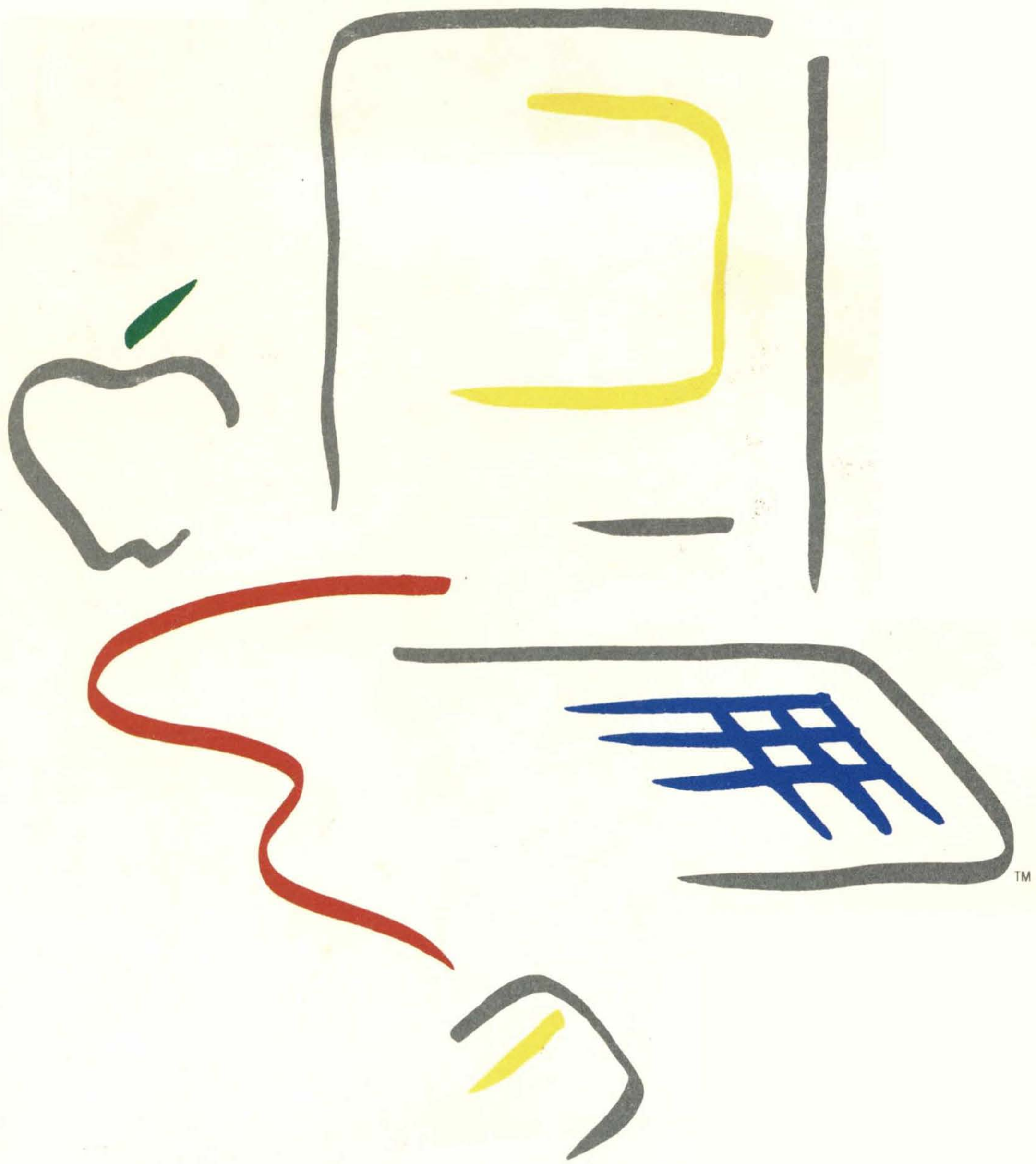
Inside Macintosh™



Promotional Edition

A Letter from the Macintosh Developers Group 15 March 1985

DearReader:

After many months of work the Macintosh Division's User Education Group (responsible for all the Macintosh documentation) has completed the manuscript for *Inside Macintosh.* We've finalized production arrangements with a major publisher and you can expect to see the final edition at better bookstores everywhere by late summer '85. However, we can't wait that long and don't expect you to either. We've therefore produced this special Promotional Edition to handle the demand for *Inside Macintosh* until the final edition becomes available. The contents of this edition are still preliminary and subject to change; the final edition will include many updates and corrections. The production quality of the final edition will be significantly improved from this inexpensive edition.

Now, here are answers to some questions we anticipate:

Q. I purchased the three-ring binder version of *Inside Macintosh* from your mail-house for $100 and also bought the Software Supplement for $100. Is this Promotional Edition the final copy I'm supposed to receive for purchasing the Software Supplement?

A. No. As promised, Supplement owners will receive a copy of the final version when it's available.

Q. How can I get Macintosh developer utilities, example programs, example source code, the libraries I need to do Lisa Pascal/Macintosh cross-development work, and additional copies of this manual?

A.The Software Supplement consists of: 1) useful Macintosh utilities, example programs, and example source code, 2) the interface files, equate files, and trap definitions in both Macintosh and Lisa readable format, 3) all of the libraries required for Lisa Pascal/Macintosh cross-development, 4) a new Lisa Pascal Compiler, which supports SANE numerics, and 5) a copy of the final published edition of *Inside Macintosh* (this will be sent to you when available). The price for the Software Supplement is $100. As of April '85 the Software Supplement has been frozen to correspond to *Inside Macintosh* and automatic updates will no longer be included in the Software Supplement price. We will, however, inform Supplement owners of other products and utilities as they become available. You may also order additional copies of this special Promotional Edition of *Inside Macintosh* for $25 per copy.

You can order the Software Supplement and/or copies of this Promotional Edition of *Inside Macintosh* by writing to (California residents please add 6.5% sales tax) :

Apple Computer, Inc.
467 Saratoga Avenue Suite 621
San Jose, CA 95129
(408) 988-6009

Q. Is there a good way to keep up-to-date with new utilities and technical notes and at the same time stay in touch with other developers?

A. We've found that electronic distribution is a very cost-effective, timely way to keep the developer world up-to-date. At least two major on-line services, Delphi and Compuserve, host the MicroNetworked Apple Users' Group (MAUG)--an electronic service open to all people who are interested in or have information to share about Apple products. MAUG is an independently run service and is not affiliated with Apple Computer, Inc. If you have a modem and communication software you can sign-on and download the latest developer utilities from the Macintosh Software Supplement, example programs, technical notes, and documentation. You can also carry on electronic conversations with hundreds of other Macintosh developers. For more information on these services please write to either:

Delphi
3 Blackstone Street
Cambridge, MA 02139
(617) 491-3393
(800) 544-4005 (toll-free outside of Massachusetts)

Compuserve
5000 Arlington Centre Boulevard
Columbus, OH 43220
(614) 457-8600

All of us in the Macintosh Division would like to thank you for your support of Macintosh development. We'll continue to provide the programs, tools, and documentation to assist your efforts.

We'd love to hear from you on any topic related to Macintosh Development. Send your letters to Apple Computer, Macintosh Developers Group, Mail Stop 4T, 20525 Mariani Avenue, Cupertino CA 95014.

Inside Macintosh

Promotional Edition

Copyright

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

Simultaneously published in the U.S.A. and Canada.

Limitation on Warranty and Liabilities

Inside Macintosh is a working document used by Apple's own programmers and developers which describes the hardware and software associated with the Macintosh computer. This is a preliminary edition that is incomplete and may contain errors. Apple itself does not rely solely on this document, but rather subjects all of its products to extensive testing prior to introduction. Further, Apple is continuing to improve its products and the contents of this manual may be obsoleted as a result of changes to Apple products. Accordingly, Apple makes no warranty that this manual is accurate or complete and notifies all readers that the specifications of its products are subject to change without notice.

APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS ACCURACY, CONTENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD " AS IS" AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO THE USE OF AND RELIANCE ON THIS MANUAL.

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

No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

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

Contents

MACINTOSH USER EDUCATION

INSIDE MACINTOSH: A ROAD MAP

/ROAD.MAP/ROAD

See Also: Macintosh User Interface Guidelines
Macintosh Memory Management: An Introduction
Programming Macintosh Applications in Assembly Language
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
The Control Manager: A Programmer's Guide
The Menu Manager: A Programmer's Guide
TextEdit: A Programmer's Guide
The Dialog Manager: 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
Macintosh Packages: A Programmer's Guide
The Memory Manager: A Programmer's Guide
The Segment Loader: A Programmer's Guide
The File Manager: A Programmer's Guide
Printing from Macintosh Applications
The Device Manager: A Programmer's Guide
The Sound Driver: A Programmer's Guide
The Vertical Retrace Manager: A Programmer's Guide
The Operating System Utilities: A Programmer's Guide
The Structure of a Macintosh Application
Putting Together a Macintosh Application
Index to Technical Documentation

Modification History:	First Draft (ROM 4.4)	Caroline Rose	8/8/83
	Second Draft (ROM 7)	Caroline Rose	12/22/83
	Third Draft	Caroline Rose	9/10/84

ABSTRACT

This manual introduces you to the Macintosh technical documentation and the "inside" of Macintosh: the Operating System and other routines that your 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 documentation. It also presents a simple example program.

Since the last draft, changes and additions have been made to the overviews, an example program has been added, and the structure of a typical Inside Macintosh manual is discussed.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual introduces you to the Macintosh technical documentation and the "inside" of Macintosh: the Operating System and User Interface Toolbox routines that your 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 documentation. To orient you to the software, it presents a simple example program. *** Eventually it will become the preface and introductory chapter in the comprehensive Inside Macintosh manual. ***

ABOUT INSIDE MACINTOSH

Inside Macintosh *** (currently a set of separate manuals) *** tells you what you need to know to write software for the Macintosh. Although directed mainly toward programmers writing standard Macintosh applications, it also contains the information necessary for writing simple utility programs, desk accessories, device drivers, or any other Macintosh software. It includes:

- the user interface guidelines for applications on the Macintosh
- a complete description of the routines available for your program to call (both those built into the Macintosh and others on disk), along with related concepts and background information
- a description of the Macintosh hardware *** (forthcoming) ***

It does **not** include:

- information about getting started as a developer (for that, see the Apple 32 Developer's Handbook, available from Apple Computer's Software Industry Relations)
- any information that's specific to the development system being used, except where indicated *** (The manual Putting Together a Macintosh Application will not be part of the final Inside Macintosh.) ***

The routines you'll need to call are written in assembly language, but they're also accessible from high-level languages. The development system currently available from Apple supports Lisa Pascal and includes Pascal interfaces to all the routines (except for a few that are called only from assembly language). Inside Macintosh documents these Pascal interfaces; if you're using a development system that supports a different high-level language, its documentation should tell you how to apply the information presented here to that system.

Inside Macintosh is intended to serve the needs of both Pascal and assembly-language programmers. Every routine is shown in its Pascal form (if it has one), but assembly-language programmers are told how to translate this to assembly code. Information of interest only to assembly-language programmers is isolated and labeled so that Pascal programmers can conveniently skip it.

Familiarity with Lisa Pascal is recommended for all readers, since it's used for most examples. Lisa Pascal is described in the *Pascal Reference Manual for the Lisa*. You should also be familiar with the basic information that's in *Macintosh*, the owner's guide, and have some experience using a standard Macintosh application (such as MacWrite).

Everything You Know Is Wrong

On an innovative system like the Macintosh, programs don't look quite the way they do on other systems. For example, instead of carrying out a sequence of steps in a predetermined order, your program is driven primarily by user actions (such as clicking and typing) whose order cannot be predicted. You'll probably find that many of your preconceptions about how to write applications don't apply here. Because of this, and because of the sheer volume of information in *Inside Macintosh*, it's essential that you read the Road Map *** (the rest of this manual) ***. It will help you get oriented and figure out where to go next.

Conventions

The following notations are used in *Inside Macintosh* to draw your attention to particular items of information:

(note)
A note that may be interesting or useful

(warning)
A point you need to be cautious about

Assembly-language note: A note of interest to assembly-language programmers only *** (in final manual, may instead be a shaded note or warning) ***

[No trap macro]

This notation is of interest only to assembly-language programmers *** (may be shaded in final manual) ***; it's explained along with other general information on using assembly language in the manual *Programming Macintosh Applications in Assembly Language*.

The Structure of a Typical Manual

*** This section refers to "manuals" for the time being; when the individual manuals become chapters of Inside Macintosh, this will be changed to "chapters". ***

Most manuals of Inside Macintosh have the same structure, as described below. Reading through this now will save you a lot of time and effort later on. It contains important hints on how to find what you're looking for within this vast amount of technical documentation.

Every manual begins with a very brief description of its subject and a list of what you should already know before reading that manual. Then there's a section called, for example, "About the Window Manager", which gives you more information about the subject, telling you what you can do with it in general, elaborating on related user interface guidelines, and introducing terminology that will be used in the manual. This is followed by a series of sections describing important related concepts and background information; unless they're noted to be for advanced programmers only, you'll have to read them in order to understand how to use the routines described later.

Before the routine descriptions themselves, there's a section called, for example, "Using the Window Manager". It introduces you to the routines, telling you how they fit into the general flow of an application program and, most important, giving you an idea of which ones you'll need to use. Often you'll need only a few routines out of many to do basic operations; by reading this section, you can save yourself the trouble of learning routines you'll never use.

Then, for the details about the routines, read on to the next section. It gives the calling sequence for each routine and describes all the parameters, effects, side effects, and so on.

Following the routine descriptions, there may be some sections that won't be of interest to all readers. Usually these contain information about advanced techniques, or behind-the-scenes details for the curious.

For review and quick reference, each manual ends with a summary of the subject matter, including the entire Pascal interface and a subsection for assembly-language programmers. *** For now, this is followed by a glossary of terms used in the manual. Eventually, all the individual glossaries will be combined into one. ***

OVERVIEW OF THE SOFTWARE

The routines available for use in Macintosh programs are divided according to function, into what are in most cases called "managers" of the application feature that they support. As shown in Figure 1 on the following page, most 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 input and output, memory management, and interrupt handling. The User Interface Toolbox is a level above the Operating System; it helps you implement the standard Macintosh user interface in your application. The Toolbox calls the Operating System to do low-level operations, and you'll also call the Operating System directly yourself.

RAM-based software is available as well. In most cases this software performs specialized operations that aren't integral to the user interface but may be useful to some applications (such as printing and floating-point arithmetic).

The Toolbox and Other High-Level Software

The Macintosh User Interface Toolbox provides a simple means of constructing application programs that conform to the standard Macintosh user interface. By offering a common set of routines that every application calls to implement the user interface, the Toolbox not only ensures familiarity and consistency for the user 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 various parts of the Toolbox in rough order of their relative level. There are many interconnections between these parts; the higher ones often call those at the lower levels. A brief description of each part is given below, to help you figure out which ones you'll need to learn more about. Details are given in the Inside Macintosh documentation on that part of the Toolbox. The basic Macintosh terms used below are explained in the Macintosh owner's guide.

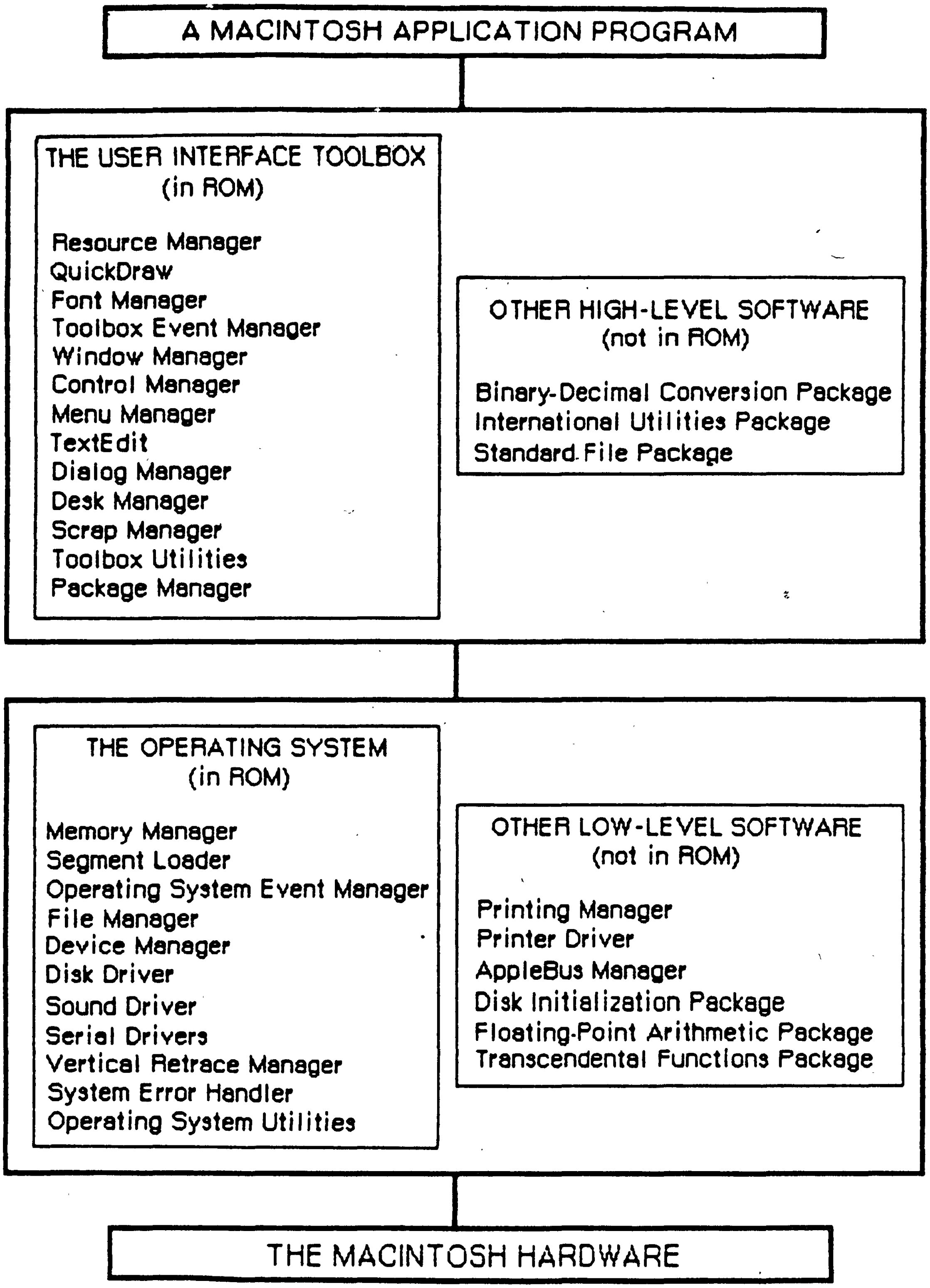


Figure 1. Overview

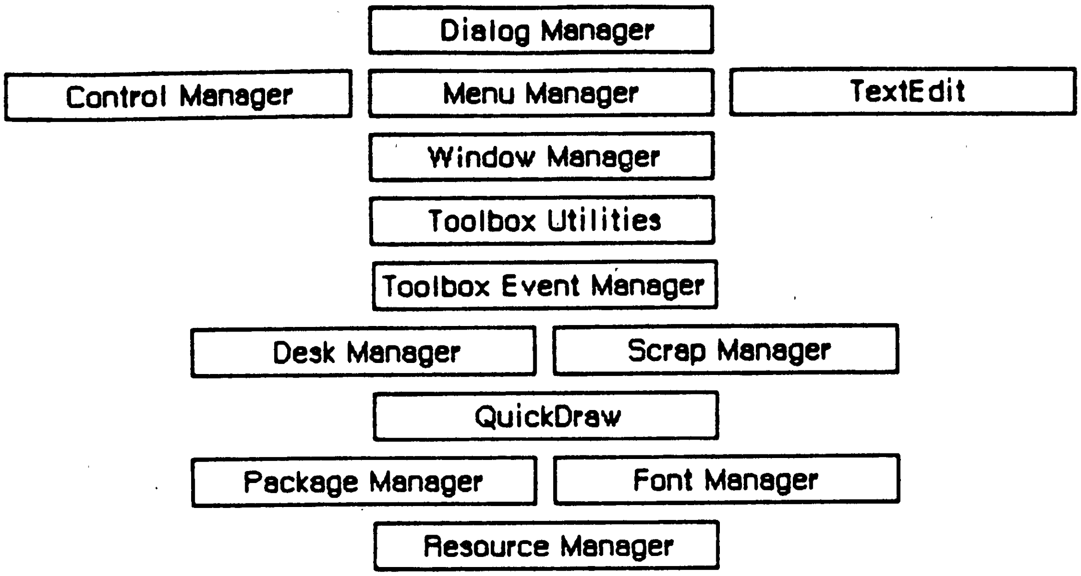


Figure 2. Parts of the Toolbox

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 I-beam pointer for inserting text, from a shared system file. When you call other parts of the Toolbox 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 because they're basic to so many other Toolbox operations.

Graphics are an important part of every Macintosh application. All graphic operations on the Macintosh are performed by QuickDraw. To draw something on the screen, you'll often call one of the other parts of the Toolbox, but it will in turn call QuickDraw. You'll also call QuickDraw directly, usually to draw inside a window, or just to set up constructs like rectangles that you'll need when making other Toolbox calls. 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 your application includes a font menu, you need to know only a minimal amount about 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. For example, the Window Manager 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, 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, including cutting and pasting text within a document via the Clipboard, your application can call TextEdit. TextEdit also handles basic formatting such as word wraparound and justification. You can use it just to display text if you like.

