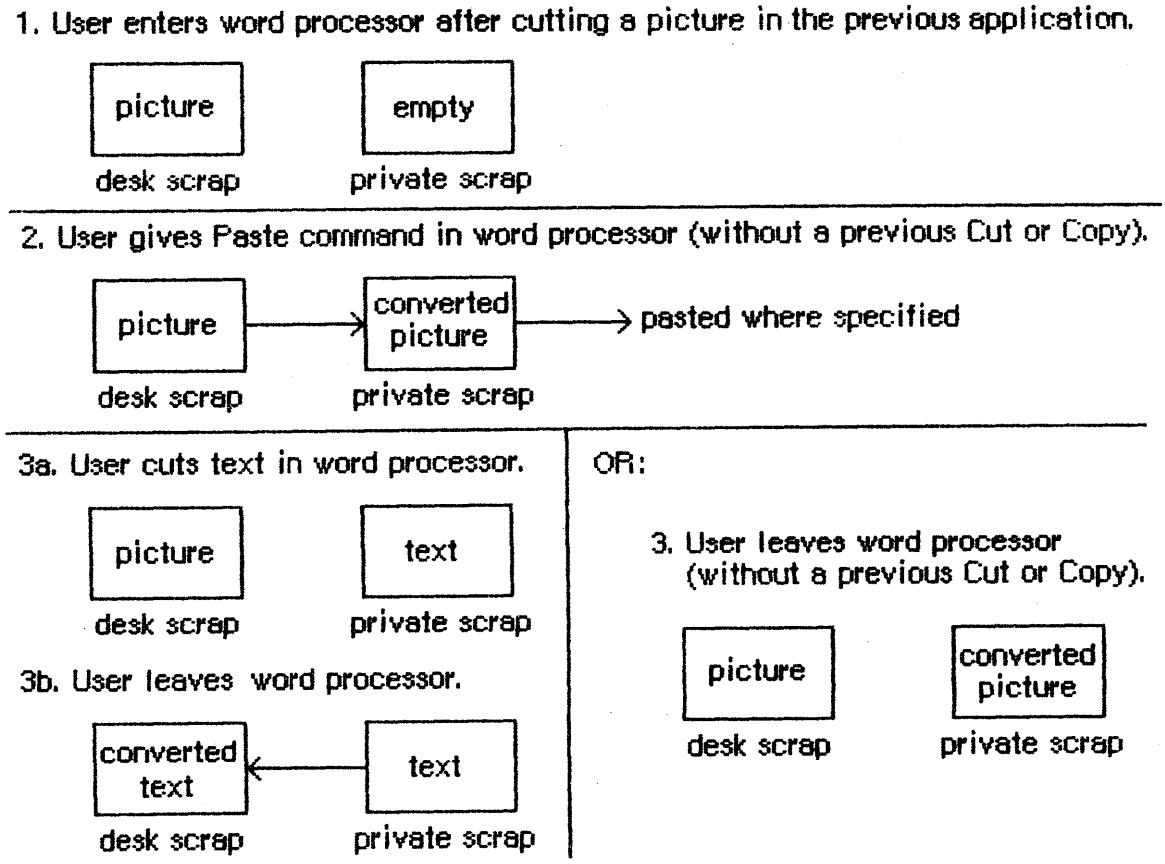


Figure 2. Interaction between Scraps

Figure 2 illustrates how the interaction between the desk scrap and the application's private scrap might occur when the user gives a Paste command in a word processor after cutting a picture in a graphics application. As the picture that was cut gets copied to the private scrap, it's converted to the format of that scrap. If the user leaves the word processor after cutting or copying text, the text first goes into the private scrap and then gets copied to the desk scrap. On the other hand, if the user never gives a Cut or Copy command, the application won't copy the private scrap to the desk scrap, so the original contents of the desk scrap will be retained.

Suppose the word processor in Figure 2 displays the contents of the Clipboard. Normally it will display its private scrap; however, to show the Clipboard contents at any time before step 2, it will have to display the desk scrap instead, or first copy the desk scrap to its private scrap. It can instead simply copy the desk scrap to its private scrap at start-up (step 1), so that showing the Clipboard contents will always mean displaying the private scrap.

A similar scheme to that shown in Figure 2 must be followed when the user reenters an application after using a desk accesory, since the user may have done cutting or copying in the accessory. The application can in fact check whether any such cutting or copying was done, by looking at a count that's returned by InfoScrap. If this count changes during use of the desk accessory, it means the contents

of the desk scrap have changed; the application will have to copy the desk scrap to the private scrap, if any, and update the contents of the Clipboard window, if there is one and if it's visible. If the count returned by InfoScrap hasn't changed, however, the application won't have to take either of these actions.

If the application encounters problems in trying to copy one scrap to another, it should alert the user. The desk scrap may be too large to copy to the private scrap, in which case the user may want to leave the application or just proceed with an empty Clipboard. If the private scrap is too large to copy to the desk scrap, either because it's disk-based and too large to copy into memory or because it exceeds the maximum size allowed for the desk scrap, the user may want to stay in the application and cut or copy something smaller.

DESK SCRAP DATA TYPES

From the user's point of view there can be only one thing in the Clipboard at a time, but internally there may be more than one data item in the desk scrap, each representing the same Clipboard contents in a different form. For example, text cut with a word processor may be stored in the desk scrap both as text and as a QuickDraw picture.

Desk scrap data types are like resource types. As defined in the Resource Manager, their Pascal type is as follows:

```
TYPE ResType = PACKED ARRAY [1..4] OF CHAR;
```

The Scrap Manager recognizes two standard types of data in the desk scrap.

- 'TEXT': a series of ASCII characters, preceded by a long word containing the number of characters and optionally followed by a comment, as described below.

- 'PICT': a QuickDraw picture, which is a saved sequence of drawing commands that can be played back with the DrawPicture command and may include picture comments. (See the QuickDraw manual for details.)

Applications must write at least one of these standard types of data to the desk scrap and must be able to read **both** types. Most applications will prefer one of these types over the other; for example, a word processor prefers text while a graphics application prefers pictures. An application should at least write its preferred standard type of data to the desk scrap, and ideally will write both types (to pass the most information possible on to the receiving application, which may prefer the other type).

An application reading the desk scrap will look for its preferred data type. If its preferred type isn't there, or if it's there but was written by an application having a different preferred type, some

information may be lost in the transfer process. For example, consider the user of a graphics application who cuts a picture consisting of text and then goes into a word processor and pastes it (as illustrated in Figure 3).

- If the graphics application writes only its preferred data type, picture, to the desk scrap (like application A in Figure 3), the text in the picture will not be editable in the word processor, because it will be seen as just a series of drawing commands and not a sequence of characters.

- On the other hand, if the graphics application takes the trouble of recognizing which characters have been drawn in the picture, and also writes them out to the desk scrap as text (like application B in Figure 3), the word processor will be able to treat them like any text, with editing or whatever. In this case, however, any part of the picture that isn't text will be lost.

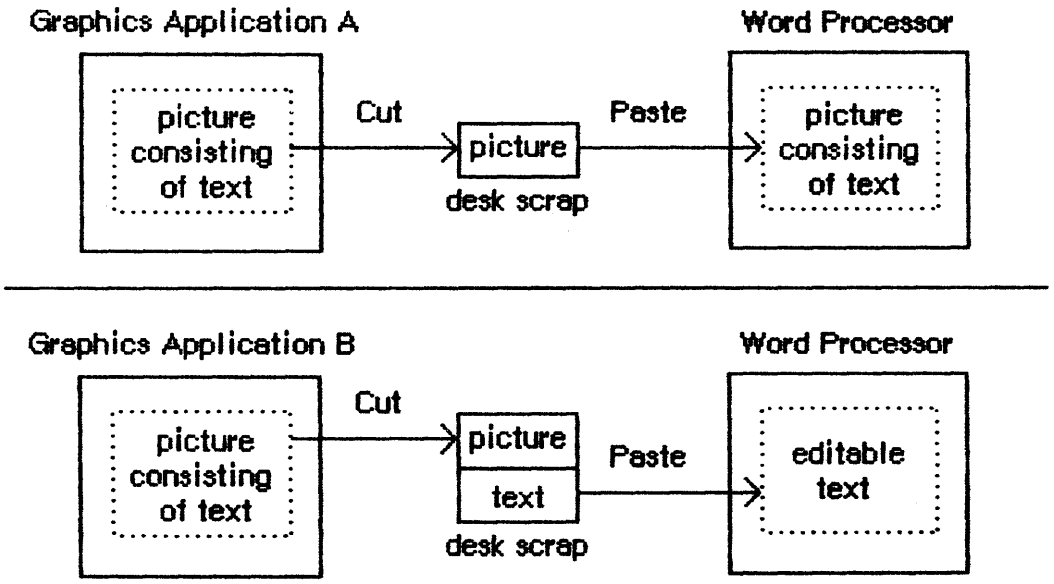


Figure 3. Inter-Application Cutting and Pasting

In addition to the two standard data types, the desk scrap may also contain application-specific types of data. If several applications are to support the transfer of a private type of data, each one will write and read that type--clearly its preferred type--but still must write at least one of the standard types and be able to read both standard types.

(eye)
There should never be more than one of each type of data in the desk scrap at a time.

The order in which data is written to the desk scrap is important: the application should write out the different types in order of preference. For example, if it's a word processor that writes out a

private type of data as well as text and pictures, it should do so in that order.

Since the size of the desk scrap is limited, it may be too costly to write out both an application-specific type of data and one (or both) of the standard types. If so, the comments that can accompany text or pictures might be useful. Instead of creating an application-specific data type, you may be able to encode additional information in these comments. For example, instead of having a data type that consists of text and formatting information combined in an application-specific way, you can encode the formatting information in the text comment. Applications that are to process that information can do so, while others can ignore it.