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 alerts the user with a box containing a message or with 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 work properly.

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

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

The final part of the Toolbox, the Package Manager, lets you use RAM-based software called packages. The Standard File Package will be called by every application whose File menu includes the standard commands for saving and opening documents; it presents the standard user interface for specifying the document. Some of the Macintosh packages can be seen as extensions to the Toolbox Utilities: the Binary-Decimal Conversion Package converts integers to decimal strings and vice versa, and the International Utilities Package gives you access to country-dependent information such as the formats for numbers, currency, dates, and times.

The Operating System and Other Low-Level Software

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 program code into memory to be executed. Your program can be loaded all at once, or you can divide it up into dynamically loaded segments to economize on memory usage. The Segment Loader also serves as a bridge between the Finder and your application, letting you know whether the application has to open or print a document on the desktop when it starts up.

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 to the application, along with higher-level, software-generated events added at the Toolbox level.) Your program will ordinarily deal only with the Toolbox Event Manager and rarely call the Operating System Event Manager 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 3 1/2-inch disks.

- The Sound Driver controls sound generation, including music composed of up to 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.

The above drivers are all in ROM; other drivers are RAM-based. There's a Serial Driver in RAM as well as the one in ROM, and there's a Printer Driver in RAM that enables applications to print information on any variety of printer via the same interface (called the Printing Manager). The AppleBus Manager is an interface to a pair of RAM drivers that enable programs to send and receive information via an AppleBus network. More RAM drivers can be added independently or built on the existing drivers. For example, the Printer Driver was built on the Serial Driver, and a music driver could be built on the Sound Driver.

The Macintosh video circuitry generates a vertical retrace interrupt 6Ø times a second. An application can 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.

If a fatal error occurs while your application is running (for example, if it runs out of memory), the System Error Handler assumes control. The System Error Handler displays a box containing an error message and provides a mechanism for the user to start up the system again or resume execution of the application.

The Operating System Utilities perform miscellaneous operations such as getting the date and time, finding out the user's preferred speaker volume and other preferences, and doing simple string comparison. (More sophisticated string comparison routines are available in the International Utilities Package.)

Finally, there are three Macintosh packages that perform low-level operations: the Disk Initialization Package, which the Standard File Package calls to initialize and name disks; the Floating-Point Arithmetic Package; and the Transcendental Functions Package.

A SIMPLE EXAMPLE PROGRAM

To illustrate various commonly used parts of the software, this section presents an extremely simple example of a Macintosh application program. Though too simple to be practical, this example shows the overall structure that every application program will have, and it does many of the basic things every application will do. By looking it over, you can become more familiar with the software and see how your own program code will be structured.

The example program's source code is shown in Figure 4, which begins on page 15. A lot of comments are included so that you can see which part of the Toolbox or Operating System is being called and what operation is being performed. These comments, and those that follow below, may contain terms that are unfamiliar to you, but for now just read along to get the general idea. All the terms are explained at length within Inside Macintosh. If you want more information right away, you can look up the terms in the Glossary or the Index *** (currently the individual glossaries in the various manuals, and the manual Index to Technical Documentation) ***

The application, called Samp, displays a single, fixed-size window in which the user can enter and edit text (see Figure 3). It has three menus: the standard Apple menu, from which desk accessories can be chosen; a File menu, containing only a Quit command; and an Edit menu, containing the standard editing commands Undo, Cut, Copy, Paste, and Clear. The Backspace key may be used to delete, and Shift-clicking will extend or shorten a selection. The user can move the document window around the desktop by dragging it by its title bar.

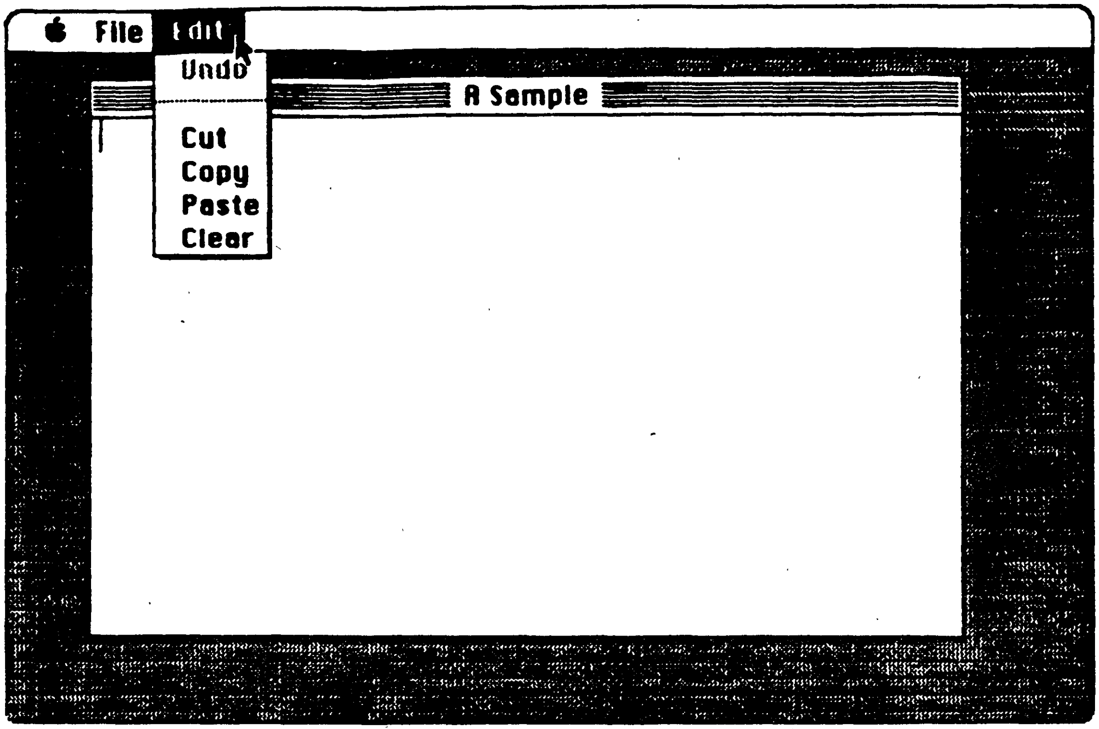


Figure 3. The Samp Application

The Undo command doesn't work in the application's document window, but it and all the other editing commands do work in any desk accessories that allow them (Note Pad, for example). Some standard features this simple example doesn't support are as follows:

- Text cannot be cut (or copied) and pasted between the document and a desk accessory.

- The pointer remains an arrow rather than changing to an I-beam within the document.

- The standard keyboard equivalents--Command-Z, X, C, and V for Undo, Cut, Copy, and Paste--aren't in the Edit menu. They won't work in the document window (but they will work in desk accessories that allow those commands).

Because the File menu contains only a Quit command, the document can't be saved or printed. Also, the application doesn't have an "About

Samp" command as the first item in its Apple menu, nor does it present any dialog boxes or alerts. All of these features and more are illustrated in programs in the Sample Macintosh Programs manual *** (forthcoming) ***.

In addition to the code shown in Figure 4, the Samp application has a resource file that includes the data listed below. The program uses the numbers in the second column to identify the resources; for example, it makes a Menu Manager call to get menu number 128 from the resource file.

Resource	Resource ID	Description
Menu	128	Menu with the apple symbol as its title and no commands in it
Menu	129	File menu with one command, Quit
Menu	13Ø	Edit menu with the commands Undo (dimmed), Cut, Copy, Paste, and Clear, in that order, with a dividing line between Undo and Cut
Window template	128	Document window without a size box; top left corner of (5Ø,4Ø) on QuickDraw's coordinate plane, bottom right corner of (3ØØ,45Ø); title "A Sample"; no close box

Each menu resource also contains a "menu ID" that's used to identify the menu when the user chooses a command from it; for all three menus, this ID is the same as the resource ID.

(note)

> To create a resource file with the above contents, you can use the Resource Editor *** (for now, the Resource Compiler) *** or any similar program that may be available on the development system you're using; for more information, see the documentation for that program. *** The Resource Compiler is documented in Putting Together a Macintosh application. The Resource Compiler input file for the Samp application is shown in the appendix of this manual. All these files will eventually be provided to developers by Macintosh Technical Support. ***

The program starts with a USES clause that specifies all the necessary Pascal interface files. (The names shown are for the Lisa Workshop development system, and may be different for other systems.) This is followed by declarations of some useful constants, to make the source code more readable. Then there are a number of variable declarations, some having simple Pascal data types and others with data types defined in the Pascal interface files (like Rect and WindowPtr). Variables used in the program that aren't declared here are global variables defined in the interface to QuickDraw.

The variable declarations are followed by two procedure declarations: SetUpMenus and DoCommand. You can understand them better after looking at the main program and seeing where they're called.

The program begins with a standard initialization sequence. Every application will need to do this same initialization (in the order shown), or something close to it.

Additional initialization needed by the program follows. This includes setting up the menus and the menu bar (by calling SetUpMenus) and creating the application's document window (reading its description from the resource file and displaying it on the screen).

The heart of every application program is its main event loop, which repeatedly calls the Toolbox Event Manager to get events and then responds to them as appropriate. The most common event is a press of the mouse button; depending on where it was pressed, as reported by the Window Manager, the sample program may execute a command, move the document window, make the window active, or pass the event on to a desk accessory. The DoCommand procedure takes care of executing a command; it looks at information received by the Menu Manager to determine which command to execute.

Besides events resulting directly from user actions such as pressing the mouse button or a key on the keyboard, events are detected by the Window Manager as a side effect of those actions. For example, when a window changes from active to inactive or vice versa, the Window Manager tells the Toolbox Event Manager to report it to the application program. A similar process happens when all or part of a window needs to be updated (redrawn). The internal mechanism in each case is invisible to the program, which simply responds to the event when notified.

The main event loop terminates when the user takes some action to leave the program--in this case, when the Quit command is chosen.

That's it! Of course, the program structure and level of detail will get more complicated as the application becomes more complex, and every actual application will be more complex than this one. But each will be based on the structure illustrated here.

```
PROGRAM Samp;

{ Samp -- A small sample application written in Pascal by Macintosh User Education }
{ It displays a single, fixed-size window in which the user can enter and edit text. }

USES  {SU Obj/MemTypes } MemTypes,   {basic Memory Manager data types}
      {SU Obj/QuickDraw} QuickDraw,  {interface to QuickDraw}
      {SU Obj/OSIntf   } OSIntf,     {interface to the Operating System}
      {SU Obj/ToolIntf } ToolIntf;   {interface to the Toolbox}

CONST appleID = 128;       {resource IDs/menu IDs for Apple, File, and Edit menus}
      fileID = 129;
      editID = 130;
      appleM = 1;          {index for each menu in array of menu handles}
      fileM = 2;
      editM = 3;
      menuCount = 3;       {total number of menus}
      windowID = 128;      {resource ID for application's window}
      undoCommand = 1;     {menu item numbers identifying commands in Edit menu}
      cutCommand = 3;
      copyCommand  = 4;
      pasteCommand = 5;
      clearCommand = 6;

VAR myMenus: ARRAY [1..menuCount] OF MenuHandle;
    dragRect, txRect: Rect;
    extended, doneFlag: BOOLEAN;
    myEvent: EventRecord;
    wRecord: WindowRecord;
    myWindow, whichWindow: WindowPtr;
    textH: TEHandle;


PROCEDURE SetUpMenus;
{ Set up menus and menu bar }

  VAR i: INTEGER;

  BEGIN
  myMenus[appleM] := GetMenu(appleID); {read Apple menu from resource file}
  AddResMenu(myMenus[appleM],'DRVR');  {add desk accessory names to Apple menu}
  myMenus[fileM] := GetMenu(fileID);   {read File menu from resource file}
  myMenus[editM] := GetMenu(editID);   {read Edit menu from resource file}
  FOR i:=1 TO menuCount DO InsertMenu(myMenus[i],0);  {install menus in menu bar }
  DrawMenuBar;                                        { and draw menu bar}
  END;    {of SetUpMenus}


PROCEDURE DoCommand (mResult: LONGINT);
{ Execute command specified by mResult, the result of MenuSelect }

  VAR theItem, temp: INTEGER;
      name: Str255;

  BEGIN
  theItem := LoWord(mResult);           {call Toolbox Utility routine to get }
                                        { menu item number from low-order word}
```

Figure 4. Example Program

```
  CASE HiWord(mResult) OF                     {case on menu ID in high-order word}

  appleID:
    BEGIN                                      {call Menu Manager to get desk accessory }
    GetItem(myMenus[appleM],theItem,name);    { name, and call Desk Manager to open }
    temp := OpenDeskAcc(name);                 { accessory (OpenDeskAcc result not used)}
    SetPort(myWindow);                         {call QuickDraw to restore application }
    END;    {of appleID}                       { window as grafPort to draw in (may have }
                                               { been changed during OpenDeskAcc)}
  fileID:
    doneFlag := TRUE;             {quit (main loop repeats until doneFlag is TRUE)}

  editID:
    BEGIN                         {call Desk Manager to handle editing command if }
    IF NOT SystemEdit(theItem-1)  { desk accessory window is the active window}
      THEN                        {application window is the active window}
        CASE theItem OF           {case on menu item (command) number}

        cutCommand:    TECut(textH);      {call TextEdit to handle command}
        copyCommand:   TECopy(textH);
        pasteCommand:  TEPaste(textH);
        clearCommand:  TEDelete(textH);

        END;    {of item case}
    END;    {of editID}

  END;    {of menu case}          {to indicate completion of command, call }
  HiliteMenu(0);                  { Menu Manager to unhighlight menu title }
                                  { (highlighted by MenuSelect)}
  END;  {of DoCommand}


BEGIN    { main program }
InitGraf(@thePort);            {initialize QuickDraw}
InitFonts;                     {initialize Font Manager}
FlushEvents(everyEvent,0);     {call OS Event Manager to discard any previous events}
InitWindows;                   {initialize Window Manager}
InitMenus;                     {initialize Menu Manager}
TEInit;                        {initialize TextEdit}
InitDialogs(NIL);              {initialize Dialog Manager}
InitCursor;                    {call QuickDraw to make cursor (pointer) an arrow}

SetUpMenus;                    {set up menus and menu bar}
WITH screenBits.bounds DO      {call QuickDraw to set dragging boundaries; ensure at }
  SetRect(dragRect,4,24,right-4,bottom-4); { least 4 by 4 pixels will remain visible}
doneFlag := FALSE;             {flag to detect when Quit command is chosen}

myWindow := GetNewWindow(windowID,@wRecord,POINTER(-1)); {put up application window}
SetPort(myWindow);             {call QuickDraw to set current grafPort to this window}
txRect := thePort^.portRect;   {rectangle for text in window; call QuickDraw to bring }
InsetRect(txRect,4,0);         { it in 4 pixels from left and right edges of window}
textH := TENew(txRect,txRect);         {call TextEdit to prepare for receiving text}

{ Main event loop }
REPEAT                                 {call Desk Manager to perform any periodic }
  SystemTask;                          { actions defined for desk accessories}
  TEIdle(textH);                       {call TextEdit to make vertical bar blink}
```

Figure 4. Example Program (continued)

```
  IF GetNextEvent(everyEvent,myEvent) {call Toolbox Event Manager to get the next }
    THEN                              { event that the application should handle}
      CASE myEvent.what OF            {case on event type}

      mouseDown:             {mouse button down:  call Window Manager to learn where}
        CASE FindWindow(myEvent.where,whichWindow) OF

        inMenuBar:           {menu bar:  call Menu Manager to learn which command; }
          DoCommand(MenuSelect(myEvent.where)); { then execute it}

        inSysWindow:         {desk accessory window: call Desk Manager to handle it}
          SystemClick(myEvent,whichWindow);

        inDrag:              {title bar:  call Window Manager to drag}
          DragWindow(whichWindow,myEvent.where,dragRect);

        inContent:                             {body of application window: }
          BEGIN                                { call Window Manager to check whether }
          IF whichWindow <> FrontWindow        { it's the active window and make it }
            THEN SelectWindow(whichWindow)     { active if not }
            ELSE
              BEGIN                            {it's already active:  call QuickDraw to }
              GlobalToLocal(myEvent.where);    { convert to window coordinates for }
                                               { TEClick, use Toolbox Utility BitAnd to }
              extended := BitAnd(myEvent.modifiers,shiftKey) <> 0;  { test for Shift }
              TEClick(myEvent.where,extended,textH);       { key down, and call TextEdit }
              END;                                         { to process the event}
          END;    {of inContent}

        END;    {of mouseDown}

      keyDown, autoKey:               {key pressed:  pass character to TextEdit}
          TEKey(CHR(BitAnd(myEvent.message,charCodeMask)),textH);

      activateEvt:
          BEGIN
          IF BitAnd(myEvent.modifiers,activeFlag) <> 0
            THEN                      {application window is becoming active: }
              BEGIN                   { call TextEdit to highlight selection }
              TEActivate(textH);      { or display blinking vertical bar, and call }
              DisableItem(myMenus[editM],undoCommand); { Menu Manager to disable }
              END                     { Undo (since application doesn't support Undo)}
            ELSE
              BEGIN                   {application window is becoming inactive: }
              TEDeactivate(textH);    { unhighlight selection or remove blinking }
              EnableItem(myMenus[editM],undoCommand);  { vertical bar, and enable }
              END;                    { Undo (since desk accessory may support it)}
          END;    {of activateEvt}

      updateEvt:                                 {window appearance needs updating}
        BEGIN
        BeginUpdate(WindowPtr(myEvent.message)); {call Window Manager to begin update}
        EraseRect(thePort^.portRect);            {call QuickDraw to erase text area}
        TEUpdate(thePort^.portRect,textH);       {call TextEdit to update the text}
        EndUpdate(WindowPtr(myEvent.message));   {call Window Manager to end update}
        END;    {of updateEvt}

      END;    {of event case}

UNTIL doneFlag;
END.
```

Figure 4. Example Program (continued)

WHERE TO GO FROM HERE

*** This section refers to "manuals" for the time being; when the individual manuals become chapters of Inside Macintosh, this will be changed to "chapters". It also refers to the "order" of the manuals; this means the order of the documentation when it's combined into a single manual. For a list of what's been distributed so far and how it will be ordered, see the cover page of this manual. Anything not listed there hasn't been distributed yet by Macintosh User Education, but programmer's notes or other preliminary documentation may be available. ***

This section contains important directions for every reader of Inside Macintosh. It will help you figure out which manuals to read next.

The Inside Macintosh documentation is ordered in such a way that you can follow it if you read through it sequentially. Forward references are given wherever necessary to any additional information that you'll need in order to understand what's being discussed. Special-purpose information that can possibly be skipped is indicated as such. Most likely you won't need to read everything in each manual and can even skip entire manuals.

You should begin by reading the following:

1. Macintosh User Interface Guidelines. All Macintosh applications should follow these guidelines to ensure that the end user is presented with a consistent, familiar interface.

2. Macintosh Memory Management: An Introduction.

3. Programming Macintosh Applications in Assembly Language, if you're using assembly language. Depending on the debugging tools available on the development system you're using, it may also be helpful or necessary for Pascal programmers to read this manual. You'll also have to read it if you're creating your own development system and want to know how to write interfaces to the routines.

4. The documentation of the parts of the Toolbox 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 overviews in this "road map" 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.

When you're ready to try something out, refer to the appropriate documentation for the development system you'll be using. *** (Lisa Workshop users, see Putting Together a Macintosh Application.) ***

APPENDIX: RESOURCE COMPILER INPUT FOR EXAMPLE PROGRAM

For Lisa Workshop users, this appendix shows the Resource Compiler input file used with the example program presented earlier. For more information on the format of the file, see Putting Together a Macintosh Application.

(note)

> This entire appendix is temporary; it will not be part of the final Inside Macintosh manual, because all the information in that manual will be independent of the development system being used. Authors of the documentation for a particular development system may choose to show how the resource file for Samp would be created on that system.

```
*  SampR -- Resource Compiler input file for Samp application
*          written by Macintosh User Education

Work/Samp.Rsrc

Type MENU
  ,128 (4)
* the apple symbol
  \14

  ,129 (4)
  File
    Quit

  ,130 (4)
  Edit
    (Undo
    (-
    Cut
    Copy
    Paste
    Clear

Type WIND
  ,128 (36)
  A Sample
  50 40 300 450
  Visible NoGoAway
  4
  0

Type SAMP = STR
  ,0
Samp Version 1.0 -- September 4, 1984

Type CODE
  Work/SampL,0
```

GLOSSARY

AppleBus Manager: An interface to a pair of RAM drivers that enable programs to send and receive information via an AppleBus network.

Binary-Decimal Conversion Package: A Macintosh package for converting integers to decimal strings and vice versa.

Control Manager: The part of the Toolbox that provides routines for creating and manipulating controls (such as buttons, check boxes, and scroll bars).

Desk Manager: The part of the Toolbox that supports the use of desk accessories from an application.

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

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

Dialog Manager: The part of the Toolbox that provides routines for implementing dialogs and alerts.

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

Disk Initialization Package: A Macintosh package for initializing and naming new disks; called by the Standard File Package.

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: The part of the Toolbox 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.

International Utilities Package: A Macintosh package that gives you access to country-dependent information such as the formats for numbers, currency, dates, and times.

main event loop: In a standard Macintosh application program, a loop that repeatedly calls the Toolbox Event Manager to get events and then responds to them as appropriate.

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

Menu Manager: The part of the Toolbox 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 I/O, memory management, and interrupt handling.

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 getting the date and time, finding out the user's preferred speaker volume and other preferences, and doing simple string comparison.

package: A set of routines and data types that's stored as a resource and brought into memory only when needed.

Package Manager: The part of the Toolbox that lets you access Macintosh RAM-based packages.

Printer Driver: The device driver for the currently installed printer.

Printing Manager: The routines and data types that enable applications to communicate with the Printer Driver to print on any variety of printer via the same interface.

QuickDraw: The part of the Toolbox 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 part of the Toolbox that reads and writes resources.

Scrap Manager: The part of the Toolbox 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.

Standard File Package: A Macintosh package for presenting the standard user interface when a file is to be saved or opened.

System Error Handler: The part of the Operating System that assumes control when a fatal error (such as running out of memory) occurs.

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

Toolbox: Same as User Interface Toolbox.

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

Toolbox Utilities: The part of the Toolbox that performs generally useful operations such as fixed-point arithmetic, string manipulation, and logical operations on bits.

User Interface Toolbox: The software in the Macintosh ROM that helps you implement the standard Macintosh user interface in your application.

vertical retrace interrupt: An interrupt generated 6Ø 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.

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

Window Manager: The part of the Toolbox that provides routines for creating and manipulating windows.

MACINTOSH USER EDUCATION

Macintosh User Interface Guidelines /INTF/USER

Modification History:			
	First Draft	Joanna Hoffman	3/17/82
	Rearranged and Revised	Chris Espinosa	5/11/82
	Total Redesign	Chris Espinosa	5/21/82
	Second Draft Prerelease	Chris Espinosa	7/11/82
	Second Draft	Chris Espinosa	10/11/82
	Third Draft	Andy Averill	7/31/84
	Fourth Draft	Andy Averill	11/30/84

ABSTRACT

The User Interface Guidelines describe the most basic common features of Macintosh applications. Unlike the rest of Inside Macintosh, these guidelines describe these features as seen by the user.

Since the last draft, this manual has been reorganized and mostly rewritten. Some new recommendations have been added, and some previous recommendations have been clarified or amplified.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Macintosh user interface, for the benefit of people who want to develop Macintosh applications. *** Eventually it will become part of the comprehensive Inside Macintosh manual. *** More details about many of these features can be found in the "About" sections of the other chapters of Inside Macintosh.

Unlike the rest of Inside Macintosh, this manual describes applications from the outside, not the inside. The terminology used is the terminology users are familiar with, which is not necessarily the same as that used elsewhere in Inside Macintosh.

The Macintosh user interface consists of those features that are generally applicable to a variety of applications. Not all of the features are found in every application. In fact, some features are hypothetical, and may not be found in any current applications.

The best time to familiarize yourself with the user interface is before beginning to design an application. Good application design on the Macintosh happens when a developer has absorbed the spirit as well as the details of the user interface.

Before reading this manual, you should have read Inside Macintosh: A Road Map and have some experience using one or more applications, preferably one each of a word processor, spreadsheet or data base, and graphics application. You should also have read Macintosh, the owner's guide, or at least be familiar with the terminology used in that manual.

INTRODUCTION

The Macintosh is designed to appeal to an audience of nonprogrammers, including people who have previously feared and distrusted computers. To achieve this goal, Macintosh applications should be easy to learn and to use. To help people feel more comfortable with the applications, the applications should build on skills that people already have, not force them to learn new ones. The user should feel in control of the computer, not the other way around. This is achieved in applications that embody three qualities: responsiveness, permissiveness, and consistency.

Responsiveness means that the user's actions tend to have direct results. The user should be able to accomplish what needs to be done spontaneously and intuitively, rather than having to think: "Let's see; to do C, first I have to do A and B and then...". For example, with pull-down menus, the user can choose the desired command directly and instantaneously. This is a typical Macintosh operation: The user moves the pointer to a location on the screen and presses the mouse button.

Permissiveness means that the application tends to allow the user to do anything reasonable. The user, not the system, decides what to do next. Also, error messages tend to come up infrequently. If the user is constantly subjected to a barrage of error messages, something is wrong somewhere.

The most important way in which an application is permissive is in avoiding modes. This idea is so important that it's dealt with in a separate section, "Avoiding Modes", below.

The third and most important principle is consistency. Since Macintosh users usually divide their time among several applications, they would be confused and irritated if they had to learn a completely new interface for each application. The main purpose of this manual is to describe the shared interface ideas of Macintosh applications, so that developers of new applications can gain leverage from the time spent developing and testing existing applications.

Fortunately, consistency is easier to achieve on the Macintosh than on many other computers. This is because many of the routines used to implement the user interface are supplied in the Macintosh Operating System and User Interface Toolbox. However, you should be aware that implementing the user interface guidelines in their full glory often requires writing additional code that isn't supplied.

Of course, you shouldn't feel that you're restricted to using existing features. The Macintosh is a growing system, and new ideas are essential. But the bread-and-butter features, the kind that every application has, should certainly work the same way so that the user can move easily back and forth between applications. The best rule to follow is that if your application has a feature that's described in these guidelines, you should implement the feature exactly as the guidelines describe it. It's better to do something completely different than to half-agree with the guidelines.

Illustrations of most of the features described in this manual can be found in various already-released applications. However, there is probably no one application that illustrates these guidelines in every particular. Although it's useful and important for you to get the feeling of the Macintosh user interface by looking at existing applications, the guidelines in this manual are the ultimate authority. Wherever an existing application disagrees with the guidelines, follow the guidelines.

Avoiding Modes

> "But, gentlemen, you overdo the mode."
>
> -- John Dryden, The Assignation, or Love in a Nunnery, 1672

A mode is a part of an application that the user has to formally enter and leave, and that restricts the operations that can be performed while it's in effect. Since people don't usually operate modally in real life, having to deal with modes in computer software reinforces the idea that computers are unnatural and unfriendly.

Modes are most confusing when you're in the wrong one. Unfortunately, this is the most common case. Being in a 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.

It's tempting to use modes in a Macintosh application, since most existing software leans on them heavily. If you yield to the temptation too frequently, however, users will consider spending time with your application a chore rather than a satisfying experience.

This is not to say that modes are never used in Macintosh applications. Sometimes a mode is the best way out of a particular problem. Most of these modes fall into one of the following categories:

- Long-term modes with a procedural basis, such as doing word processing as opposed to graphics editing. Each application program is a mode in this sense.

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

- Alert modes, where the user must rectify an unusual situation before proceeding. These modes should be kept to a minimum.

Other modes are acceptable if they meet one of the following requirements:

- They emulate a familiar real-life model that is itself modal, like picking up different-sized paintbrushes in a graphics editor. MacPaint and other palette-based applications are examples of this use of modes.

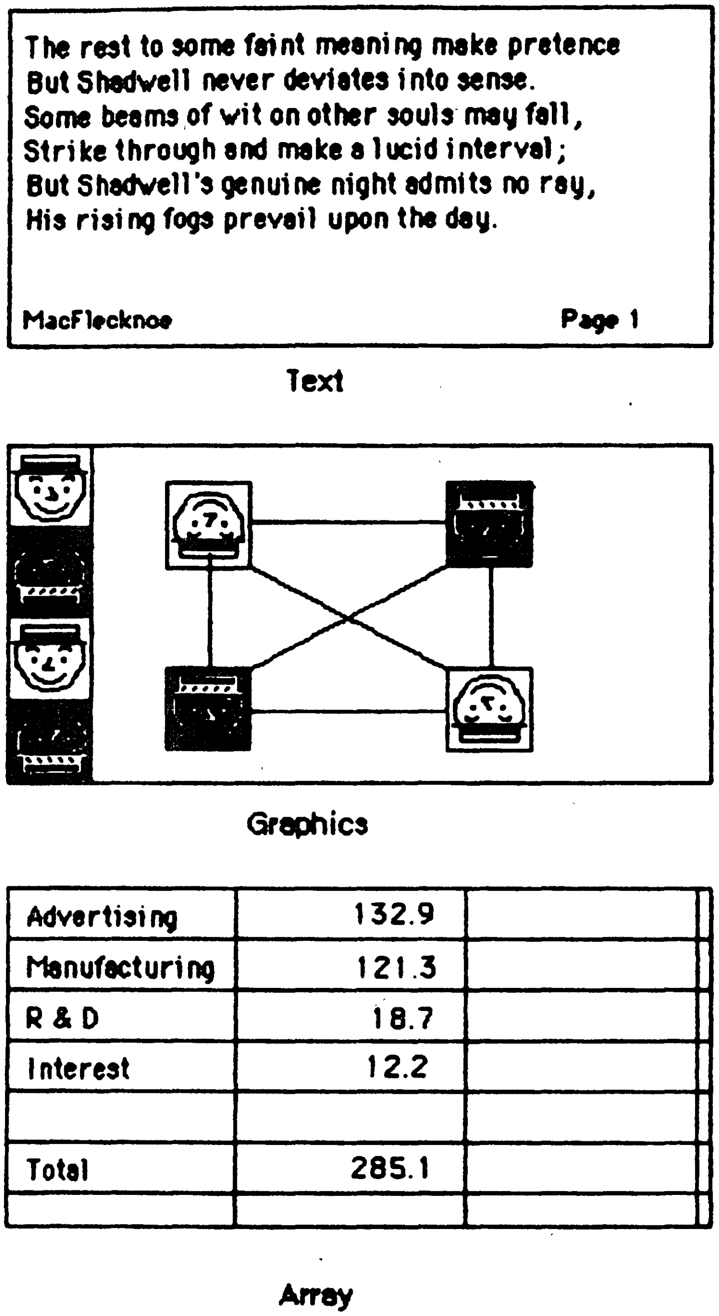
- They change only the attributes of something, and not its behavior, like the boldface and underline modes of text entry.

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

If an application uses modes, 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. It should also be very easy to get into or out of the mode (such as by clicking on a palette symbol).

TYPES OF APPLICATIONS

Everything on a Macintosh screen is displayed graphically; the Macintosh has no text mode. Nevertheless, it's useful to make a distinction among three types of objects that an application deals with: text, graphics, and arrays. Examples of each of these are shown in Figure 1.



The rest to some faint meaning make pretence
But Shadwell never deviates into sense.
Some beams of wit on other souls may fall,
Strike through and make a lucid interval;
But Shadwell's genuine night admits no ray,
His rising fogs prevail upon the day.

MacFlecknoe Page 1

Text

Graphics

Advertising	132.9	
Manufacturing	121.3	
R & D	18.7	
Interest	12.2	
Total	285.1	

Array

Figure 1. Ways of Structuring Information

Text can be arranged in a variety of ways on the screen. Some applications, such as word processors, might consist of nothing but text, while others, such as graphics-oriented applications, use text almost incidentally. It's useful to consider all the text appearing together in a particular context as a block of text. The size of the block can range from a single field, as in a dialog box, to the whole document, as in a word processor. Regardless of its size or arrangement, the application sees each block as a one-dimensional string of characters. Text is edited the same way regardless of where it appears.

Graphics are pictures, drawn either by the user or by the application. Graphics in a document tend to consist of discrete objects, which can be selected individually. Graphics are discussed further below, under "Using Graphics".

Arrays are one- or two-dimensional arrangements of fields. If the array is one-dimensional, it's called a form; if it's two-dimensional it's called a table. Each field, in turn, contains a collection of information, usually text, but conceivably graphics. A table can be readily identified on the screen, since it consists of rows and columns of fields (often called cells), separated by horizontal and vertical lines. A form is something you fill out, like a credit-card application. A form may not be as obvious to the user as a table, since the fields can be arranged in any appropriate way. Nevertheless, the application regards the fields as in a definite linear order.

Each of these three ways of presenting information retains its integrity, regardless of the context in which it appears. For example, a field in an array can contain text. When the user is manipulating the field as a whole, the field is treated as part of the array. When the user wants to change the contents of the field, the contents are edited in the same way as any other text.

Another case is text that appears in a graphics application. Depending on the circumstances, the text can be treated as text or as graphics. In MacDraw, for example, the way text is treated depends on which palette symbol is in effect. If the text symbol is in effect, text can be edited in the usual way, but cannot be moved around on the screen. If the selecting arrow is in effect, a block of text can be moved around, or even stretched or shrunk, but cannot be edited.

USING GRAPHICS

A key feature of the Macintosh is its high-resolution graphics screen. To use this screen to its best advantage, Macintosh applications use graphics copiously, even in places where other applications use text. As much as possible, all commands, features, and parameters of an application, and all the user's data, appear as graphic objects on the screen. Figure 2 shows some of the ways in which applications can use graphics to communicate with the user.

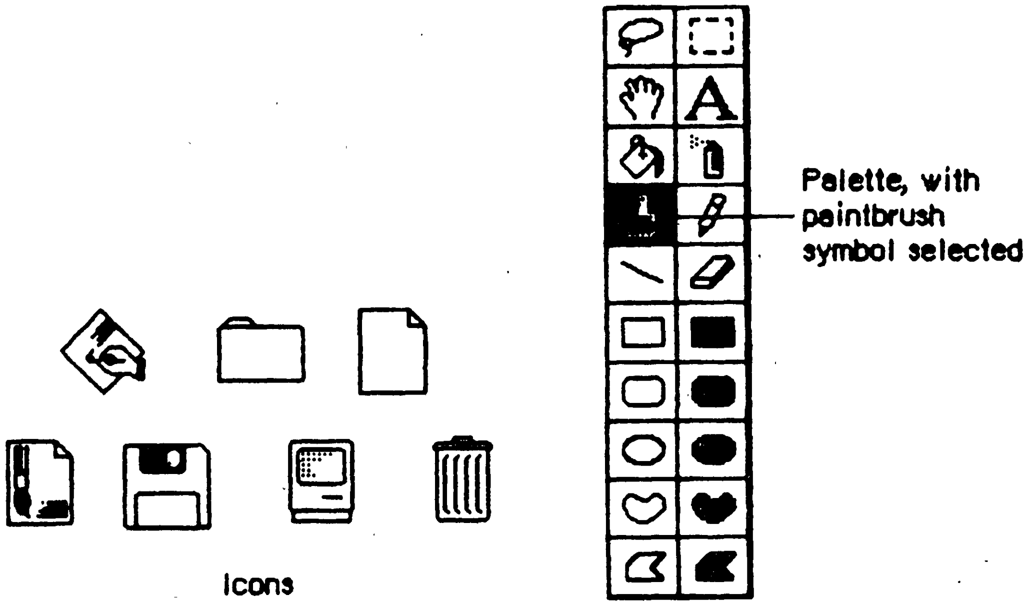


Figure 2. Objects on the Screen

Objects, whenever applicable, resemble the familiar material objects they resemble. Objects that act like pushbuttons "light up" when pressed; the Trash icon looks like a trash can.

Objects are designed to look good on the screen. Predefined graphics patterns 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.

Generally, when the user clicks on an object, it's highlighted to distinguish it from its peers. The most common way to show this highlighting is by inverting the object: reversing its black and white pixels. In some situations, other forms of highlighting, such as the knobs used in MacDraw, may be more appropriate. The important thing is that there should always be some sort of feedback, so that the user knows that the click had an effect.

One special aspect of the appearance of a document on the screen is visual fidelity. This principle is also known as "what you see is what you get". It primarily refers to printing: The version of a document shown on the screen should be as close as possible to its printed version, taking into account inevitable differences due to different media.

Icons

A fundamental object in Macintosh software is the icon, a small graphic object that is usually symbolic of an operation or of a larger entity such as a document.

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

Palettes

Some applications use palettes as a quick way for the user to change from one operation to another. A palette is a collection of small squares, each containing a symbol. A symbol can be an icon, a pattern, a character, or just a drawing, that stands for an operation. When the user clicks on one of the symbols, it's distinguished from the other symbols, such as by highlighting, and the previous symbol goes back to its normal state.

Typically, the symbol that's selected determines what operations the user can perform. Selecting a palette symbol puts the user into a mode. This use of modes can be justified because changing from one mode to another is almost instantaneous, and the use can always see at a glance which mode is in effect. Like all modal features, palettes should be used only when they're the most natural way to structure an application.

A palette can either be part of a window (as in MacDraw), or a separate window (as in MacPaint). Each system has its disadvantages. If the palette is part of the window, then parts of the palette might be concealed if the user makes the window smaller. On the other hand, if it's not part of the window, then it takes up extra space on the desktop. If an application supports multiple documents open at the same time, it might be better to put a separate palette in each window, so that a different palette symbol can be in effect in each document.

COMPONENTS OF THE MACINTOSH SYSTEM

This section explains the relationship among the principal large-scale components of the Macintosh system (from an external point of view).

The main vehicle for the interaction of the user and the system is the application. Only one application is active at a time. When an application is active, it's in control of all communications between the user and the system. The application's menus are in the menu bar, and the application is in charge of all windows as well as the desktop.

To the user, the main unit of information is the document. Each document is a unified collection of information--a single business letter or spreadsheet or chart. A complex application, such as a data base, might require several related documents. Some documents can be processed by more than one application, but each document has a principal application, which is usually the one that created it. The other applications that process the document are called secondary applications.

The only way the user can actually see the document (except by printing it) is through a window. The application puts one or more windows on the screen; each window shows a view of a document or of auxiliary information used in processing the document. The part of the screen underlying all the windows is called the desktop.

The user returns to the Finder to change applications. When the Finder is active, if the user double-clicks on either the application's icon or the icon of a document belonging to that application (or opens the document or application by choosing Open from the File menu), the application becomes active and displays the document window.

Internally, applications and documents are both kept in files. However, the user never sees files as such, so they don't really enter into the user interface.

THE KEYBOARD

The Macintosh keyboard is used primarily for entering text. Since commands are chosen from menus or by clicking somewhere on the screen, the keyboard is not needed for this function, although it can be used for alternative ways to enter commands.

The keys on the keyboard are arranged in familiar typewriter fashion. The U.S. keyboard is shown in Figure 3.

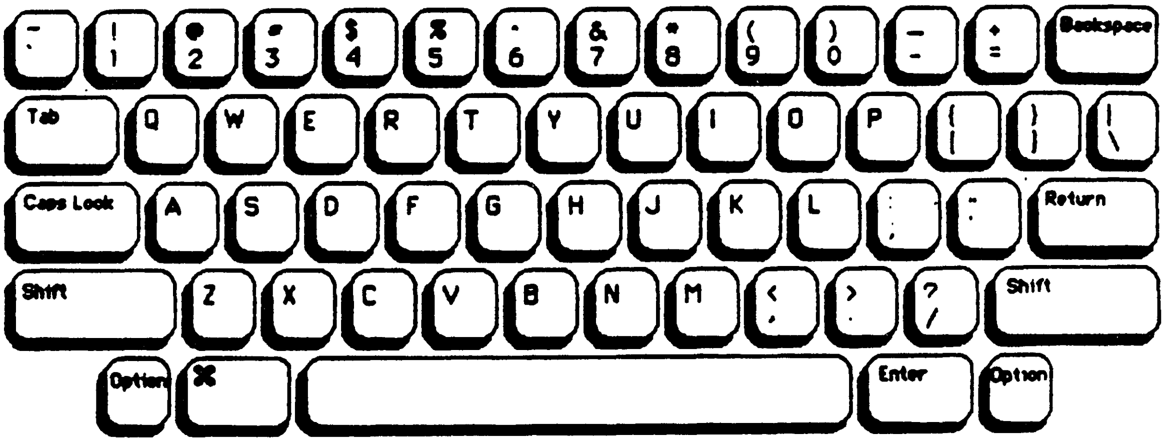


Figure 3. The Macintosh U.S. Keyboard

There are two kinds of keys: character keys and modifier keys. A character key sends characters to the computer; a modifier key alters the meaning of a character key if it's held down while the character key is pressed.

Character Keys

Character keys include keys for letters, numbers, and symbols, as well as the space bar. If the user presses one of these keys while entering text, the corresponding character is added to the text. Other keys, such as the Enter, Tab, Return, Backspace, and Clear keys, are also considered character keys. However, the result of pressing one of these keys depends on the application and the context.

The Enter key tells the application that the user is through entering information in a particular area of the document, such as a field in an array. Most applications add information to a document as soon as the user types or draws it. However, the application may need to wait until a whole collection of information is available before processing it. In this case, the user presses the Enter key to signal that the information is complete.

The Tab key is a signal to proceed: It signals movement to the next item in a sequence. Tab often implies an Enter operation before the Tab 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 can also imply an Enter operation before the Return operation.

(note)

> Return and Enter also dismiss dialog and alert boxes (see "Dialogs and Alerts").

Backspace is used to delete text or graphics. The exact use of Backspace in text is described in the section on text editing.

The Clear key on the keypad has the same effect as the Clear command in the Edit menu; that is, it removes the selection from the document without putting it on the Clipboard. This is also explained in the section on text editing. Because the keypad is optional equipment, no application should ever require use of the Clear key or any other key on the pad.

Modifier Keys: Shift, Caps Lock, Option, and Command

There are six keys on the keyboard that change the interpretation of keystrokes: two labeled Shift, two labeled Option, one labeled Caps Lock, and one labeled with the "freeway interchange" symbol, which is usually called the Command key. These keys change the interpretation of keystrokes, and sometimes mouse actions. When one of these keys is held down, the effect of the other keys (or the mouse button) may change.

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. The Shift key is also used in conjunction with the mouse for extending a selection; see "Selecting". Option gives an alternate character set interpretation, including international characters, special symbols, and so on. 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, but 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.

Pressing a character key while holding down the Command key usually tells the application to interpret the key as a command, not as a character (see "Commands").

Typeahead and Auto-Repeat

If the user types when the Macintosh is unable to process the keystrokes immediately, or types more quickly than the Macintosh can handle, the extra keystrokes are queued, to be processed later. This queuing is called typeahead. There's a limit to the number of keystrokes that can be queued, but the limit is usually not a problem unless the user types while the application is performing a lengthy operation.