A text comment follows the last character in the text and must begin with the application ID, a four-character sequence that you choose to uniquely identify your application when you build it. *** (This ID will be discussed further in a future revision of the manual "Putting Together a Macintosh Application".) *** Any data that you like can follow the application ID.

As described in the QuickDraw manual, picture comments may be stored in the definition of a picture with the QuickDraw procedure PicComment. The DrawPicture procedure passes any such comments to a special routine set up by the application for that purpose.

USING THE SCRAP MANAGER

This section discusses how the Scrap Manager routines fit into the general flow of an application program and gives you an idea of which ones you'll need to use. The routines themselves are described in detail in the next section.

The application should inquire as early as possible about the size of the desk scrap to determine whether there will be enough room for itself and the scrap to coexist in the heap; it can do so by calling the InfoScrap function. If there won't be enough room for the desk scrap in the heap, the application should call the UnloadScrap procedure to write the scrap from memory onto the disk. InfoScrap also provides a handle to the desk scrap if it's in memory, its file name on the disk, and a count that's useful for testing whether the contents of the desk scrap have changed during the use of a desk accessory.

If a Paste command is given before the first Cut or Copy command after the application starts up, the application must copy the contents of the desk scrap to its private scrap, if any. It can do this either upon starting up or when the Paste command that needs to use the desk scrap is given. The latter method usually suffices, but applications that support display of the Clipboard will benefit from copying the desk scrap at start-up. The Scrap Manager routine that gets data from the desk scrap is called GetScrap.

When the user gives a command that terminates the application, the application's private scrap will usually have to be copied to the desk scrap. If the desk scrap is on the disk, it must first be read into memory with the LoadScrap function. The application must call ZeroScrap to reinitialize the desk scrap and clear its previous contents, and then PutScrap to put data in the scrap.

(eye)

Do not copy the private scrap to the desk scrap unless a Cut or Copy command was given that changed the contents of the Clipboard.

The same kind of scrap interaction that occurs at application start-up also applies to returning to the application from a desk accessory (that is, an activate event that activates an application window after deactivating a system window). Similarly, the interaction when an application terminates also applies to accessing a desk accessory from the application (as reported by an activate event that deactivates an application window and activates a system window). Note, however, that a desk accessory shouldn't concern itself with writing or reading the desk scrap from the disk.

Cutting and pasting between two desk accessories follows an analogous scenario. As described in the Desk Manager manual, the way a desk accessory learns it must respond to an editing command is that its control routine receives a message telling it to perform the command; the application needs to call the Desk Manager function SystemEdit to make this happen.

SCRAP MANAGER ROUTINES

This section describes all the Scrap Manager routines. They are presented in their Pascal form; for information on using them from assembly language, see "Using the Toolbox from Assembly Language" *** for now, see "Using QuickDraw from Assembly Language" in the QuickDraw manual ***.

Getting Scrap Information

```
FUNCTION InfoScrap : PScrapStuff;
```

InfoScrap returns a pointer to information about the desk scrap. The PScrapStuff data type is defined as follows:

```
TYPE PScrapStuff = ^ScrapStuff;
     ScrapStuff  = RECORD
                     scrapSize:   LongInt;
                     scrapHandle: Handle;
                     scrapCount:  INTEGER;
                     scrapState:  INTEGER;
                     scrapName:   StringPtr
                   END;
```

ScrapSize is the size of the entire desk scrap in bytes. ScrapHandle is a handle to the scrap if it's in memory, or NIL if not. ScrapCount is a count that changes every time ZeroScrap is called and is useful for testing whether the contents of the desk scrap have changed during the use of a desk accessory (see ZeroScrap under "Writing to the Scrap", below). ScrapState is positive if the desk scrap is in memory or Ø if it's on the disk. ScrapName is a pointer to the name of the scrap file, usually DeskScrap.

Keeping the Scrap on the Disk

```
FUNCTION UnloadScrap : LongInt;
```

UnloadScrap writes the desk scrap from memory to the scrap file. If the desk scrap is already on the disk, it does nothing. If no error occurs, UnloadScrap returns Ø; otherwise, it returns an appropriate Operating System error code.

Assembly-language note: The macro you invoke to call UnloadScrap from assembly language is named _UnlodeScrap.

```
FUNCTION LoadScrap : LongInt;
```

LoadScrap reads the desk scrap from the scrap file into memory. If the desk scrap is already in memory, it does nothing. If no error occurs, LoadScrap returns Ø; otherwise, it returns an appropriate Operating System error code.

Assembly-language note: The macro you invoke to call LoadScrap from assembly language is named _LodeScrap.