When the user holds down a character key for a certain amount of time, it starts repeating automatically. The delays and the rates of repetition are global preferences that the user can set through the Control Panel desk accessory. An application can tell whether a series of n keystrokes was generated by auto-repeat or by pressing the same key n times. It can choose to disregard keystrokes generated by auto-repeat; this is usually a good idea for menu commands chosen with the Command key.

Holding down a modifier key has the same effect as pressing it once. However, if the user holds down a modifier key and a character key at the same time, the effect is the same as if the user held down the modifier key while pressing the character key repeatedly.

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

Versions of the Keyboard

There are two physical versions of the keyboard: U.S. and European. The European version has one more key than the U.S. version. The standard layout on the European version is designed to conform to the ISO (Internation Standards Organization) standard; the U.S. key layout mimics that of common American office typewriters. European keyboards have different labels on the keys in different countries, but the overall layout is the same.

The Numeric Keypad

An optional numeric keypad can be hooked up between the main unit and the standard keyboard; see Figure 4.

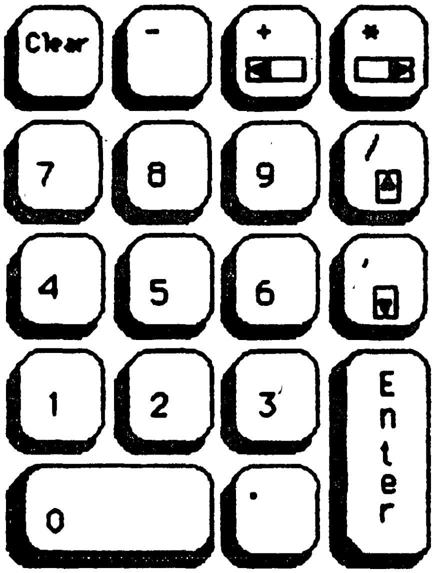


Figure 4. Numeric Keypad

The keypad contains 18 keys, some of which duplicate keys on the main keyboard, and some of which are unique to the keypad. The application can tell whether the keystrokes have come from the main keyboard or the numeric keypad.

The character keys on the keypad are labeled with the digits Ø through 9, a decimal point, the four standard arithmetic operators for addition, subtraction, multiplication, and division, and a comma. The keypad also contains the Enter and Clear keys; it has no modifier keys.

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

Four keys on the numeric keypad are labeled with "field-motion" symbols: small rectangles with arrows pointing in various directions.

Some applications may use these keys to select objects in the direction indicated by the key; the most likely use for this feature is in tables. When a key is used this way, the user must use the Shift key to obtain the four characters (+ * / ,) normally available on those keys.

Since the numeric keypad is optional equipment, no application should require it or any keys available on it in order to perform standard functions. Specifically, since the Clear key is not available on the main keyboard, a Clear function may be implemented with this key only as the equivalent of the Clear command in the Edit menu.

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's a button on the top of the mouse. The user holds the mouse and rolls it on a flat, smooth surface. 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 "hot spot" of the pointer over an object on the screen and presses and releases the mouse button. The hot spot should be intuitive, like the point of an arrow or the center of a crossbar.

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 holding down the mouse button without moving the mouse
- dragging: positioning the pointer with the mouse, holding down the mouse button, moving the mouse to a new position, and releasing the button

The system provides "mouse-ahead"; that is, any mouse actions the user performs when the application isn't ready to process them are saved in a buffer and can be processed at the application's convenience. Alternatively, the application can choose to ignore saved-up mouse actions, but should do so only to protect the user from possibly damaging consequences.

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

For certain kinds of objects, pressing on the object has the same effect as clicking it repeatedly. For example, clicking a scroll arrow causes a document to scroll one line; pressing on a scroll arrow causes the document to scroll repeatedly until the mouse button is released or the end of the document is reached.

Dragging can have different effects, depending on what's under the pointer when the mouse button is pressed. The uses of dragging include choosing a menu item, selecting a range of objects, moving an object from one place to another, and shrinking or expanding an object.

Some objects, especially graphic objects, can be moved by dragging. In this case, the application attaches a dotted outline of the object to the pointer and redraws the outline continually as the user moves the pointer. When the user releases the mouse button, the application redraws the complete object at the new location.

An object being moved can be restricted to certain boundaries, such as the edges of a window frame. If the user moves the pointer outside of the boundaries, the application stops drawing the dotted outline of the object. If the user releases the mouse button while the pointer is outside of the boundaries, the object isn't moved. If, on the other hand, the user moves the pointer back within the boundaries again before releasing the mouse button, the outline is drawn again.

In general, moving the mouse changes nothing except the location, and possibly the shape, of the pointer. Pressing the mouse button indicates the intention to do something, and releasing the button completes the action. Pressing by itself should have no effect except in well-defined areas, such as scroll arrows, where it has the same effect as repeated clicking.

Multiple-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 a short amount of time (as set by the user in the Control Panel), and if the locations of the two clicks are reasonably close together, the two clicks constitute a double-click. Its most common use is as a faster or easier way to perform an action that can also be performed in another way. For example, clicking twice on an icon is a faster way to open it than choosing Open; clicking twice on a word to select it is faster than dragging through it.

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

Triple-clicking is also possible; it should similarly represent an extension of a double-click.

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.

- The result of any mouse action depends 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 actions possible in each area.

- Where an application uses modes for different functions, the pointer can be a different shape in each mode. For example, in MacPaint, the pointer shape always reflects the active palette symbol.

Figure 5 shows some examples of pointers and their effect. An application can design additional pointers for other contexts.

Pointer	Used for
↖	Scroll bar and other controls, size box title bar, menu bar, desktop, and so on
I	Selecting text
+	Drawing, shrinking, or stretching graphic objects
✚	Selecting fields in an array
⌚	Showing that a lengthy operation is in progress

Figure 5. Pointers

SELECTING

The user selects an object to distinguish it from other objects, just before performing an operation on it. Selecting the object of an operation before identifying the operation is a fundamental characteristic of the Macintosh system.

Selecting an object has no effect on the contents of a document. Making a selection shouldn't commit the user to anything; the user is never penalized for making an incorrect selection. The user fixes an incorrect selection by making the correct selection.

Although there is a variety of ways to select objects, they fall into easily recognizable groups. Users get used to doing specific things to select objects, and applications that use these methods are therefore easier to learn. Some of these methods apply to every type of application, and some only to particular types of applications.

This section discusses first the general methods, and then the specific methods that apply to text applications, graphics applications, and arrays. Figure 6 shows a comparison of some of the general methods.

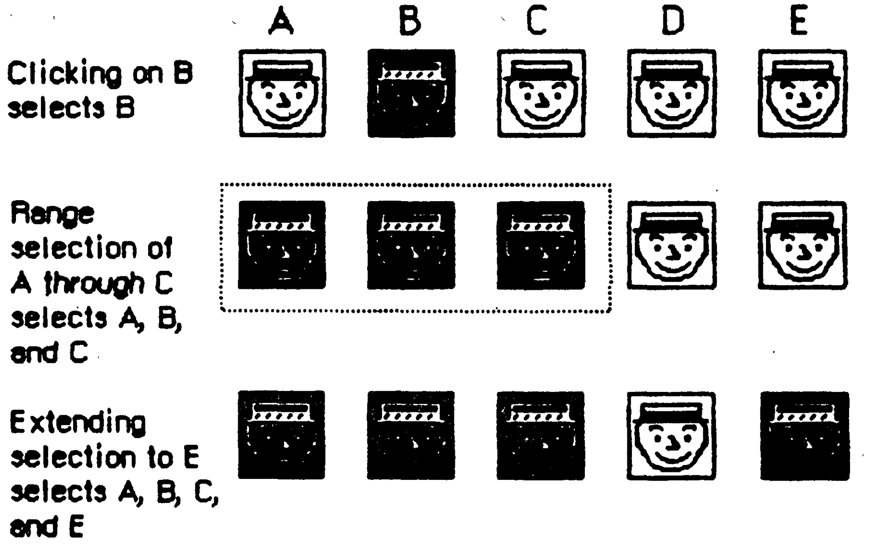

A
B
C
D
E
Clicking on B selects B
Range selection of A through C selects A, B, and C
Extending selection to E selects A, B, C, and E

Figure 6. Selection Methods

Selection by Clicking

The most straightforward method of selecting an object is by clicking on it once. Most things that can be selected in Macintosh applications can be selected this way.

Some applications support selection by double-clicking and triple-clicking. As always with multiple clicks, the second click extends the effect of the first click, and the third click extends the effect of the second click. In the case of selection, this means that the second click selects the same sort of thing as the first click, only more of them. The same holds true for the third click.

For example, in text, the first click selects an insertion point, whereas the second click selects a whole word. The third click might select a whole block or paragraph of text. In graphics, the first click selects a single object, and double- and triple-clicks might select increasingly larger groups of objects.

Range Selection

The user selects a range of objects by dragging through them. Although the exact meaning of the selection depends on the type of application, the procedure is always the same:

1. The user positions the pointer at one corner of the range and presses the mouse button. This position is called the anchor point of the range.

2. The user moves the pointer in any direction. As the pointer is moved, visual feedback keeps the user informed of the objects that would be selected if the mouse button were released. For text and arrays, the selected area is continually highlighted. For graphics, a dotted rectangle expands or contracts to show the range that will be selected.

3. When the feedback shows the desired range, the user releases the mouse button. The point at which the button is released is called the endpoint of the range.

Extending a Selection

A user can change the extent of an existing selection by holding down the Shift key and clicking the mouse button. Exactly what happens next depends on the context.

In text or an array, the result of a Shift-click is always a range. The position where the button is clicked becomes the new endpoint or anchor point of the range; the selection can be extended in any direction. If the user clicks within the current range, the new range will be smaller than the old range.

In graphics, a selection is extended by adding objects to it; the added objects do not have to be adjacent to the objects already selected. The user can add either an individual object or a range of objects to the selection by holding down the Shift key before making the additional selection. If the user holds down the Shift key and selects one or more objects that are already highlighted, the objects are deselected.

Extended selections can be made across the panes of a split window. (See "Splitting Windows".)

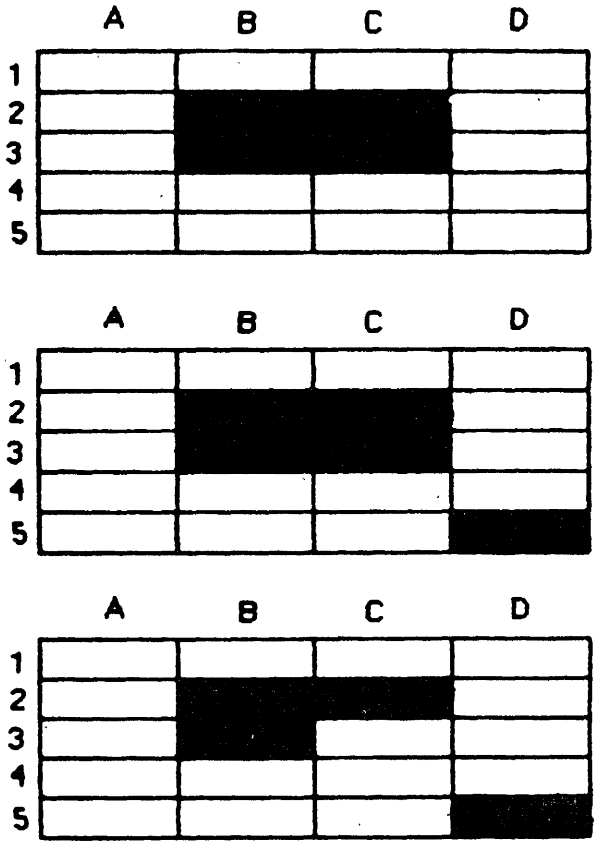
Making a Discontinuous Selection

In graphics applications, objects aren't usually considered to be in any particular sequence. Therefore, the user can use Shift-click to extend a selection by a single object, even if that object is nowhere near the current selection. When this happens, the objects between the current selection and the new object are not automatically included in the selection. This kind of selection is called a discontinuous selection. In the case of graphics, all selections are discontinuous selections.

This is not the case with arrays and text, however. In these two kinds of applications, an extended selection made by a Shift-click always includes everything between the old selection and the new endpoint. To provide the possibility of a discontinuous selection in these applications, Command-click is included in the user interface.

To make a discontinuous selection in a text or array application, the user selects the first piece in the normal way, then holds down the Command key before selecting the remaining pieces. Each piece is selected in the same way as if it were the whole selection, but because the Command key is held down, the new pieces are added to the existing selection instead of supplanting it.

If one of the pieces selected is already within an existing part of the selection, then instead of being added to the selection it's removed from the selection. Figure 7 shows a sequence in which several pieces are selected and deselected.



Cells B2, B3, C2, and C3 are selected

The user holds down the Command key and clicks in D5

The user holds down the Command key and clicks in C3

Figure 7. Discontinuous Selection

Not all applications support discontinuous selections, and those that do might restrict the operations that a user can perform on them. For example, a word processor might allow the user to choose a font after making a discontinuous selection, but not to choose Cut or Paste.

Selecting Text

Text is used in most applications; it's selected and edited in a consistent way, regardless of where it appears.

A block of text is a string of characters. A text selection is a substring of this string, which can have any length from zero characters to the whole block. Each of the text selection methods selects a different kind of substring. Figure 8 shows different kinds of text selections.

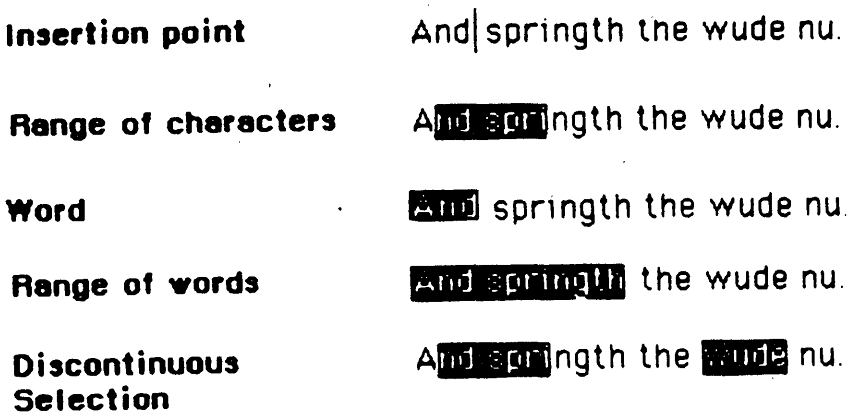

Insertion point
And|springth the wude nu.
Range of characters
And springth the wude nu.
Word
And springth the wude nu.
Range of words
And springth the wude nu.
Discontinuous Selection
And springth the wude nu.

Figure 8. Text Selections

Insertion Point

The insertion point is a zero-length text selection. The user establishes the location of the insertion point by clicking between two characters. The insertion point then appears at the nearest character boundary. If the user clicks to the right of the last character on a line, the insertion point appears immediately after the last character. The converse is true if the user clicks to the left of the first character in the line.

The insertion point shows where text will be inserted when the user begins typing, or where the contents of the Clipboard will be pasted. After each character is typed, the insertion point is relocated to the right of the insertion.

If, between the mouse-down and the mouse-up, the user moves the pointer more than about half the width of a character, the selection is a range selection rather than an insertion point.

Selecting Words

The user selects a whole word by double-clicking somewhere within that word. If the user begins a double-click sequence, but then drags the mouse between the mouse-down and the mouse-up of the second click, the selection becomes a range of words rather than a single word. As the pointer moves, the application highlights or unhighlights a whole word at a time.

A word, or range of words, can also be selected in the same way as any other range; whether this type of selection is treated as a range of characters or as a range of words depends on the operation. For example, in MacWrite, a range of individual characters that happens to coincide with a range of words is treated like characters for purposes of extending a selection, but is treated like words for purposes of intelligent cut and paste.

A word is defined as any continuous substring that contains only the following characters:

- a letter (including letters with diacritical marks)
- a digit
- a nonbreaking space (Option-space)
- a dollar sign, cent sign, English pound symbol, or yen symbol
- a percent sign
- a comma between digits
- a period before a digit
- an apostrophe between letters or digits
- a hyphen, but not a minus sign (Option-hyphen) or a dash (Option-Shift-hyphen)

This is the definition in the United States and Canada; in other countries, it would have to be changed to reflect local formats for numbers, dates, and currency.

If the user double-clicks over any character not on the list above, only that character is selected.

Examples of words:

```
        $123,456.78
        shouldn't
        3 1/2 [with a nonbreaking space]
        .5%
```

Examples of nonwords:

```
       7/10/6
        blue cheese [with a breaking space]
        "Yoicks!"  [the quotation
                    marks and exclamation point aren't part of the word]
```

Selecting a Range of Text

The user selects a range of text by dragging through the range. A range is either a range of words or a range of individual characters, as described under "Selecting Words", above.

If the user extends the range, the way the range is extended depends on what kind of range it is. If it's a range of individual characters, it can be extended one character at a time. If it's a range of words (including a single word), it's extended only by whole words.

Graphics Selections

There are several different ways to select graphic objects and to show selection feedback in existing Macintosh applications. MacDraw, MacPaint, and the Finder all illustrate different possibilities. This section describes the MacDraw paradigm, which is the most extensible to other kinds of applications.

A MacDraw document is a collection of individual graphic objects. To select one of these objects, the user clicks once on the object, which is then shown with knobs. (The knobs are used to stretch or shrink the object, and won't be discussed in this manual.) Figure 9 shows some examples of selection in MacDraw.

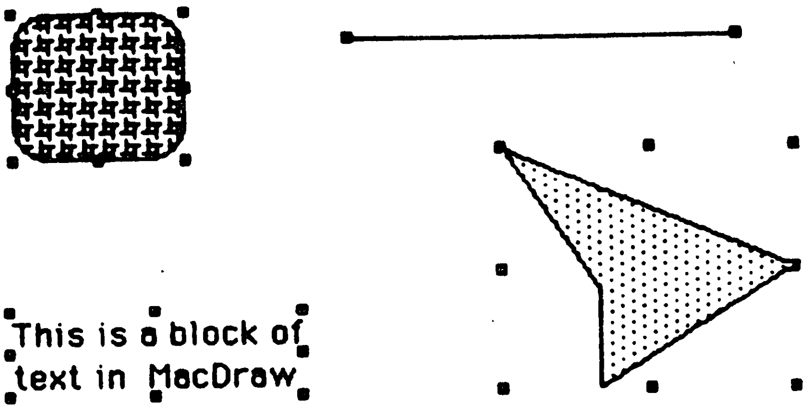


Figure 9. Graphics Selections in MacDraw

To select more than one object, the user can select either a range or a multiple selection. A range selection includes every object completely contained within the dotted rectangle that encloses the range, while an extended selection includes only those objects explicitly selected.

Selections in Arrays

As described above, under "Types of Applications", an array is a one- or two-dimensional arrangement of fields. If the array is one-dimensional, it's called a form; if it's two-dimensional, it's called a table. The user can select one or more fields, or part of the contents of a field.

To select a single field, the user clicks in the field. The user can also implicitly select a field by moving into it with the Tab or Return key.

The Tab key cycles through the fields in an order determined by the application. From each field, the Tab key selects the "next" field. Typically, the sequence of fields is first from left to right, and then from top to bottom. When the last field in a form is selected, pressing the Tab key selects the first field in the form. In a form, an application might prefer to select the fields in logical, rather than physical, order.

The Return key selects the first field in the next row. If the idea of rows doesn't make sense in a particular context, then the Return key should have the same effect as the Tab key.

Tables are more likely than forms to support range selections and extended selections. A table can also support selection of rows and columns. The most convenient way for the user to select a column is to click in the column header. To select more than one column, the user drags through several column headers. The same applies to rows.

To select part of the contents of a field, the user must first select the field. The user then clicks again to select the desired part of the field. Since the contents of a field are either text or graphics, this type of selection follows the rules outlined above. Figure 1Ø shows some selections in an array.

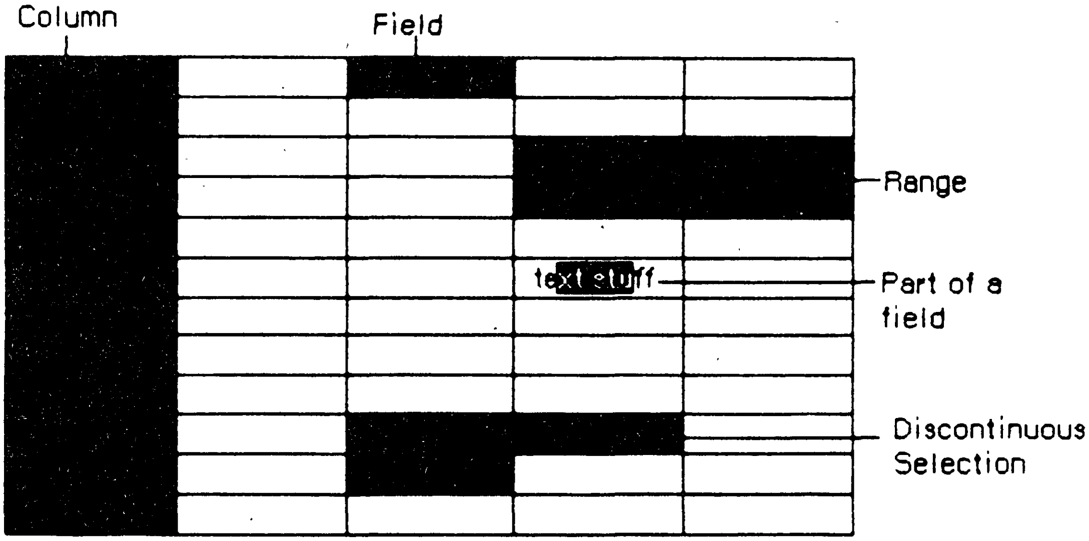


Figure 1Ø. Array Selections

WINDOWS

Windows are the rectangles on the desktop that display information. The most commmon types of windows are document windows, desk accessories, dialog boxes, and alert boxes. (Dialog and alert boxes are discussed under "Dialogs and Alerts".) Some of the features described in this section are applicable only to document windows. Figure 11 shows a typical active window and some of its components.

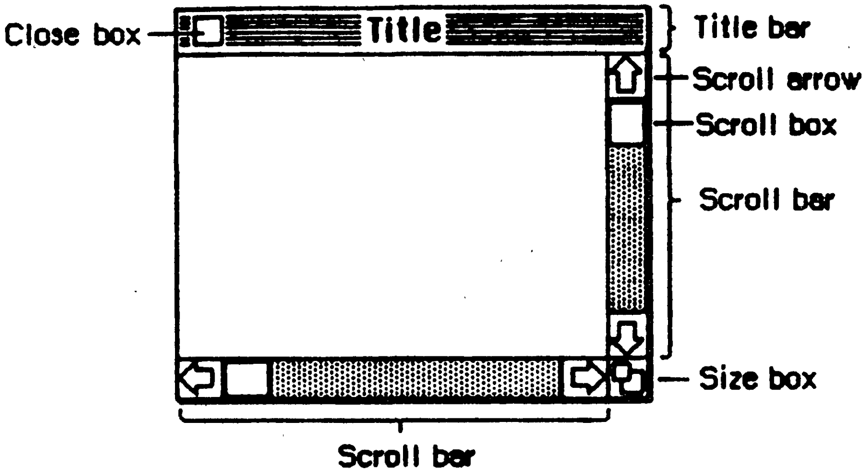


Figure 11. An Active Window

Multiple Windows

Some applications may be able to keep several windows on the desktop at the same time. Each window is in a different plane. Windows can be moved around on the Macintosh's desktop much like pieces of paper can be moved around on a real desktop. Each window can overlap those behind it, and can be overlapped by those in front of it. Even when windows don't overlap, they retain their front-to-back ordering.

Different windows can represent:

- different parts of the same document, such as the beginning and end of a long report

- different interpretations of the same document, such as the tabular and chart forms of a set of numerical data

- related parts of a logical whole, like the listing, execution, and debugging of a program

- separate documents being viewed or edited simultaneously

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

The advantage of multiple windows is that the user can isolate unrelated chunks of information from each other. The disadvantage is that the desktop can become cluttered, especially if some of the windows can't be moved. Figure 12 shows multiple windows.

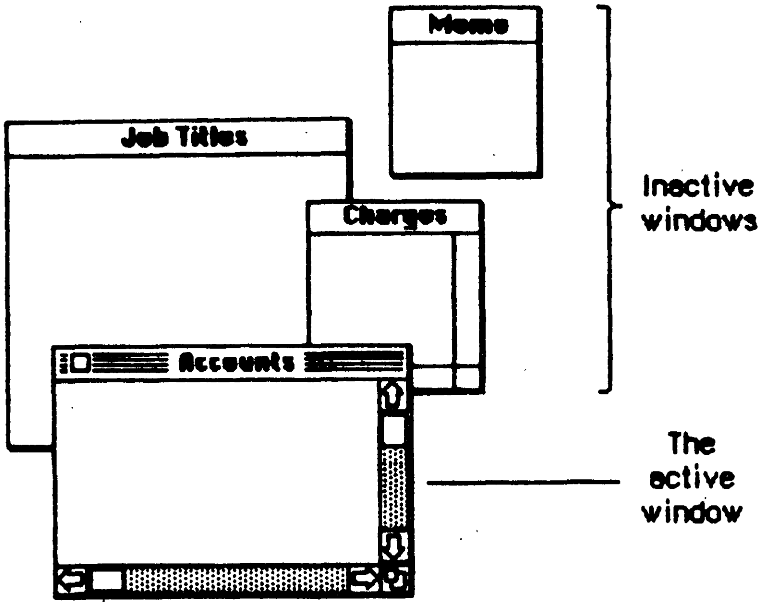


Figure 12. Multiple Windows

Opening and Closing Windows

Windows come up onto the screen in different ways as appropriate to the purpose of the window. The application controls at least the initial size and placement of its windows.

Most windows have a close box that, when clicked, makes the window go away. The application in control of the window determines what's done with the window visually and logically when the close box is clicked. Visually, the window can either shrink to a smaller object such as an icon, or leave no trace behind when it closes. Logically, the information in the window is either retained and then restored when the window is reopened (which is the usual case), or else the window is reinitialized each time it's opened. When a document is closed, the user is given the choice whether to save any changes made to the document since the last time it was saved.

If an application doesn't support closing a window with a close box, it should not include a close box on the window.

The Active Window

Of all the windows that are open on the desktop, the user can work in only one window at a time. This window is called the active window. All other open windows are inactive. To make a window active, the user clicks in it. Making a window active has two immediate consequences:

- The window's title bar is highlighted, the scroll bars and size box are shown, and any controls inside the window become active. If the window is being reactivated, the selection that was in effect when it was deactivated is rehighlighted.

- The window is moved to the frontmost plane, so that it's shown in front of any windows that it overlaps.

Clicking in a window does nothing except activate it. To make a selection in the window, the user must click again. When the user clicks in a window that has been deactivated, the window should be reinstated just the way it was when it was deactivated, with the same position of the scroll box, and the same selection highlighted.

When a window becomes inactive, all the visual changes that took place when it was activated are reversed. The title bar becomes unhighlighted, the scroll bars and size box aren't shown, any controls inside the window are dimmed, and no selection is shown in the window.

Moving a Window

Each application initially places windows on the screen wherever it wants them. The user can move a window--to make more room on the desktop or to uncover a window it's overlapping--by dragging it by its title bar. As soon as the user presses in the title bar, that window becomes the active window. A dotted 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. Moving a window doesn't affect the appearance of the document within the window.

If the user holds down the Command key while moving the window, the window isn't made active; it moves in the same plane.

The application should ensure that a window can never be moved completely off the screen.

Changing the Size of a Window

If a window has a size box in its bottom 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.

Dragging the size box attaches a dotted outline of the window to the pointer. 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 shape of the dotted outline.

Moving windows and sizing them go hand in hand. If a window can be moved, but not sized, then the user ends up constantly moving windows on and off the screen. The reason for this is that if the user moves the window off the right or bottom edge of the screen, the scroll bars are the first thing to disappear. To scroll the window, the user must move the window back onto the screen again. If, on the other hand, the window can be resized, then the user can change its size instead of moving it off the screen, and will still be able to scroll.

Sizing a window doesn't change the position of the top left corner of the window over the document or the appearance of the part of the view that's still showing; it changes only how much of the view is visible inside the window. One exception to this rule is a command such as Reduce to Fit in MacDraw, which changes the scaling of the view to fit the size of the window. If, after choosing this command, the user resizes the window, the application changes the scaling of the view.

The application can define a minimum window size. Any attempt to shrink the window below this size is ignored.

Scroll Bars

Scroll bars are used to change which part of a document view is shown in a window. Only the active window can be scrolled.

A scroll bar (see Figure 11 above) 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 scroll box) represents the location of 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 scroll bar moves correspondingly. If the document is no larger than the window, the scroll bars are inactive (the scrolling apparatus isn't shown in them). If the document window is inactive, the scroll bars aren't shown at all.

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

Clicking a scroll arrow moves the document in the opposite direction from 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 scroll box 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 application: one line for a word processor, one row or column for a spreadsheet, and so on. Within a document, units should always be the same size, for smooth scrolling. Pressing the scroll arrow causes continuous movement in its direction.

Clicking the mouse anywhere in the gray area of the scroll bar advances the document by windowfuls. The scroll box, and the document view, move toward the place where the user clicked. Clicking below the scroll box, for example, brings the user the next windowful towards the bottom of the document. Pressing in the gray area keeps windowfuls flipping by until the user releases the button, or until the location of the scroll box catches up to the location of the pointer. Each windowful is the height or width of the window, minus one unit overlap (where a unit is the distance the view scrolls when the scroll arrow is clicked once).

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

If the user starts dragging the scroll box, and then moves the pointer a certain distance outside the scroll bar, the scroll box detaches itself from the pointer and stops following it; if the user releases the mouse button, the scroll box 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 scroll bar, the scroll box reattaches itself to the pointer and can be dragged as usual.

If a document has a fixed size, and the user scrolls to the right or bottom edge of the document, the application displays a small amount of gray background (the same pattern as the desktop) between the edge of the document and the window frame.

Automatic Scrolling

There are several instances when the application, rather than the user, scrolls the document. These instances involve some potentially sticky problems about how to position the document within the window after scrolling.

The first case is when the user moves the pointer out of the window while selecting by dragging. The window keeps up with the selection by scrolling automatically in the direction the pointer has been moved. The rate of scrolling is the same as if the user were pressing on the corresponding scroll arrow or arrows.

The second case is when the selection isn't currently showing in the window, and the user performs an operation on it. When this happens, it's usually because the user has scrolled the document after making a selection. In this case, the application scrolls the window so that the selection is showing before performing the operation.

The third case is when the application performs an operation whose side effect is to make a new selection. An example is a search operation, after which the object of the search is selected. If this object isn't showing in the window, the application must scroll the window so as to show it.

The second and third cases present the same problem: Where should the selection be positioned within the window after scrolling? The primary rule is that the application should avoid unnecessary scrolling; users prefer to retain control over the positioning of a document. The following guidelines should be helpful:

- If part of the new selection is already showing in the window, don't scroll at all. An exception to this rule is when the part of the selection that isn't showing is more important than the part that's showing.

- If scrolling in one orientation (horizontal or vertical) is sufficient to reveal the selection, don't scroll in both orientations.

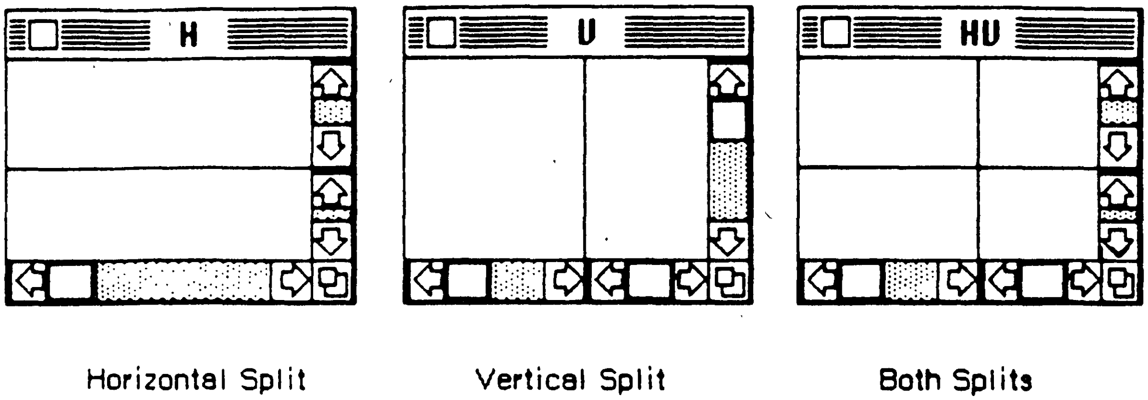
- If the selection is smaller than the window, position the selection so that some of its context is showing on each side. It's better to put the selection somewhere near the middle of the window than right up against the corner.

- Even if the selection is too large to show in the window, it might be preferable to show some context rather than to try to fit as much as possible of the selection in the window.

Splitting a Window

Sometimes it's desirable to be able to see disjoint parts of a document simultaneously. Applications that accommodate such a capability allow the window to be split into independently scrollable panes.

Applications that support splitting a window into panes place split bars at the top of the vertical scroll bar and to the left of the horizontal one. Pressing a split bar attaches it to the pointer. Dragging the split bar positions it anywhere along the scroll bar; releasing the mouse button moves the split bar to a new position, splits the window at that location, and divides the appropriate scroll bar (horizontal or vertical) into separate scroll bars for each pane. Figure 13 shows the ways a window can be split.



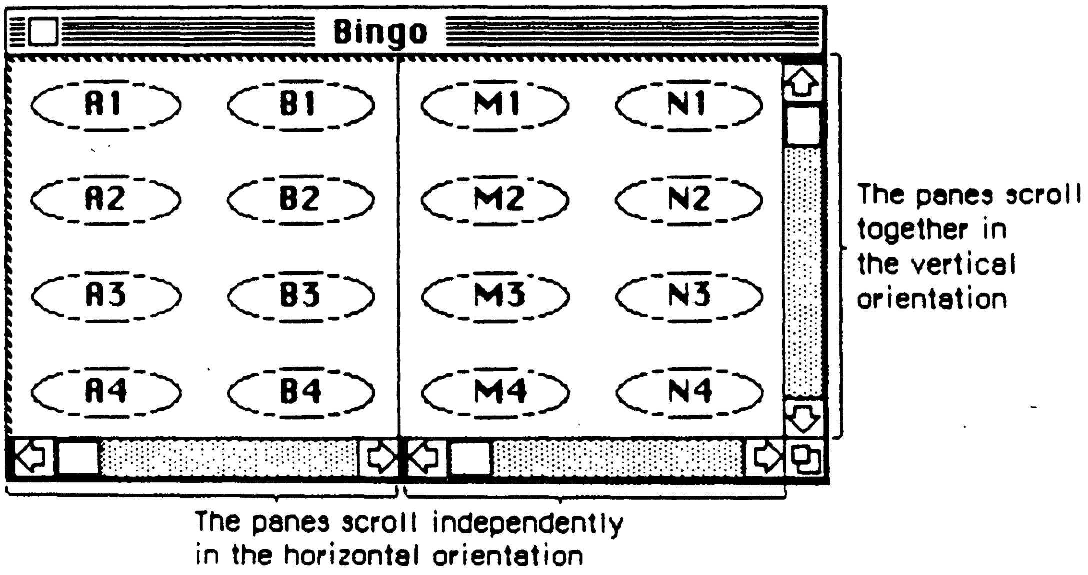
Horizontal Split

Vertical Split

Both Splits

Figure 13. Types of Split Windows

After a split, the document appears the same, except for the split line lying across it. But there are now separate scroll bars for each pane. The panes are still scrolled together in the orientation of the split, but can be scrolled independently in the other orientation. For example, if the split is horizontal, then horizontal scrolling (using the scroll bar along the bottom of the window), is still synchronous. Vertical scrolling is controlled separately for each pane, using the two scroll bars along the right of the window. This is shown in Figure 14.



The panes scroll together in the vertical orientation

The panes scroll independently in the horizontal orientation

Figure 14. Scrolling a Split Window

To remove a split, the user drags the split bar to the bottom or the right of the window.

The number of views in a document doesn't alter the number of selections per document: that is, one. The active selection appears highlighted in all views that show it. If the application has to scroll automatically to show the selection, the pane that should be scrolled is the last one that the user clicked in. If the selection is already showing in one of the panes, no automatic scrolling takes place.

Panels

If a document window is more or less permanently divided into different regions, each of which has different content, these regions are called panels. Unlike panes, which show different parts of the same document but are functionally identical, panels are functionally different from each other but might show different interpretations of the same part of the document. For example, one panel might show a graphic version of the document while another panel shows a textual version.

Panels can behave much like subwindows; they can have scroll bars, and can even be split into more than one pane. An example of a panel with scroll bars is the list of files in the Open dialog box.

Whether to use panels instead of separate windows is up to the application. Multiple panels in the same window are more compact than separate windows, but they have to be moved, opened, and closed as a unit.

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.

Macintosh's pull-down menus have the advantage that they're not visible until the user wants to see them; at the same time they're easy for the user to see and choose items from.

Most commands either do something, in which case they're verbs or verb phrases, or else they specify an attribute of an object, in which case they're adjectives. They usually apply to the current selection, although some commands apply to the whole document or window.

When you're designing your application, don't assume that everything has to be done through menu commands. Sometimes it's more appropriate for an operation to take place as a result of direct user manipulation of a graphic object on the screen, such as a control or icon. Alternatively, a single command can execute complicated instructions if it brings up a dialog box for the user to fill in.

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 associated with the current application. Each application has its own menu bar. The names of the menus do not change, except when the user calls for a desk accessory that uses different menus.

Only menu titles appear in the menu bar. If all of the commands in a menu are currently disabled (that is, the user can't choose them), the menu title should be dimmed (in gray type). The user can pull down the menu to see the commands, but can't choose any of them.

Choosing a Menu Command

To choose a command, the user positions the pointer over the menu title and presses the mouse button. The application highlights the title and displays the menu, as shown in Figure 15.

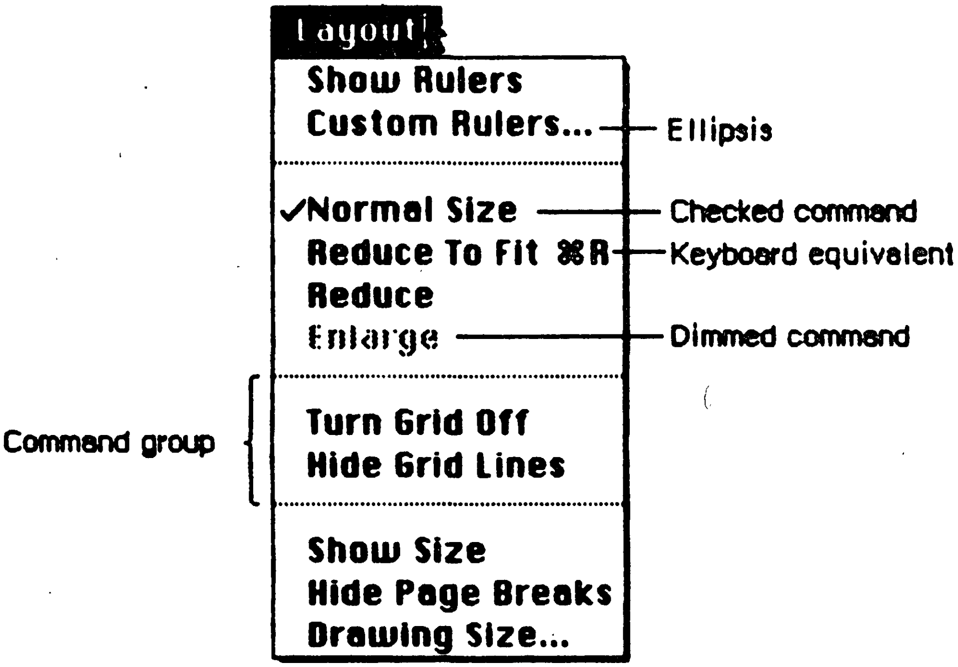


Figure 15. Menu

While holding down the mouse button, the user moves the pointer down the menu. As the pointer moves to each command, the command is highlighted. The command that's highlighted when the user releases the mouse button is chosen. As soon as the mouse button is released, the command blinks briefly, the menu disappears, and the command is executed. (The user can set the number of times the command blinks in the Control Panel desk accessory.) The menu title in the menu bar remains highlighted until the command has completed execution.

Nothing actually happens until the user chooses the command; the user can look at any of the menus without making a commitment to do anything.

The most frequently used commands should be at the top of a menu; research shows that the easiest item for the user to choose is the second item from the top. The most dangerous commands should be at the bottom of the menu, preferably isolated from the frequently used commands.

Appearance of Menu Commands

The commands in a particular menu should be logically related to the title of the menu. In addition to command names, three features of menus help the user understand what each command does: command groups, toggles, and special visual features.

Command Groups

As mentioned above, menu commands can be divided into two kinds: verbs and adjectives, or actions and attributes. An important difference between the two kinds of commands is that an attribute stays in effect until it's cancelled, while an action ceases to be relevant after it has been performed. Each of these two kinds can be grouped within a menu. Groups are separated by gray lines, which are implemented as disabled commands.

The most basic reason to group commands is to break up a menu so it's easier to read. Commands grouped for this reason are logically related, but independent. Commands that are actions are usually grouped this way, such as Cut, Copy, Paste, and Clear in the Edit menu.

Attribute commands that are interdependent are grouped to show this interdependence. Two kinds of attribute command groups are mutually exclusive groups and accumulating groups.

In a mutually exclusive attribute group, only one command in the group is in effect at the same time. The command that's in effect is preceded by a check mark. If the user chooses a different command in the group, the check mark is moved to the new command. An example is the Font menu in MacWrite; no more than one font can be in effect at a time.

In an accumulating attribute group, any number of attributes can be in effect at the same time. One special command in the group cancels all the other commands. An example is the Style menu in MacWrite: the user can choose any combination of Bold, Italic, Underline, Outline, or Shadow, but Plain Text cancels all the other commands.

Toggles

Another way to show the presence or absence of an attribute is by a toggled command. In this case, the attribute has two states, and a single command allows the user to toggle between the states. For example, when rulers are showing in MacWrite, a command in the Format menu reads "Hide Rulers". If the user chooses this command, the rulers are hidden, and the command is changed to read "Show Rulers". This kind of group should be used only when the wording of the commands makes it obvious that they're opposites.

Special Visual Features

In addition to the command names and how they're grouped, several other features of commands communicate information to the user:

- A check mark indicates whether an attribute command is currently in effect.
- An ellipsis (...) after a command name means that choosing that command brings up a dialog box. The command isn't actually executed until the user has finished filling in the dialog box and has clicked the OK button or its equivalent.
- The application dims a command when the user can't choose it. If the user moves the pointer over a dimmed item, it isn't highlighted.
- If a command can be chosen from the keyboard, it's followed by the Command key symbol and the character used to choose it. To choose a command this way, the user holds down the Command key and then presses the character key.