Reading from the Scrap

```
FUNCTION GetScrap (hDest: Handle; theType: ResType; VAR offset:
          LongInt) : LongInt;
```

GetScrap reads the data of type theType from the desk scrap (whether in memory or on the disk), makes a copy of it in memory, and sets up the hDest handle to point to the copy. Usually you'll pass an empty handle in hDest. In the offset parameter, GetScrap returns the location of the data as an offset (in bytes) from the beginning of the desk scrap. If no error occurs, the function result is the length of the data in bytes; otherwise, it's either an appropriate Operating System error code (which will be negative) or the following predefined constant:

```
CONST noTypeErr = -1Ø2;  {there's no data of the requested type}
```

For example, given an empty handle declared as

```
VAR pHndl: PicHandle
```

you can make the following calls:

```
GetScrap(POINTER(ORD(pHndl)),'PICT');
DrawPicture(pHndl);
```

Your application should pass its preferred data type to GetScrap. If it doesn't prefer one data type over any other, it should try getting different types until the offset returned is Ø. An offset of Ø means that data was the first to be written out and so should be the preferred type of the application that wrote it.

If you pass NIL in hDest, GetScrap will not read in the data. This is useful if you want to be sure the data is there before allocating space for its handle, or if you just want to know the size of the data. If there isn't enough room in memory for a copy of the data, as may be the case for a complicated picture, you can customize QuickDraw's picture retrieval so that DrawPicture will read from the picture directly from the scrap file. (QuickDraw also lets you customize how pictures are saved so you can save them in a file; see the QuickDraw manual for details about customizing.)

Writing to the Scrap

```
FUNCTION ZeroScrap : LongInt;
```

ZeroScrap initializes the desk scrap, clearing its contents; you must call it before the first time you call PutScrap (described below). If no error occurs, ZeroScrap returns Ø; otherwise, it returns an

appropriate Operating System error code.

ZeroScrap also changes the scrapCount field of the record of information provided by InfoScrap. This is useful for testing whether the contents of the desk scrap have changed during the use of a desk accessory. The application can save the value of the scrapCount field when one of its windows is deactivated and a system window is activated. Then, each time through its event loop, it can check to see whether the value of the field has changed. If so, it means the desk accessory called ZeroScrap (and, presumably, PutScrap) and thus changed the contents of the desk scrap.

```
FUNCTION PutScrap (length: LongInt; theType: ResType; source: Ptr) :
        LongInt;
```

PutScrap writes the data pointed to by the source parameter to the desk scrap (whether in memory or on the disk). The length parameter indicates the number of bytes to write, and theType is the data type (which should be different from the type of any data already in the desk scrap). If no error occurs, the function result is Ø; otherwise, it's an appropriate Operating System error code.

(eye)

Don't forget to call ZeroScrap (above) to clear the scrap before your first call to PutScrap.

FORMAT OF THE DESK SCRAP

In general, the desk scrap consists of a series of data items that have the following format:

Number of bytes	Contents
4 bytes	Type (a sequence of four characters)
4 bytes	Length of following data in bytes
n bytes	Data; n must be even (if the above length is odd, include an extra byte)

The standard types are 'TEXT' and 'PICT'. You may use any other four-character sequence for types specific to your application.

The format of the data for the 'TEXT' type is as follows:

Number of bytes	Contents
4 bytes	Number of characters in the text
n bytes	The characters in the text
m bytes	Optional comment: the 4-byte application ID followed by any information desired

The data for the 'PICT' type is a QuickDraw picture, which consists of the size of the picture in bytes, the picture frame, and the picture

definition data (which may include picture comments). See the QuickDraw manual for details.

SUMMARY OF THE SCRAP MANAGER

Constants

```
CONST noTypeErr = -1Ø2;  {there's no data of the requested type}
```

Data Structures

```
TYPE PScrapStuff = ^ScrapStuff;
     ScrapStuff  = RECORD
                     scrapSize:   LongInt;
                     scrapHandle: Handle;
                     scrapCount:  INTEGER;
                     scrapState:  INTEGER;
                     scrapName:   StringPtr
                   END;
```

Routines

Getting Scrap Information

```
FUNCTION InfoScrap : PScrapStuff;
```

Keeping the Scrap on the Disk

```
FUNCTION UnloadScrap : LongInt;
FUNCTION LoadScrap :   LongInt;
```

Reading from the Scrap

```
FUNCTION GetScrap (hDest: Handle; theType: ResType; VAR offset: LongInt)
                  : LongInt;
```

Writing to the Scrap

```
FUNCTION ZeroScrap : LongInt;
FUNCTION PutScrap   (length: LongInt; theType: ResType; source: Ptr) :
                     LongInt;
```

Assembly-Language Information

Constants

```
noTypeErr      .EQU    -1Ø2      ;there's no data of the requested type
```

Scrap Information Data Structure

scrapSize	Size of desk scrap in bytes *** (currently named scrapInfo) ***
scrapHandle	Handle to desk scrap in memory
scrapCount	Count changed by ZeroScrap
scrapState	Positive if desk scrap in memory, Ø if on disk
scrapName	Pointer to name of scrap file

Special Macro Names

Routine name	Macro name
LoadScrap	_LodeScrap
UnloadScrap	_UnlodeScrap

GLOSSARY

application ID: A four-character sequence that you choose to identify your application when you you build it.

desk scrap: The place in memory or on the disk where data that's cut (or copied) and pasted between applications is stored.

scrap file: The file containing the desk scrap.

COMMENTS?

Macintosh User Education encourages your comments on this manual.

- What do you like or dislike about it?

- Were you able to find the information you needed?

- Was it complete and accurate?

- Do you have any suggestions for improvement?

Please send your comments to the author (indicated on the cover page) at 10460 Bandley Drive M/S 3-G, Cupertino CA 95014. Mark up a copy of the manual or note your remarks separately. (We'll return your marked-up copy if you like.)

Thanks for your help!

MACINTOSH USER EDUCATION

The Toolbox Utilities: A Programmer's Guide /TOOLUTIL/UTIL

See Also: Macintosh Operating System Reference Manual
QuickDraw: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Memory Manager: A Programmer's Guide

Modification History:			
	First Draft	C. Rose	5/16/83
	Second Draft (ROM 7)	C. Rose	1/4/84
	Erratum Added	C. Rose	2/8/84

ABSTRACT

This manual describes the Toolbox Utilities, a set of routines and data types in the User Interface Toolbox that perform generally useful operations such as fixed-point arithmetic, string manipulation, and logical operations on bits.

Erratum:

When the Munger function does a replacement operation, it returns the offset of **the first byte past** where the replacement occurred.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Toolbox Utilities, a set of routines and data types in the User Interface Toolbox that perform generally useful operations such as fixed-point arithmetic, string manipulation, and logical operations on bits. *** Eventually it will become part of a comprehensive manual describing the entire Toolbox and Operating System. ***

You should already be familiar with Lisa Pascal. Depending on which Toolbox Utilities you're interested in using, you may also need to know about the Macintosh Operating System's Memory Manager, the Resource Manager, and the basic concepts and structures behind QuickDraw.

This manual begins with a discussion of fixed-point numbers. This is followed by the detailed descriptions of all Toolbox Utility procedures and functions, their parameters, calling protocol, effects, side effects, and so on. Finally, there's a summary of the Toolbox Utilities, for quick reference, followed by a glossary of terms used in this manual. *** The glossary has only two entries, but eventually it will be merged with the glossaries from the other Toolbox and Operating System documentation. ***

FIXED-POINT NUMBERS

The Toolbox Utilities include routines for operating on fixed-point numbers. A fixed-point number is a 32-bit quantity containing an integer part in the high-order word and a fractional part in the low-order word (see Figure 1). Since these numbers occupy the same number of bits as long integers, they could be given the data type LongInt; however, to reflect the different interpretation the bits have as fixed-point numbers, the following data type is defined in the Toolbox Utilities:

```
TYPE Fixed = LongInt;
```

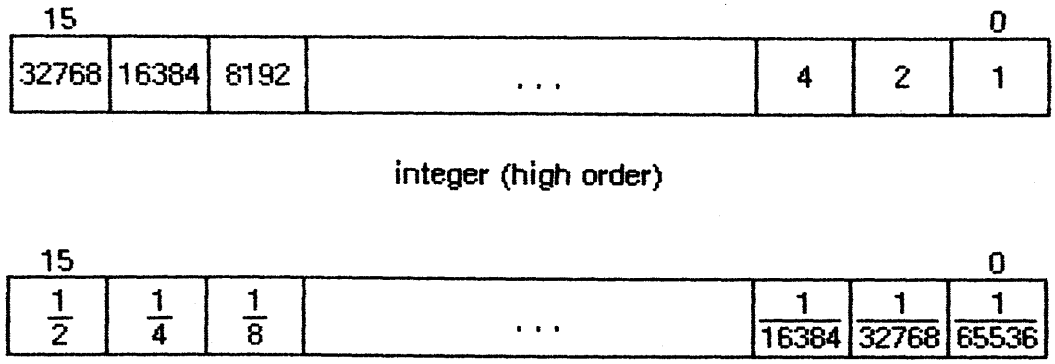


Figure 1. Fixed-Point Numbers

As described in the following section, there are Toolbox Utility routines for converting an integer numerator and denominator into a fixed-point number, multiplying two fixed-point numbers, and rounding a fixed-point number to the nearest integer. You can also use the general-purpose function HiWord (or LoWord) to extract the integer (or fractional) part of a fixed-point number.

TOOLBOX UTILITY ROUTINES

This section describes all the Toolbox Utility procedures and functions. They're presented in their Pascal form; for information on using them from assembly language, see "Using the Toolbox from Assembly Language" *** doesn't exist, but see "Using QuickDraw from Assembly Language" in the QuickDraw manual ***.