Some characters are reserved for special purposes, but there are different degrees of stringency. Since almost every application has a File menu and an Edit menu, the keyboard equivalents in those menus are strongly reserved, and should never be used for any other purpose:

Character	Command
C	Copy (Edit menu)
Q	Quit (File menu)
V	Paste (Edit menu)
X	Cut (Edit menu)
Z	Undo (Edit menu)

The keyboard equivalents in the Style menu are conditionally reserved. If an application has this menu, it shouldn't use these characters for any other purpose, but if it doesn't, it can use them however it likes:

Character	Command
B	Bold
I	Italic
O	Outline
P	Plain text
S	Shadow
U	Underline

One keyboard command doesn't have a menu equivalent:

Character	Command
.	Stop current operation

Several other menu features are also supported:

- A command can be shown in Bold, Italic, Outline, Underline, or Shadow type style.

- A command can be preceded by an icon.

- The application can draw its own type of menu. An example of this is the Fill menu in MacDraw.

STANDARD MENUS

One of the strongest ways in which Macintosh applications can take advantage of the consistency of the user interface is by using standard menus. The operations controlled by these menus occur so frequently that it saves considerable time for users if they always match exactly. Three of these menus, the Apple, File, and Edit menus, appear in almost every application. The Font, FontSize, and Style menus affect the appearance of text, and appear only in applications where they're relevant.

The Apple Menu

Macintosh doesn't allow two applications to be running at once. Desk accessories, however, are mini-applications that are available while using any application.

At any time the user can issue a command to call up one of several desk accessories; the available accessories are listed in the Apple menu, as shown in Figure 16.

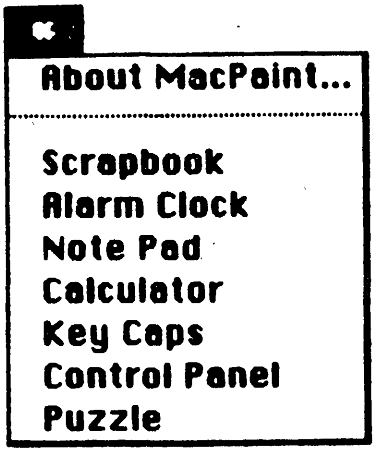


Figure 16. Apple Menu

Accessories are disk-based: Only those accessories on an available disk can be used. The list of accessories is expanded or reduced according to what's available. More than one accessory can be on the desktop at a time.

For a description of these desk accessories, see Macintosh, the owner's guide. An application can also provide its own desk accessories.

The Apple menu also contains the "About xxx" menu item, where "xxx" is the name of the application. Choosing this item brings up a dialog box with the name and copyright information for the application, as well as any other information the application wants to display.

The File Menu

The File menu allows the user to perform certain simple filing operations without leaving the application and returning to the Finder. It also contains the commands for printing and for leaving the application. The standard File menu includes the commands shown in Figure 17.

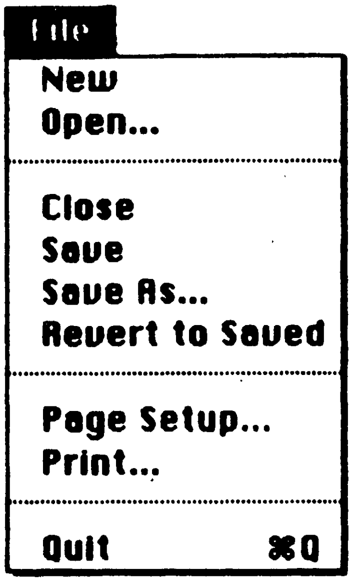


Figure 17. File Menu

Other frequently used commands are Print Draft, Print Final, and Print One. All of these commands are described below.

New

New opens a new, untitled document. The user names the document the first time it's saved. This command is disabled when the maximum number of documents allowed by the application is already open.

Open

Open opens an existing document. To select the document, the user is presented with a dialog box (Figure 18). This dialog box shows a list of all the documents on the disk whose name is displayed that can be handled by the current application. The user can scroll this list forward and backward. The dialog box also gives the user the chance to look at the documents on the disk in the other disk drive that belong to the current application, or to eject either disk.

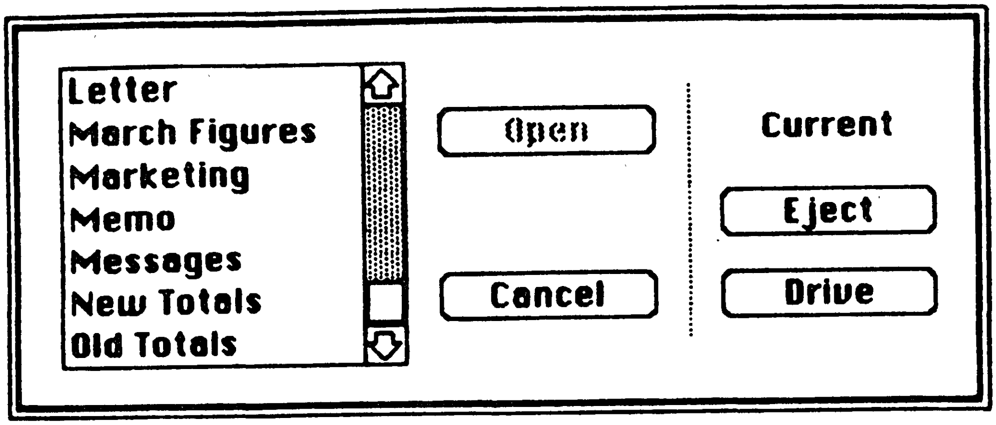


Figure 18. Open Dialog Box

Using the Open command, the user can only open a document that can be processed by the current application. Opening a document that can only be processed by a different application requires leaving the application and returning to the Finder.

This command is disabled when the maximum number of documents allowed by the application is already open.

Close

Close closes the active document or desk accessory. If the user has changed the document since the last time it was saved, the command presents an alert box giving the user the choice of whether or not to save the changes.

Clicking in the close box of a window is the same as choosing Close.

Save

Save makes permanent any changes to the active document since the last time it was saved. It leaves the document open.

If the user chooses Save for a new document that hasn't been named yet, the application presents the Save As dialog (see below) to name the document, and then continues with the save. The active document remains active.

If there's not enough room on the disk to save the document, the application asks if the user wants to save the document on another disk. If the answer is yes, the application goes through the Save As dialog to find out which disk.

Save As

Save As saves a copy of the active document under a file name provided by the user.

If the document already has a name, Save As closes the old version of the document, creates a copy, and displays the copy in the window.

If the document is untitled, Save As saves the original document under the specified name. The active document remains active.

Revert to Saved

Revert to Saved returns the document to the state it was in the last time it was saved. Before doing so, it puts up an alert box to confirm that this is what the user wants.

Page Setup

Page Setup lets the user specify printing parameters such as what its paper size and printing orientation are. These parameters remain with the document.

Print

Print lets the user specify various parameters such as print quality and number of copies, and then prints the document. The parameters apply only to the current printing operation.

Quit

Quit leaves the application and returns to the Finder. If any open documents have been changed since the last time they were saved, the application presents the same alert box as for Close, once for each document.

Other Commands

Other commands that are in the File menu in some applications include:

- Print Draft. This command prints one copy of a rough version of a document more quickly than Print. It's useful in applications where ordinary printing is slow. If an application has this command, it should change the name of the Print command to Print Final.

- Print One. This command saves time by printing one copy using default parameters without bringing up the Print dialog box.

The Edit Menu

The Edit menu contains the commands that delete, move, and copy objects, as well as commands such as Undo, Show Clipboard, and Select All. This section also discusses the Clipboard, which is controlled by the Edit menu commands. Text editing methods that don't use menu commands are discussed under "Text Editing".

If the application supports desk accessories, the order of commands in the Edit menu should be exactly as shown here. This is because, by default, the application passes the numbers, not the names, of the menu commands to the desk accessories. (For more details, see the Desk Manager manual.) In particular, your application must provide an Undo command for the benefit of the desk accessories, even if it doesn't support the command (in which case it can disable the command until a desk accessory is opened).

The standard order of commands in the Edit menu is shown in Figure 19.

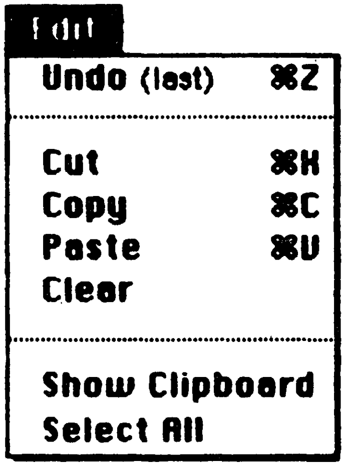


Figure 19. Edit Menu

The Clipboard

The Clipboard is a special kind of window with a well-defined function: it holds whatever is cut or copied from a document. Its contents stay intact when the user changes documents, opens a desk accessory, or leaves the application. An application can choose whether to have the Clipboard open or closed when the application starts up.

The Clipboard looks like a document window, with a close box but with no scroll bars. Its contents cannot be edited.

Every time the user performs a Cut or Copy on the current selection, a copy of the selection replaces the previous contents of the Clipboard. The previous contents are kept around in case the user chooses Undo.

The user can see the contents of the Clipboard but can't edit them. In most other ways the Clipboard behaves just like any other window.

There is only one Clipboard, which is present for all applications that support Cut, Copy, and Paste. The user can see the Clipboard window by choosing Show Clipboard from the Edit menu. If the window is already showing, it's hidden by choosing Hide Clipboard. (Show Clipboard and Hide Clipboard are a single toggled command.)

Because the contents of the Clipboard remain unchanged when applications begin and end, or when the user opens a desk accessory, the Clipboard can be used for transferring data among mutually compatible applications and desk accessories.

Undo

Undo reverses the effect of the previous operation. Not all operations can be undone; the definition of an undoable operation is somewhat application-dependent. The general rule is that operations that change the contents of the document are undoable, and operations that don't are not. Most menu items are undoable, and so are typing sequences.

A typing sequence is any sequence of characters typed from the keyboard or numeric keypad, including Backspace, Return, and Tab, but not including keyboard equivalents of commands.

Operations that aren't undoable include selecting, scrolling, and splitting the window or changing its size or location. None of these operations interrupts a typing sequence. That is, if the user types a few characters and then scrolls the document, the Undo command still undoes the typing. Whenever the location affected by the Undo operation isn't currently showing on the screen, the application should scroll the document so the user can see the effect of the Undo.

An application should also allow the user to undo any operations that are initiated directly on the screen, without a menu command. This includes operations controlled by setting dials, clicking check boxes, and so on, as well as drawing graphic objects with the mouse.

The actual wording of the Undo command as it appears in the Edit menu is "Undo xxx", where xxx is the name of the last operation. If the last operation isn't a menu command, use some suitable term after the word Undo. If the last operation can't be undone, the command reads "Undo", but is disabled.

If the last operation was Undo, the menu command says "Redo xxx", where xxx is the operation that was undone. If this command is chosen, the Undo is undone.

Cut

The user chooses Cut either to delete the current selection or to move it. If it's a move, it's eventually completed by choosing Paste.

When the user chooses Cut, the application removes the current selection from the document and puts it in the Clipboard, replacing the Clipboard's previous contents. The place where the selection used to be becomes the new selection; the visual implications of this vary among applications. For example, in text, the new selection is an insertion point, while in an array, it's an empty but highlighted cell. If the user chooses Paste immediately after choosing Cut, the document should be just as it was before the cut; the Clipboard is unchanged.

When the user chooses Cut, the application doesn't know if it's a deletion or the first step of a move. Therefore, it must be prepared for either possibility.

Copy

Copy is the first stage of a copy operation. Copy puts a copy of the selection in the Clipboard, but the selection also remains in the document.

Paste

Paste is the last stage of a copy or move operation. It pastes the contents of the Clipboard to the document, replacing the current selection. The user can choose Paste several times in a row to paste multiple copies. After a paste, the new selection is the object that was pasted, except in text, where it's an insertion point immediately after the pasted text. The Clipboard remains unchanged.

Clear

When the user chooses Clear, or presses the Clear key on the numeric keypad, the application removes the selection, but doesn't put it on the Clipboard. The new selection is the same as it would be after a Cut.

Show Clipboard

Show Clipboard is a toggled command. Initially, the Clipboard isn't displayed, and the command is "Show Clipboard". If the user chooses the command, the Clipboard is displayed and the command changes to "Hide Clipboard".

Select All

Select All selects every object in the document.

Font-Related Menus

Three standard menus affect the appearance of text: Font, which determines the font of a text selection; FontSize, which determines the size of the characters; and Style, which determines aspects of its appearance such as boldface, italics, and so on.

Font Menu

A font is a set of typographical characters created with a consistent design. Things that relate characters in a font include the thickness of vertical and horizontal lines, the degree and position of curves and swirls, and the use of serifs. A font has the same general appearance, regardless of the size of the characters. The Font menu always lists the fonts that are currently available. Figure 2Ø shows a Font menu with some of the most common fonts.

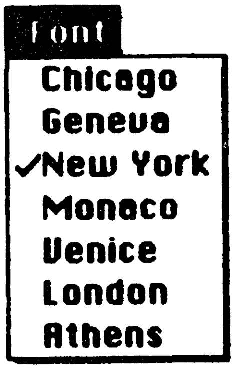


Figure 2Ø. Font Menu

FontSize Menu

Font sizes are measured in points; a point is about 1/72 of an inch. Each font is available in predefined sizes. The numbers of these sizes for each font are shown outlined in the FontSize menu. The font can also be scaled to other sizes, but it may not look as good. Figure 21 shows a FontSize menu with the standard font sizes.

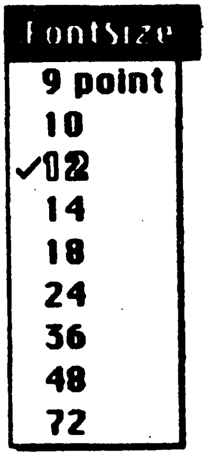


Figure 21. FontSize Menu

If there's insufficient room in the menu bar for the word FontSize, it can be abbreviated to Size. If there's insufficient room for both a Font menu and a Size menu, the sizes can be put at the end of the Font or Style menu.

Style Menu

The commands in the Style menu are Plain Text, Bold, Italic, Underline, Outline, and Shadow. All the commands except Plain Text are accumulating attributes; the user can choose any combination. They are also toggled commands; a command that's in effect for the current selection is preceded by a check mark. Plain Text cancels all the other choices. Figure 22 shows these styles.

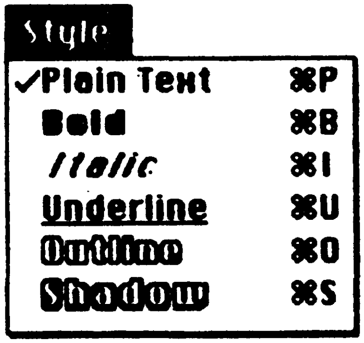


Figure 22. Style Menu

TEXT EDITING

In addition to the operations described under "The Edit Menu" above, there are other ways to edit text that don't use menu items.

Inserting Text

To insert text, the user selects an insertion point by clicking where the text is to go, and then starts typing it. As the user types, the application continually moves the insertion point to the right of each new character.

Applications with multiline text blocks should support word wraparound, according to the definition of a word given above. The intent is that no word be broken between lines.

Backspace

When the user presses the Backspace key, one of two things happens:

- If the current selection is one or more characters, it's deleted.
- If the current selection is an insertion point, the previous character is deleted.

In both cases, the deleted characters don't go into the Clipboard, and the insertion point replaces the deleted characters in the document.

Replacing Text

If the user starts typing when the selection is one or more characters, the characters that are typed replace the selection. The deleted characters don't go into the Clipboard, but the replacement can be undone by immediately choosing Undo.

Intelligent Cut and Paste

An application that lets the user select a word by double-clicking should also see to it that the user doesn't regret using this feature. The only way to do this is by providing "intelligent" cut and paste.

To understand why this feature is necessary, consider the following sequence of events in an application that doesn't provide it:

1. A sentence in the user's document reads: "Returns are only accepted if the merchandise is damaged." The user wants to change this to: "Returns are accepted only if the merchandise is damaged."

2. The user selects the word "only" by double-clicking. The letters are highlighted, but not either of the adjacent spaces.

3. The user chooses Cut, clicks just before the word "if", and chooses Paste.

4. The sentence now reads: "Returns are accepted onlyif the merchandise is damaged." To correct the sentence, the user has to remove a space between "are" and "accepted", and add one between "only" and "if". At this point he or she may be wondering why the Macintosh is supposed to be easier to use than other computers.

If an application supports intelligent cut and paste, the rules to follow are:

- If the user selects a word or a range of words, highlight the selection, but not any adjacent spaces.

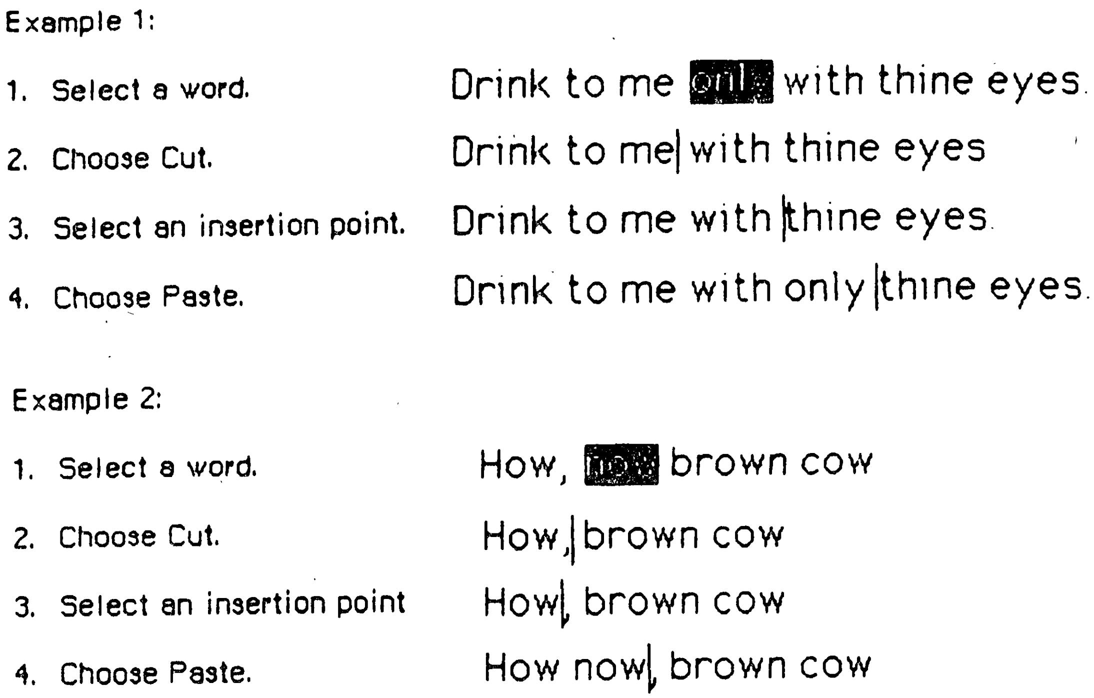
- When the user chooses Cut, if the character to the left of the selection is a space, discard it.

- When the user chooses Paste, if the character to the left of the current selection isn't a space, add a space. If the character to the right of the current selection isn't a punctuation mark or a space, add a space. Punctuation marks include the period, comma, exclamation point, question mark, apostrophe, colon, semicolon, and quotation mark.

This feature makes more sense if the application supports the full definition of a word (as detailed above under "Selecting a Word"), rather than the definition of a word as anything between two spaces.

These rules apply to any selection that's one or more whole words, whether it was chosen with a double-click or as a range selection.

Figure 23 shows some examples of intelligent cut and paste.



Example 1:

1. Select a word. Drink to me only with thine eyes.
2. Choose Cut. Drink to me|with thine eyes
3. Select an insertion point. Drink to me with |thine eyes.
4. Choose Paste. Drink to me with only |thine eyes.

Example 2:

1. Select a word. How, now brown cow
2. Choose Cut. How,|brown cow
3. Select an insertion point How|, brown cow
4. Choose Paste. How now|, brown cow

Figure 23. Intelligent Cut and Paste

Editing Fields

If an application isn't primarily a text application, but does use text in fields (such as in a dialog box), it may not be able to provide the full text editing capabilities described so far.

It's important, however, that whatever editing capabilities the application provides under these circumstances be upward-compatible with the full text editing capability. The following list shows the capabilities that can be provided, going from the minimal to the most sophisticated:

- The user can select the whole field and type in a new value.

- The user can backspace.

- The user can select a substring of the field and replace it.

- The user can select a word by double-clicking.

- The user can choose Undo, Cut, Copy, Paste, and Clear, as described above under "The Edit Menu". In the most sophisticated version, the application implements intelligent cut and paste.

An application should also perform appropriate edit checks. For example, if the only legitimate value for a field is a string of digits, the application might issue an alert if the user typed any nondigits. Alternatively, the application could wait until the user is through typing before checking the validity of the contents of the field. In this case, the appropriate time to check the field is when the user clicks anywhere other than within the field.

DIALOGS AND ALERTS

The "select-then-choose" paradigm is sufficient whenever operations are simple and act on only one object. But occasionally a command will require more than one object, or will need additional parameters before it can be executed. And sometimes a command won't be able to carry out its normal function, or will be unsure of the user's real intent. For these special circumstances the Macintosh user interface includes two additional features:

- dialogs, to allow the user to provide additional information before a command is executed

- alerts, to notify the user whenever an unusual situation occurs

Since both of these features lean heavily on controls, controls are described in this section, even though controls are also used in other places.

Controls

Friendly systems act by direct cause-and-effect; they do what they're told. Performing actions on a system in an indirect fashion reduces the sense of direct manipulation. To give Macintosh users the feeling that they're in control of their machines, many of an application's features are implemented with controls: graphic objects that, when directly manipulated by the mouse, cause instant action with visible results.

There are four main types of controls: buttons, check boxes, radio buttons, and dials. These four kinds are shown in Figure 24.

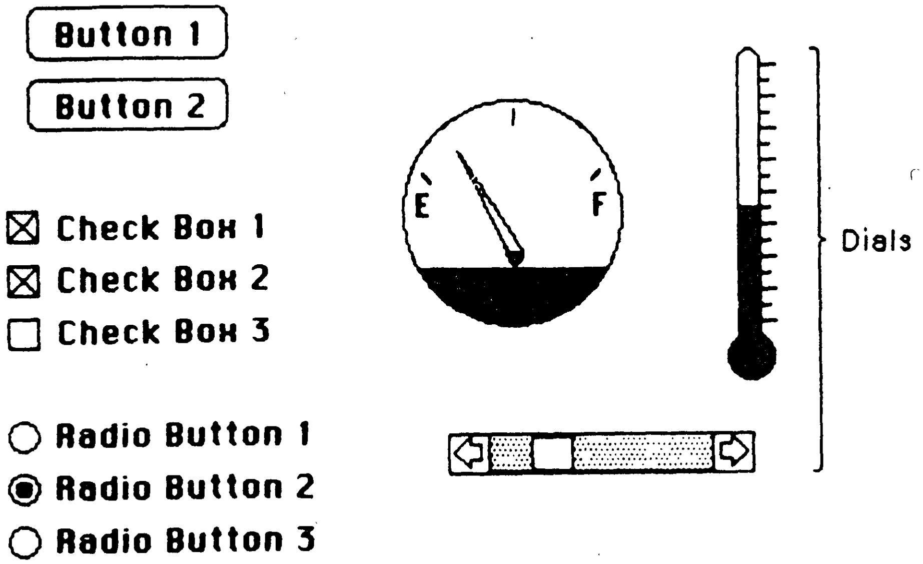


Figure 24. Controls

Buttons

Buttons are small objects, usually inside a window, labeled with text. Clicking or pressing a button performs the action described by the button's label.

Buttons perform instantaneous actions, such as completing operations defined by a dialog box or acknowledging error messages. Conceivably they could perform continuous actions, in which case the effect of pressing on the button would be the same as the effect of clicking it repeatedly.

Two particular buttons, OK and Cancel, are especially important in dialogs and alerts; they're discussed under those headings below.

Check Boxes and Radio Buttons

Whereas buttons perform instantaneous or continuous actions, check boxes and radio buttons let the user choose among alternative values for a parameter.

Check boxes act like toggle switches; they're used to indicate the state of a parameter that must be either off or on. The parameter is on if the box is checked, otherwise it's off. The check boxes appearing together in a given context are independent of each other; any number of them can be off or on.

Radio buttons typically occur in groups; they're round and are filled in with a black circle when on. They're called radio buttons because they act like the buttons on a car radio. At any given time, exactly one button in the group is on. Clicking one button in a group turns off the current button.

Both check boxes and radio buttons are accompanied by text that identifies what each button does.

Dials

Dials display the value, magnitude, or position of something in the application 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 most common example of a dial is the scroll bar. The indicator of the scroll bar is the scroll box; it represents the position of the window over the length of the document. The user can drag the scroll box to change that position.

Dialogs

Commands in menus normally act on only one 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. The user can tell which commands bring up dialog boxes because they're followed by an ellipsis (...) in the menu.

A dialog box is a rectangle that may contain text, controls, and icons. There should be some text in the box that indicates which command brought up the dialog box.

Other than explanatory text, the contents of a dialog box are all objects that the user sets to provide the needed information. These objects include controls and text fields. When the application puts up the dialog box, it should set the controls to some default setting and fill in the text fields with default values, if possible. One of the text fields (the "first" field) should be highlighted, so that the user can change its value just by typing in the new value. If all the text fields are blank, there should be an insertion point in the first field.

Editing text fields in a dialog box should conform to the guidelines detailed above, under "Text Editing".

When the user is through editing an item:

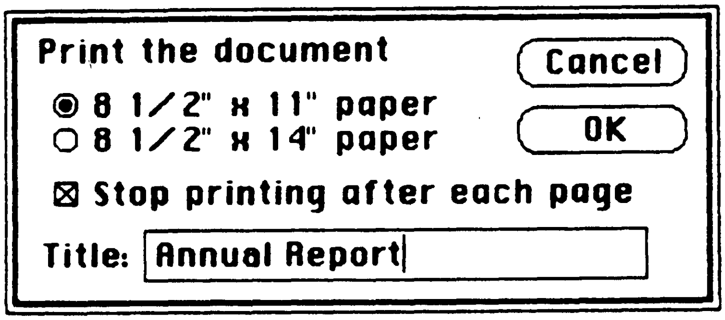
- Pressing Tab accepts the changes made to the item, and selects the next item in sequence.

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

Dialog boxes are either modal or modeless, as described below.

Modal Dialog Boxes

A modal dialog box is one that the user must explicitly dismiss before doing anything else, such as making a selection outside the dialog box or choosing a command. Figure 25 shows a modal dialog box.



Print the document

◉ 8 1/2" x 11" paper
○ 8 1/2" x 14" paper

☒ Stop printing after each page

Title: Annual Report

Figure 25. A Modal Dialog Box

Because it restricts the user's freedom of action, this type of dialog box should be used sparingly. In particular, the user can't choose a menu item while a modal dialog box is up, and therefore can only do the simplest kinds of text editing.

For these reasons, the main use of a modal dialog box is when it's important for the user to complete an operation before doing anything else.

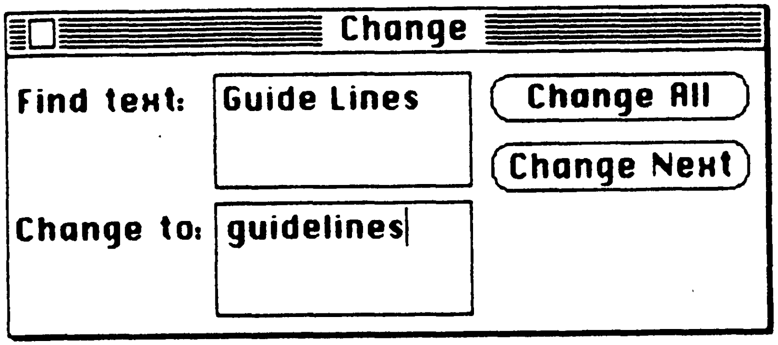
A modal dialog box usually has at least two buttons: OK and Cancel. OK dismisses the dialog box and performs the original command according to the information provided; it can be given a more descriptive name than "OK". Cancel dismisses the dialog box and cancels the original command; it must always be called "Cancel".

A dialog box can have other kinds of buttons as well; these may or may not dismiss the dialog box. One of the buttons in the dialog box may be outlined boldly. The outlined button is the default button; if no button is outlined, then the OK button is the default button. The default button should be the safest button in the current situation. Pressing the Return or Enter key has the same effect as clicking the default button. If there is no default button, then Return and Enter have no effect.

A special type of modal dialog box is one with no buttons. This type of box is just to inform the user of a situation without eliciting any response. Usually, it would describe the progress of an ongoing operation. Since it has no buttons, the user has no way to dismiss it. Therefore, the application must leave it up long enough for the user to read it before taking it down again.

Modeless Dialog Boxes

A modeless dialog box allows the user to perform other operations without dismissing the dialog box. Figure 26 shows a modeless dialog box.



Find text: Guide Lines

Change to: guidelines

Figure 26. A Modeless Dialog Box

A modeless dialog box is dismissed by clicking in the close box or by choosing Close when the dialog is active. The dialog box is also dismissed implicitly when the user chooses Quit. It's usually a good idea for the application to remember the contents of the dialog box after it's dismissed, so that when it's opened again, it can be restored exactly as it was.

Controls work the same way in modeless dialog boxes as in modal dialog boxes, except that buttons never dismiss the dialog box. In this context, the OK button means "go ahead and perform the operation, but leave the dialog box up", while Cancel usually terminates an ongoing operation.

A modeless dialog box can also have text fields; since the user can choose menu commands, the full range of editing capabilities can be made available.

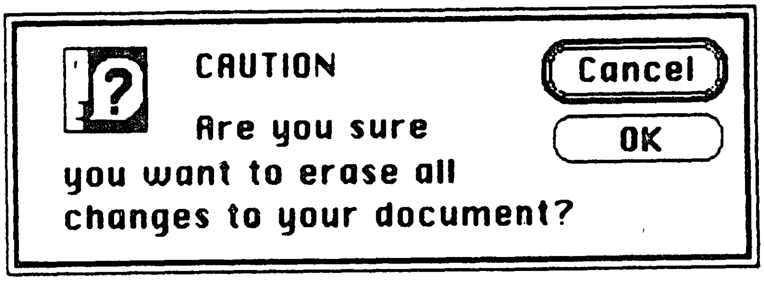
Alerts

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 disk, users will do things an application can't cope with in a normal manner. Alerts give applications a way to respond to errors not only in a consistent manner, but in stages according to the severity of the error, the user's level of expertise, and the particular history of the error.

The two kinds of alerts are beeps and alert boxes.

Beeps are used for errors that are both minor and immediately obvious. For example, if the user tries to backspace past the left boundary of a text field, the application could choose to beep instead of putting up an alert box. A beep can also be part of a staged alert, as described below.

An alert box looks like a modal dialog box, except that it's somewhat narrower and appears lower on the screen. An alert box is primarily a one-way communication from the system to the user; the only way the user can respond is by clicking buttons. Therefore alert boxes might contain dials and buttons, but usually not text fields, radio buttons, or check boxes. Figure 27 shows a typical alert box.


CAUTION

Are you sure you want to erase all changes to your document?

Figure 27. An Alert Box

There are three types of alert boxes:

- Note: A minor mistake that wouldn't have any disastrous consequences if left as is.

- Caution: An operation that may or may not have undesirable results if it is allowed to continue. The user is given the choice whether or not to continue.

- Stop: A situation that requires remedial action by the user. The situation could be either a serious problem, or something as simple as a request by the application to the user to change diskettes.

An application can define several stages for an alert, so that if the user persists in the same mistake, the application can issue increasingly more helpful (or sterner) messages. A typical sequence is for the first two occurrences of the mistake to result in a beep, and for subsequent occurrences to result in an alert box. This type of sequence is especially appropriate when the mistake is one that has a high probability of being accidental. An example is when the user chooses Cut when the selection is an insertion point.

How the buttons in an alert box are labeled depends on the nature of the box. If the box presents the user with a situation in which no alternative actions are available, the box has a single button that says OK. Clicking this button means "I have read the alert." If the user is given alternatives, then typically the alert is phrased as a question that can be answered "yes" or "no". In this case, buttons labeled Yes and No are appropriate, although some variation such as Save and Don't Save is also acceptable. OK and Cancel can be used, as long as their meaning isn't ambiguous.

The preferred (safest) button to use in the current situation is boldly outlined. This is the alert's default button; its effect occurs if the user presses Return or Enter.

It's important to phrase messages in alert boxes so that users aren't left guessing the real meaning. Avoid computer jargon.

Use icons whenever possible. Graphics can better describe some error situations than words, and familiar icons help users distinguish their alternatives better. Icons should be internationally comprehensible; they should not contain any words, or any symbols that are unique to a particular country.

Generally, it's 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 an adequate description of the information needed by the user to take appropriate action.

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?

DO'S AND DON'TS OF A FRIENDLY USER INTERFACE

Do:

- Let the user have as much control as possible over the appearance of objects on the screen--their arrangement, size, and visibility.

- Use verbs as menu commands.

- Make alert messages self-explanatory.

- Use controls and other graphics instead of just menu commands.

- Take the time to use good graphic design; it really helps.

Don't:

- Overuse modes, including modal dialog boxes.

- Require using the keyboard for an operation that would be easier with the mouse, or require using the mouse for an operation that would be easier with the keyboard.

- Change the way the screen looks unexpectedly, especially by scrolling automatically more than necessary.

- Make up your own menus and then give them the same names as standard menus.

- Take an old-fashioned prompt-based application originally developed for another machine and pass it off as a Macintosh application.

Macintosh Memory Management: An Introduction /MEM/INTRO

See Also: The Memory Manager: A Programmer's Guide
Programming Macintosh Applications in Assembly Language

Modification History: First Draft Steve Chernicoff and
Bradley Hacker 8/20/84

ABSTRACT

This manual contains the minimum information needed about memory management on the Macintosh. Memory management is covered in greater detail in the manual The Memory Manager: A Programmer's Guide.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual contains the minimum information needed about memory management on the Macintosh. *** Eventually it will form an early chapter in the comprehensive Inside Macintosh manual. *** Memory management is covered in greater detail in the Memory Manager manual.

This manual assumes you're familiar with Lisa Pascal and the information in Inside Macintosh: A Road Map.

THE STACK AND THE HEAP

A running program can dynamically allocate and release memory in two different ways: from the stack or the heap. The stack is an area of memory that can grow or shrink at one end while the other end remains fixed, as shown in Figure 1. This means that space on the stack is always allocated and released in LIFO (last-in-first-out) order: the last item allocated is always the first to be released. It also means that the allocated area of the stack is always contiguous. Space is released only at the top of the stack, never in the middle, so there can never be any unallocated "holes" in the stack.

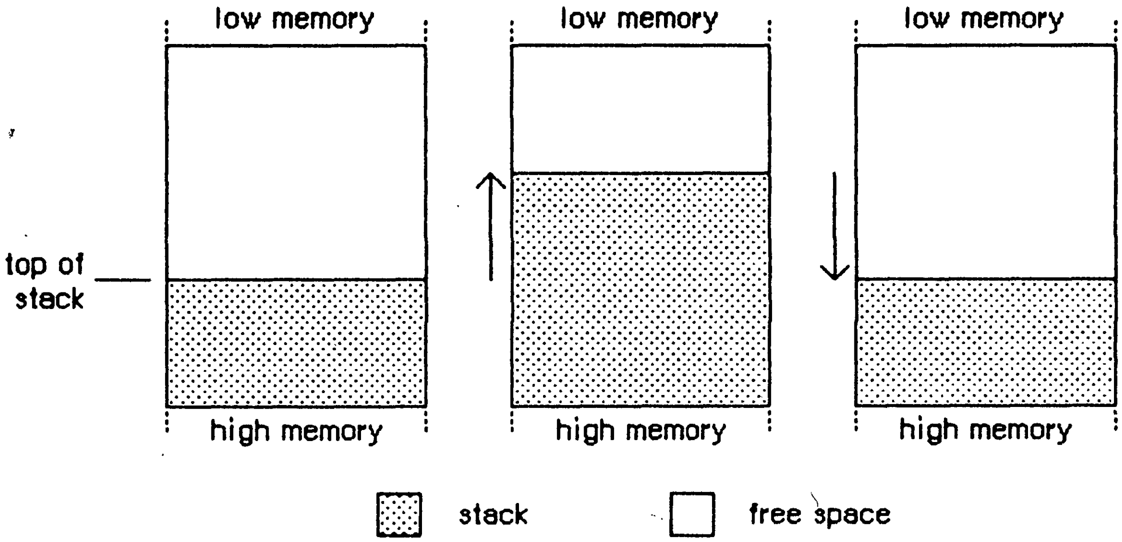


Figure 1. The Stack

By convention, the stack grows from high toward low memory addresses. The end of the stack that grows and shrinks is usually referred to as the "top" of the stack, even though it's actually at the lower end of the stack in memory.

The other method of dynamic memory allocation is from the heap. Unlike stack space, which is implicitly tied to a program's subroutine structure, heap space is allocated and released only at the program's explicit request. In Pascal, objects on the heap are referred to by means of pointers instead of by name.

Space on the heap is allocated in blocks, which may be of any size needed for a particular object. The Macintosh Operating System's Memory Manager does all the necessary "housekeeping" to keep track of the blocks as they're allocated and released. Because these operations can occur in any order, the heap doesn't grow and shrink in an orderly way like the stack. After a program has been running for a while, the heap tends to become fragmented into a patchwork of allocated and free blocks, as shown in Figure 2.

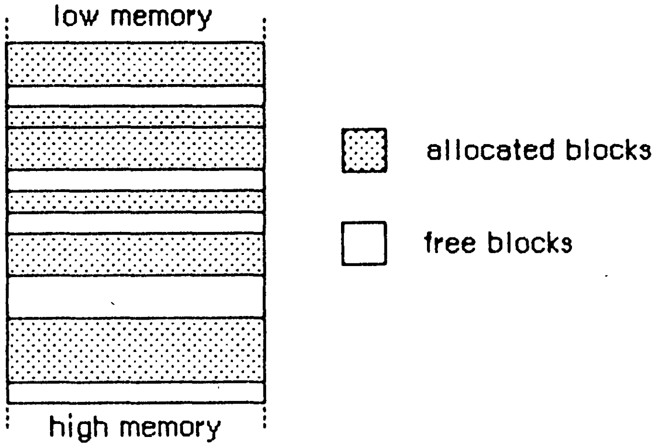


Figure 2. A Fragmented Heap

As a result of heap fragmentation, when the program asks to allocate a new block of a certain size, it may be impossible to satisfy the request even though there's enough free space available, because the space is broken up into blocks smaller than the requested size. When this happens, the Memory Manager will try to create the needed space by compacting the heap: moving already allocated blocks together in order to collect the free space into a single larger block (see Figure 3).

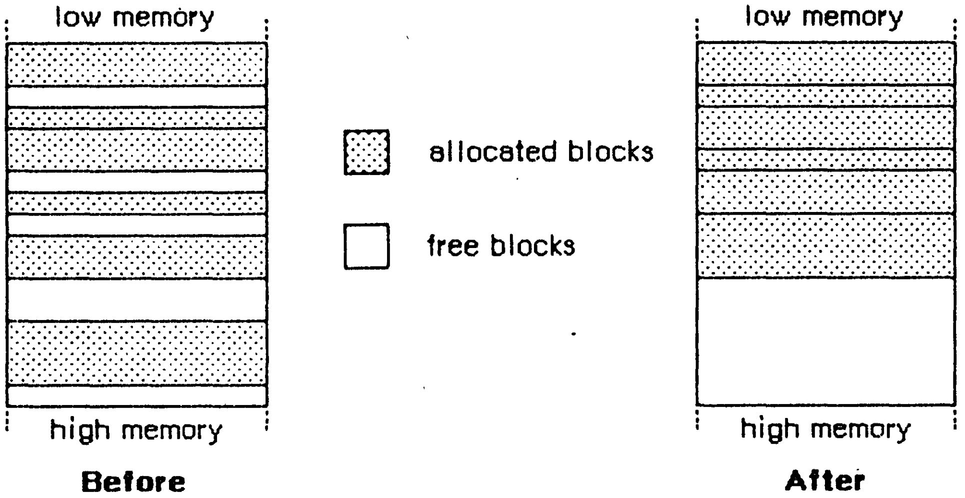


Figure 3. Heap Compaction

There are always two independent heap areas in memory: the system heap, which is used by the Toolbox and Operating System, and the application heap, which is used by the application program.

POINTERS AND HANDLES

The Memory Manager contains a few fundamental routines for allocating and releasing heap space. The NewPtr function allocates a block on the heap of a requested size and returns a pointer to the block. The DisposPtr procedure releases the block the variable points to and sets the variable to NIL.

For example, after the declarations

```
      TYPE ThingPtr = ^Thing;
           Thing    = RECORD
                        . . .
                      END;

      VAR aThingPtr: ThingPtr;
```

the statement

```
      aThingPtr := NewPtr(SIZEOF(Thing))
```

will allocate heap space for a new variable of type Thing and set aThingPtr to point to it. The amount of space to be allocated is determined by the size of Thing. To allocate a 2K-byte memory block, you can use:

```
      aThingPtr := NewPtr($2ØØØ)
```

Once you've used NewPtr to allocate a block and obtain a pointer to it, you can make as many copies of the pointer as you need and use them in any way your program requires. When you're finished with the block, you can release the memory it occupies (returning it to available free space) with the statement

```
      DisposPtr(aThingPtr)
```

Any pointers you may have to the block are now invalid, since the block they're supposed to point to no longer exists. You have to be careful not to use such "dangling" pointers. This type of bug can be very difficult to diagnose and correct, since its effects typically aren't discovered until long after the pointer is left dangling.

Another way a pointer can be left dangling is for its underlying block to be moved to a different location within the heap. To avoid the problem, blocks that are referred to through simple pointers, as in Figure 4, are nonrelocatable. The Memory Manager will never move a nonrelocatable block, so you can rely on all pointers to it to remain correct for as long as the block remains allocated.