Fixed-Point Arithmetic

See also HiWord and LoWord under "Other Operations on Long Integers" below.

```
FUNCTION FixRatio (numerator,denominator: INTEGER) : Fixed;
```

FixRatio returns the fixed-point number having the given numerator and denominator (either of which may be any signed integer).

```
FUNCTION FixMul (a,b: Fixed) : Fixed;
```

FixMul multiplies the given fixed-point numbers and returns the result.

```
FUNCTION FixRound (x: Fixed) : INTEGER;
```

FixRound rounds the given fixed-point number to the nearest integer and returns the result.

String Manipulation

These routines use the StringHandle data type, which is defined in the Toolbox Utilities as follows:

```
TYPE StringPtr    = ^Str255;
     StringHandle = ^StringPtr;
```

```
FUNCTION NewString (s: Str255) : StringHandle;
```

NewString allocates the string specified by s as a relocatable object on the heap and returns a handle to it.

```
PROCEDURE SetString (h: StringHandle; s: Str255);
```

SetString sets the string whose handle is passed in h to the string specified by s.

```
FUNCTION GetString (stringID: INTEGER) : StringHandle;
```

GetString returns a stringHandle to the string having the given resource ID, reading it from the resource file if necessary. It calls the Resource Manager function GetResource('STR ',stringID).

Byte Manipulation

```
FUNCTION Munger (h: Handle; offset: LongInt; ptr1: Ptr; len1: LongInt;
            ptr2: Ptr; len2: LongInt) : LongInt;
```

*** There's currently no Pascal interface to this routine; declare it as EXTERNAL in your program. ***

Munger manipulates bytes in the string of bytes (the "destination string") to which h is a handle. The offset parameter specifies a byte offset into the destination string. The exact nature of the operation done by Munger depends on the values of the remaining parameters, two pointer/length pairs. In general, (ptr1,len1) defines a substring to be replaced by the second substring (ptr2,len2). If these four parameters are all positive and nonzero, Munger looks for (ptr1,len1) in the destination string, starting from the given offset and ending at the end of the string; the first occurrence it finds is replaced by (ptr2,len2), and the offset at which the replacement occurred is returned. Figure 2 illustrates this; the bytes represent ASCII characters as shown.

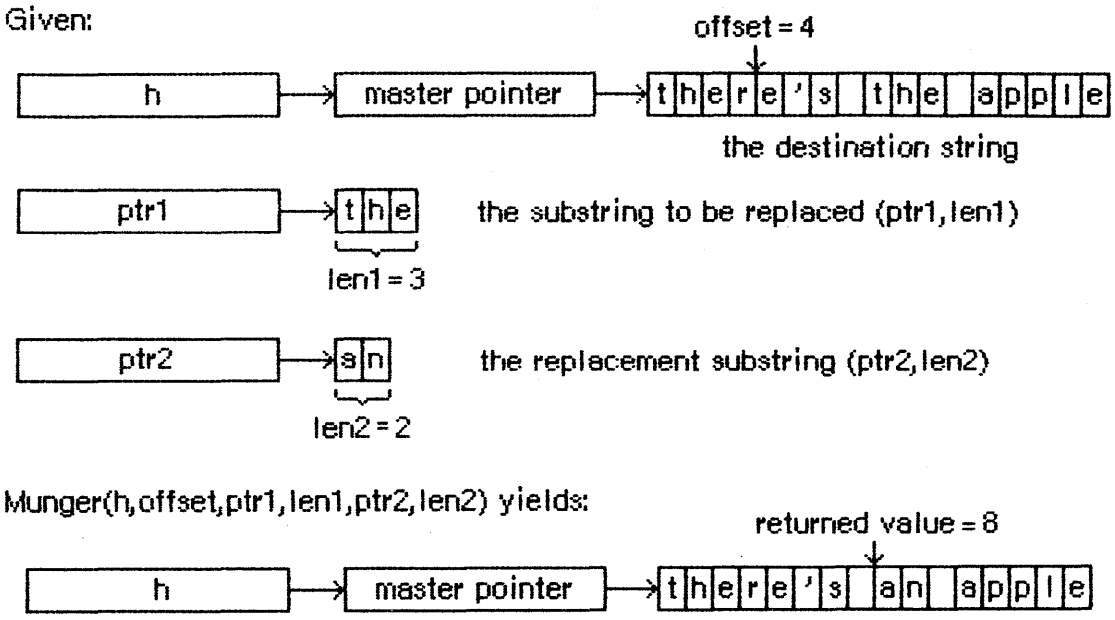


Figure 2. Munger Function

Different operations occur if any of the pointers or lengths is ∅:

- If ptr1 is ∅, the substring of length len1 starting at the given offset is replaced by (ptr2,len2). If len1 is negative, the substring from the given offset to the end of the destination string is replaced by (ptr2,len2).

- If len1 is ∅, the substring (ptr2,len2) is simply inserted at the given offset.

- If ptr2 is ∅, the destination string isn't changed; Munger just returns the offset at which it found (ptr1,len1).

- If len2 is ∅, the replacement substring is empty, so (ptr1,len1) is deleted rather than replaced.

Munger returns the offset at which the operation occurred--whether replacement, insertion, deletion, or just location of a substring. It returns a negative value if it can't find (ptr1,len1) in the destination string.

(eye)

Be careful not to specify an offset that's greater than the length of the destination string, or unpredictable things may happen.

Bit Manipulation

These routines manipulate a bit in data pointed to by a given pointer. A bit number indicates which bit; it starts at Ø for the high-order bit of the first byte pointed to and may be any positive long integer specifying an offset from that bit (see Figure 3).

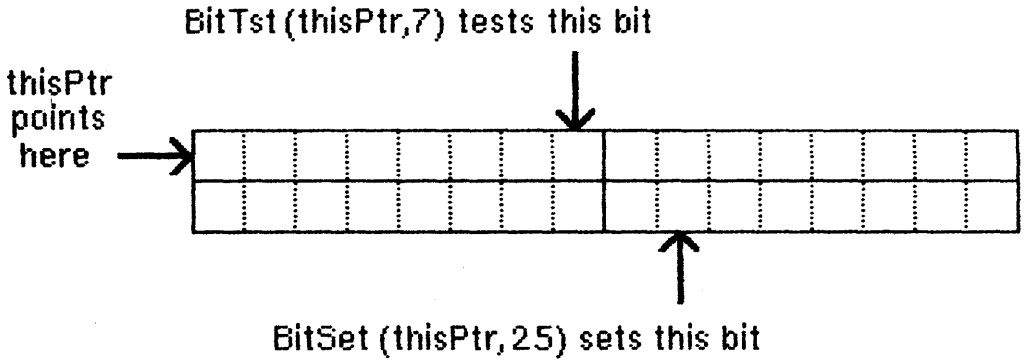


Figure 3. Bit Numbering for Utility Routines

(hand)

Note that this bit numbering is the opposite of the MC68ØØØ bit numbering.

```
FUNCTION BitTst (bytePtr: Ptr; bitNum: LongInt) : BOOLEAN;
```

BitTst tests whether a given bit is set and returns TRUE if so or FALSE if not. The bit is specified by bitNum, an offset from the high-order bit of the byte pointed to by bytePtr.

```
PROCEDURE BitSet (bytePtr: Ptr; bitNum: LongInt);
```

BitSet sets the bit specified by bitNum, an offset from the high-order bit of the byte pointed to by bytePtr.

```
PROCEDURE BitClr (bytePtr: Ptr; bitNum: LongInt);
```

BitSet clears the bit specified by bitNum, an offset from the high-order bit of the byte pointed to by bytePtr.

Logical Operations

FUNCTION BitAnd (long1,long2: LongInt) : LongInt;

BitAnd returns the result of the AND logical operation on the bits comprising the given long integers (long1 AND long2).

FUNCTION BitOr (long1,long2: LongInt) : LongInt;

BitOr returns the result of the OR logical operation on the bits comprising given long integers (long1 OR long2).

FUNCTION BitXor (long1,long2: LongInt) : LongInt;

BitXor returns the result of the XOR logical operation on the bits comprising the given long integers (long1 XOR long2).

FUNCTION BitNot (long: LongInt) : LongInt;

BitXor returns the result of the NOT logical operation on the bits comprising the given long integer.

FUNCTION BitShift (long: LongInt; count: INTEGER) : LongInt;

BitShift logically shifts the bits of the given long integer. Count specifies the direction and extent of the shift, and is taken modulo 31. If count is positive, BitShift shifts that many positions to the left; if count is negative, it shifts to the right. Zeros are shifted into empty positions at either end.

Other Operations on Long Integers

FUNCTION HiWord (x: LongInt) : INTEGER;

HiWord returns the high-order word of the given long integer. One use of this function is to extract the integer part of a fixed-point number.

FUNCTION LoWord (x: LongInt) : INTEGER;

LoWord returns the low-order word of the given long integer. One use of this function is to extract the fractional part of a fixed-point number.

```
PROCEDURE LongMul (a,b: LongInt; VAR dest: Int64Bit);
```

LongMul multiplies the given long integers and returns the result in dest, which has the following data type:

```
TYPE Int64Bit = RECORD
                  hiLong:  LongInt;
                  loLong:  LongInt
                END;
```

Graphics Utilities

```
FUNCTION GetIcon (iconID: INTEGER) : Handle;
```

GetIcon returns a handle to the icon having the given resource ID, reading it from the resource file if necessary. It calls the Resource Manager function GetResource('ICON',iconID).

```
PROCEDURE PlotIcon (theRect: Rect; theIcon: Handle);
```

*** There's currently no Pascal interface to this routine; declare it as EXTERNAL in your program. ***

PlotIcon draws the icon whose handle is theIcon in the rectangle theRect, which is in the local coordinates of the current grafPort. It calls the QuickDraw procedure CopyBits and uses the srcCopy transfer mode. (You must have initialized QuickDraw before calling PlotIcon.)

```
FUNCTION GetPattern (patID: INTEGER) : PatHandle;
```

GetIcon returns a handle to the pattern having the given resource ID, reading it from the resource file if necessary. It calls the Resource Manager function GetResource('PAT ',patID). The PatHandle data type is *** not yet, but soon will be *** defined in the Toolbox Utilities as follows:

```
TYPE PatPtr    = ^Pattern;
     PatHandle = ^PatPtr;
```

```
FUNCTION GetCursor (cursorID: INTEGER) : CursHandle;
```

GetIcon returns a handle to the cursor having the given resource ID, reading it from the resource file if necessary. It calls the Resource Manager function GetResource('CURS',cursorID). The CursHandle data type is *** not yet, but soon will be *** defined in the Toolbox Utilities as follows:

```
TYPE CursPtr    = ^Cursor;
     CursHandle = ^CursPtr;
```

```
PROCEDURE ShieldCursor (left,top,right,bottom: INTEGER);
```

Given the global coordinates of a rectangle, ShieldCursor removes the cursor from the screen if the cursor and the rectangle intersect.

```
FUNCTION GetPicture (pictureID: INTEGER) : PicHandle;
```

GetPicture returns a handle to the picture having the given resource ID, reading it from the resource file if necessary. It calls the Resource Manager function GetResource('PICT',pictureID). The PicHandle data type is defined in QuickDraw.

SUMMARY OF THE TOOLBOX UTILITIES

```
TYPE Fixed = LongInt;

     Int64Bit = RECORD
                  hiLong:  LongInt;
                  loLong:  LongInt
                END;

     StringPtr    = ^Str255;
     StringHandle = ^StringPtr;

     CursPtr    = ^Cursor;
     CursHandle = ^CursPtr;

     PatPtr    = ^Pattern;
     PatHandle = ^PatPtr;
```

Fixed-Point Arithmetic

```
FUNCTION FixRatio (numerator,denominator: INTEGER) : Fixed;
FUNCTION FixMul   (a,b: Fixed) : Fixed;
FUNCTION FixRound (x: Fixed) : INTEGER;
```

String Manipulation

```
FUNCTION  NewString (s: Str255) : StringHandle;
PROCEDURE SetString (h: StringHandle; s: Str255);
FUNCTION  GetString (stringID: INTEGER) : StringHandle;
```

Byte Manipulation

```
FUNCTION Munger (h: Handle; offset: LongInt; ptr1: Ptr; len1: LongInt;
                 ptr2: Ptr; len2: LongInt) : LongInt;
```

Bit Manipulation

```
FUNCTION  BitTst (bytePtr: Ptr; bitNum: LongInt) : BOOLEAN;
PROCEDURE BitSet (bytePtr: Ptr; bitNum: LongInt);
PROCEDURE BitClr (bytePtr: Ptr; bitNum: LongInt);
```

Logical Operations

```
FUNCTION BitAnd   (long1,long2: LongInt) : LongInt;
FUNCTION BitOr    (long1,long2: LongInt) : LongInt;
FUNCTION BitXor   (long1,long2: LongInt) : LongInt;
FUNCTION BitNot   (long: LongInt) : LongInt;
FUNCTION BitShift (long: LongInt; count: INTEGER) : LongInt;
```

Other Operations on Long Integers

```
FUNCTION  HiWord  (x: LongInt) : INTEGER;
FUNCTION  LoWord  (x: LongInt) : INTEGER;
PROCEDURE LongMul (a,b: LongInt; VAR dest: Int64Bit);
```

Graphics Utilities

```
FUNCTION  GetIcon      (iconID: INTEGER) : Handle;
PROCEDURE PlotIcon     (theRect: Rect; theIcon: Handle);
FUNCTION  GetPattern   (patID: INTEGER) : PatHandle;
FUNCTION  GetCursor    (cursorID: INTEGER) : CursHandle;
PROCEDURE ShieldCursor (left,top,right,bottom: INTEGER);
FUNCTION  GetPicture   (pictureID: INTEGER) : PicHandle;
```

GLOSSARY

fixed-point number: A 32-bit quantity containing an integer part in the high-order word and a fractional part in the low-order word.

icon: A 32-by-32 bit image that represents an object, concept, or message.

COMMENTS?

Macintosh User Education encourages your comments on this manual.

- What do you like or dislike about it?

- Were you able to find the information you needed?

- Was it complete and accurate?

- Do you have any suggestions for improvement?

Please send your comments to the author (indicated on the cover page) at 10460 Bandley Drive M/S 3-G, Cupertino CA 95014. Mark up a copy of the manual or note your remarks separately. (We'll return your marked-up copy if you like.)

Thanks for your help!