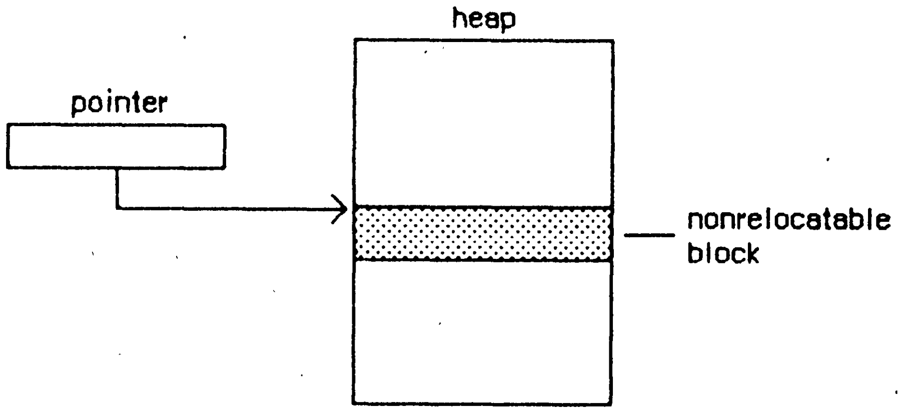


Figure 4. A Pointer to a Nonrelocatable Block

If all blocks on the heap were nonrelocatable, there would be no way to prevent the heap's free space from becoming fragmented. Since the Memory Manager needs to be able to move blocks around in order to compact the heap, it also uses relocatable blocks. (All the allocated blocks shown earlier in Figure 3, the illustration of heap compaction, are relocatable.) To keep from creating dangling pointers, the Memory Manager maintains a single master pointer to each relocatable block. Whenever a relocatable block is created, a master pointer is allocated from the heap at the same time and set to point to the block. All references to the block are then made by double indirection, through a pointer to the master pointer, called a handle to the block (see Figure 5). If the Memory Manager needs to move the block during compaction, it has only to update the master pointer to point to the block's new location; the master pointer itself is never moved. Since all copies of the handle point to this same master pointer, they can be relied on not to dangle, even after the block has been moved.

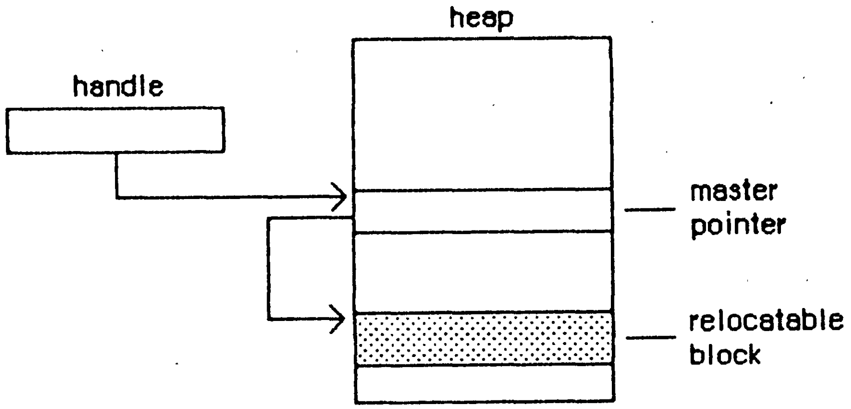


Figure 5. A Handle to a Relocatable Block

Given a handle to an object on the heap, you can access the object itself by double indirection. For example, after the following additional declarations

```
TYPE ThingHandle = ^ThingPtr;

VAR aThingHandle: ThingHandle;
```

you can access the Thing referred to by the handle aThingHandle with the expression

```
aThingHandle^^
```

Once you've allocated a block and obtained a handle to it, you can make as many copies of the handle as you need and use them in any way your program requires. When you're finished with the block, you can free the space it occupies with the statement

```
DisposHandle(aThingHandle)
```

(note)

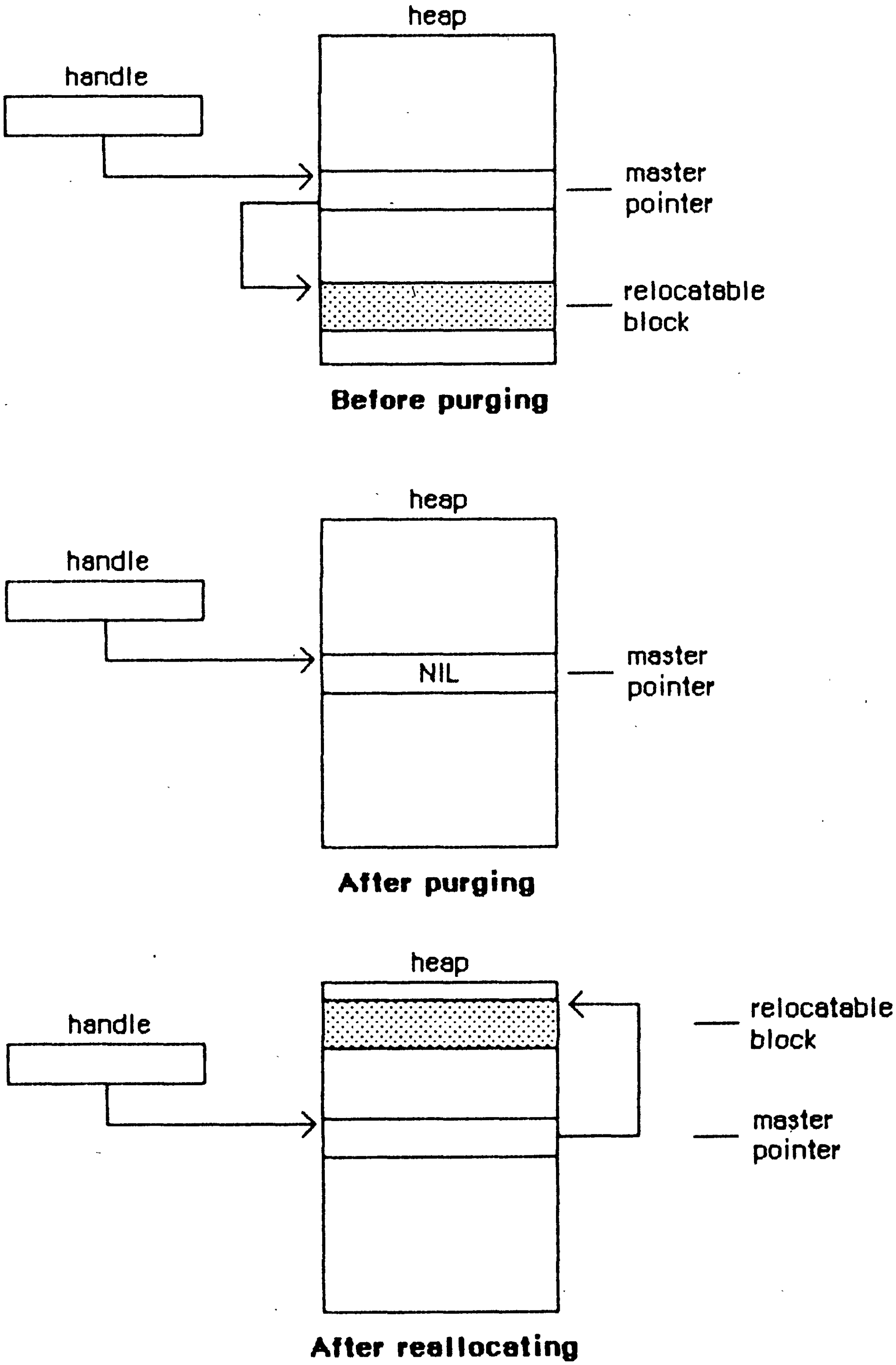
> Toolbox routines that create new objects of various kinds, such as NewWindow and NewControl, implicitly call the NewPtr and NewHandle routines to allocate the space they need. There are also analogous routines for releasing these objects, such as DisposeWindow and DisposeControl.

If the Memory Manager can't allocate a block of a requested size even after compacting the entire heap, it can try to free some space by purging blocks from the heap. Purging a block removes it from the heap and frees the space it occupies. The block's master pointer is set to NIL, but the space occupied by the master pointer itself remains allocated. Any handles to the block now point to a NIL master pointer, and are said to be empty. If your program later needs to refer to the purged block, it can detect that the handle has become empty and ask the Memory Manager to reallocate the block. This operation updates the original master pointer, so that all handles to the block are left referring correctly to its new location (see Figure 6 on the following page).

(warning)

> Reallocating a block recovers only the space it occupies, not its contents. Any information the block contains is lost when the block is purged. It's up to your program to reconstitute the block's contents after reallocating it.

Relocatable and nonrelocatable are permanent properties of a block that can never be changed once the block is allocated. A relocatable block can also be locked or unlocked, purgeable or unpurgeable; your program can set and change these attributes as necessary. Locking a block temporarily prevents it from being moved, even if the heap is compacted. The block can later be unlocked, again allowing the Memory Manager to move it during compaction. A block can be purged only if it's relocatable, unlocked, and purgeable. A newly allocated relocatable block is initially unlocked and unpurgeable.



Before purging

After purging

After reallocating

Figure 6. Purging and Reallocating a Block

GENERAL-PURPOSE DATA TYPES

The Memory Manager includes a number of type definitions for general-purpose use. For working with pointers and handles, there are the following definitions:

```
TYPE SignedByte = -128..127;
     Byte       = Ø..255;
     Ptr        = ^SignedByte;
     Handle     = ^Ptr;
```

SignedByte stands for an arbitrary byte in memory, just to give Ptr and Handle something to point to. You can define a buffer of, say, bufSize untyped memory bytes as a PACKED ARRAY [1..bufSize] OF SignedByte. Byte is an alternative definition that treats byte-length data as unsigned rather that signed quantities.

Because of Pascal's strong typing rules, you can't directly assign a value of type Ptr to a variable of some other pointer type. Instead, you have to convert the pointer from one type to another. For example, after the declarations

```
TYPE Thing = RECORD
               . . .
             END;

     ThingPtr = ^Thing;

VAR aPtr: Ptr;
    aThingPtr: ThingPtr;
```

Lisa Pascal allows you to make aThingPtr point to the same object as aPtr with the assignment

```
aThingPtr := ThingPtr(aPtr)
```

or, you can refer to a field of a record of type Thing with the expression

```
ThingPtr(aPtr)^.field
```

In fact, you can use this same syntax to equate any two variables of the same length. For example:

```
VAR aChar: CHAR;
    aByte: Byte;
    . . .

aByte := Byte(aChar);
```

You can also use the Lisa Pascal functions ORD, ORD4, and POINTER, to convert variables of different length from one type to another. For example:

```
VAR anInteger: INTEGER;
    aLongInt: LONGINT;
    aPointer: Ptr;
    . . .
    anInteger := ORD(aLongInt);      {two low-order bytes only}
    anInteger := ORD(aPointer);      {two low-order bytes only}
    aLongInt := ORD(anInteger);      {packed into high-order bytes}
    aLongInt := ORD4(anInteger);     {packed into low-order bytes}
    aLongInt := ORD(aPointer);
    aPointer := POINTER(anInteger);
    aPointer := POINTER(aLongInt);
```

Assembly-language note: Of course, assembly-language programmers needn't bother with type conversion.

For working with strings, pointers to strings, and handles to strings, the Memory Manager includes the following definitions:

```
TYPE Str255       = STRING[255];
     StringPtr    = ^Str255;
     StringHandle = ^StringPtr;
```

For treating procedures and functions as data objects, there's the ProcPtr data type:

```
TYPE ProcPtr = Ptr;
```

For example, after the declarations

```
VAR aProcPtr: ProcPtr;
    . . .

PROCEDURE MyProc;
  BEGIN
    . . .
  END;
```

you can make aProcPtr point to MyProc by using Lisa Pascal's @ operator, as follows:

```
aProcPtr := @MyProc
```

With the @ operator, you can assign procedures and functions to variables of type ProcPtr, embed them in data structures, and pass them as arguments to other routines. Notice, however, that the data type ProcPtr technically points to an arbitrary byte (SignedByte), not an actual routine. As a result, there's no way in Pascal to access the underlying routine via this pointer in order to call it. Only routines written in assembly language (such as those in the Operating System and the Toolbox) can actually call the routine designated by a pointer of type ProcPtr.

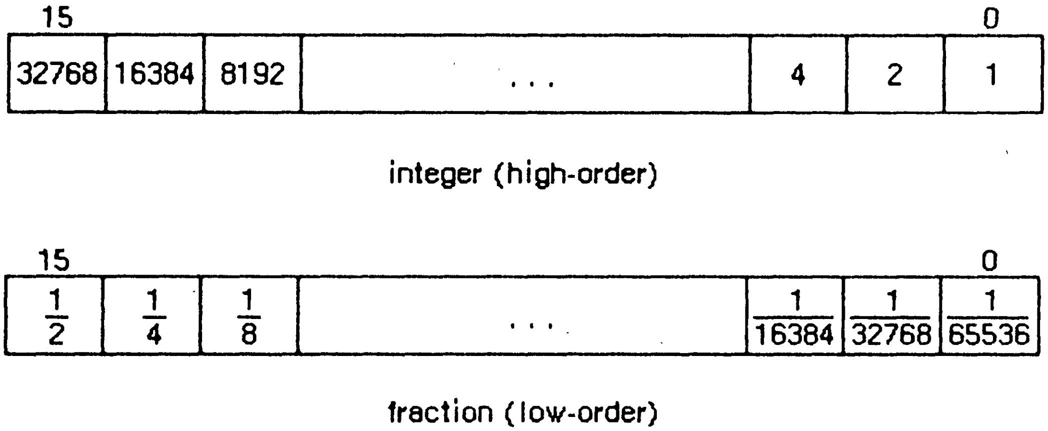
(warning)

> Procedures and functions that are nested within other routines can't be passed with the @ operator.

Finally, for treating long integers as fixed-point numbers, there's the following data type:

```
TYPE Fixed = LONGINT;
```

As illustrated in Figure 7, 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. Negative numbers are the two's complement (formed by inverting each bit and adding 1).



15						0
32768	16384	8192	. . .	4	2	1

integer (high-order)

15						0
$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	. . .	$\frac{1}{16384}$	$\frac{1}{32768}$	$\frac{1}{65536}$

fraction (low-order)

Figure 7. Fixed-Point Numbers

*** (The discussion of Fixed will be removed from the next draft of the Toolbox Utilities manual.) ***

SUMMARY

```
TYPE SignedByte    = -128..127;
     Byte          = Ø..255;
     Ptr           = ^SignedByte;
     Handle        = ^Ptr;

     Str255        = STRING[255];
     StringPtr     = ^Str255;
     StringHandle  = ^StringPtr;

     ProcPtr = Ptr;

     Fixed = LONGINT;
```

GLOSSARY

allocate: To reserve an area of memory for use.

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

block: An area of contiguous memory on the heap.

compaction: The process of moving allocated blocks within the heap in order to collect the free space into a single block.

empty handle: A handle that points to a NIL master pointer, signifying that the underlying relocatable block has been purged.

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.

handle: A pointer to a master pointer, which designates a relocatable block on the heap by double indirection.

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

lock: To temporarily prevent a relocatable block from being moved during heap compaction.

master pointer: A single pointer to a relocatable block, maintained by the Memory Manager and updated whenever the block is moved, purged, or reallocated. All handles to a relocatable block refer to it by double indirection through the master pointer.

nonrelocatable block: A block whose location in the heap is fixed and can't be moved during heap compaction.

purge: To remove a relocatable block from the heap, leaving its master pointer allocated but set to NIL.

purgeable block: A relocatable block that can be purged from the heap.

reallocate: To allocate new space on the heap for a purged block, updating its master pointer to point to its new location.

release: To free an allocated area of memory, making it available for reuse.

relocatable block: A block that can be moved within the heap during compaction.

stack: The area of memory in which space is allocated and released in LIFO (last-in-first-out) order.

system heap: The portion of the heap reserved for use by the Toolbox and Operating System.

unlock: To allow a relocatable block to be moved during heap compaction.

unpurgeable block: A relocatable block that can't be purged from the heap.

MACINTOSH USER EDUCATION

Programming Macintosh Applications in Assembly Language /INTRO/ASSEM

See Also: Inside Macintosh: A Road Map
Macintosh Memory Management: An Introduction
The Memory Manager: A Programmer's Guide
The Operating System Utilities: A Programmer's Guide

Modification History:	First Draft	Steve Chernicoff	2/27/84
	Second Draft	Bradley Hacker	8/2Ø/84
	Third Draft	Caroline Rose	1/22/85

ABSTRACT

This manual gives you general information that you'll need to write all or part of your Macintosh application program in assembly language.

Summary of significant changes and additions since last draft:

- Some additional generally useful global variables are documented (page 4).

- Additions, corrections, and clarifications have been made to the sections "Pascal Data Types" (page 4) and "Calling Conventions" (page 9).

- All illustrations of the stack now place high memory at the top.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual gives you general information that you'll need to write all or part of your Macintosh application program in assembly language. *** Eventually it will become part of the comprehensive Inside Macintosh manual. *** It assumes you already know how to write assembly-language programs for the Motorola MC68ØØØ, the microprocessor in the Macintosh. You should also be familiar with the information in the manuals Inside Macintosh: A Road Map and Macintosh Memory Management: An Introduction.

*** Lisa running MacWorks is called "Macintosh XL" in this manual. ***

DEFINITION FILES

The primary aids to assembly-language programmers are a set of definition files for symbolic names used in assembly-language programs. The definition files include equate files, which equate symbolic names with values, and macro files, which define the macros used to call Toolbox and Operating System routines from assembly language. The equate files define a variety of symbolic names for various purposes, such as:

- useful numeric quantities
- masks and bit numbers
- offsets into data structures
- addresses of global variables (which often in turn contain addresses)

It's a good idea to always use the symbolic names defined in an equate file in place of the corresponding numeric values (even if you know them), since some of these values may change. Note that the names of the offsets for a data structure don't always match the field names in the corresponding Pascal definition. In the 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.

Some generally useful global variables are defined in the equate files as follows:

Name	Contents
OneOne	$ØØØ1ØØØ1
MinusOne	$FFFFFFFF
Lo3Bytes	$ØØFFFFFF
Scratch2Ø	2Ø-byte scratch area
Scratch8	8-byte scratch area
ToolScratch	8-byte scratch area
ApplScratch	12-byte scratch area reserved for use by applications

Scratch2Ø, Scratch8, and ToolScratch will not be preserved across calls to the routines in the Macintosh ROM. ApplScratch will be preserved; it should be used only by application programs and not by desk accessories or other drivers.

PASCAL DATA TYPES

Pascal's strong typing ability lets Pascal programmers write programs without really considering the size of variables. But assembly-language programmers must keep track of the size of every variable. The sizes of the standard Pascal data types, and some of the basic types defined in the Memory Manager, are listed below. (See the Apple Numerics Manual (Apple Product #nnn) *** fill in the number *** for more information about REAL, DOUBLE, EXTENDED, and COMP.)

Type	Size	Contents
INTEGER	2 bytes	Two's complement integer
LONGINT	4 bytes	Two's complement integer
BOOLEAN	1 byte	Boolean value in bit Ø
CHAR	2 bytes	Extended ASCII code in low-order byte
REAL	4 bytes	IEEE standard single format
DOUBLE	8 bytes	IEEE standard double format
EXTENDED	1Ø bytes	IEEE standard extended format
COMP	8 bytes	Two's complement integer with reserved value
STRING[n]	n+1 bytes	Byte containing string length (not counting length byte) followed by bytes containing ASCII codes of characters in string
SignedByte	1 byte	Two's complement integer
Byte	2 bytes	Value in low-order byte
Ptr	4 bytes	Address of data
Handle	4 bytes	Address of master pointer

Other data types are constructed from these. For some commonly used data types, the size in bytes is available as a predefined constant.

Before allocating space for any variable whose size is greater than one byte, Pascal adds "padding" to the next word boundary, if it isn't already at a word boundary. It does this not only when allocating variables declared successively in VAR statements, but also within arrays and records. As you would expect, the size of a Pascal array or record is the sum of the sizes of all its elements or fields (which are stored with the first one at the lowest address). For example, the size of the data type

```
TYPE TestRecord = RECORD
                    testHandle: Handle;
                    testBoolA:  BOOLEAN;
                    testBoolB:  BOOLEAN;
                    testChar:   CHAR
                  END;
```

is eight bytes: four for the handle, one each for the Booleans, and two for the character. If the testBoolB field weren't there, the size would be the same, because of the byte of padding Pascal would add to make the character begin on a word boundary.

In a packed record or array, type BOOLEAN is stored as a bit, and types CHAR and Byte are stored as bytes. The padding rule described above still applies. For example, if the TestRecord data type shown above were declared as PACKED RECORD, it would occupy only six bytes: four for the handle, one for the Booleans (each stored in a bit), and one for the character. If the last field were INTEGER rather than CHAR, padding before the 2-byte integer field would cause the size to be eight bytes.

(note)

> The packing algorithm may not be what you expect. If you need to exactly how data is packed, or if you have questions about the size of a particular data type, the best thing to do is write a test program in Pascal and look at the results. (You can use the SIZEOF function to get the size.)

THE TRAP DISPATCH TABLE

The Toolbox and Operating System reside in ROM. However, to allow flexibility for future development, application code must be kept free of any specific ROM addresses. So all references to Toolbox and Operating System routines are made indirectly through the trap dispatch table in RAM, which contains the addresses of the routines. As long as the location of the trap 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 Toolbox and Operating System routines is encoded in compressed form in the ROM itself. When the system is started up, this encoded information is expanded to form the trap dispatch table. Because the trap dispatch table resides in RAM, 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 trap dispatch table to point to them. It also allows an application program to replace specific Toolbox and Operating System routines with its own "custom" versions. A pair of utility routines for manipulating the trap dispatch table, GetTrapAddress and SetTrapAddress, are described in the Operating System Utilities manual.

For compactness, entries in the trap dispatch table are encoded into one word each, instead of a full long-word address. Since the trap dispatch table is 1Ø24 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; for routines in RAM, it's the beginning of the system heap. The two base addresses are kept in a pair of global variables named ROMBase and RAMBase.

The offset in a trap 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). As illustrated in Figure 1, the system does the following to find the absolute address of the routine:

1. checks the high-order bit of the trap dispatch table entry to find out which base address to use

2. doubles the offset to convert it from words to bytes (by left-shifting one bit)

3. adds the result to the designated base address

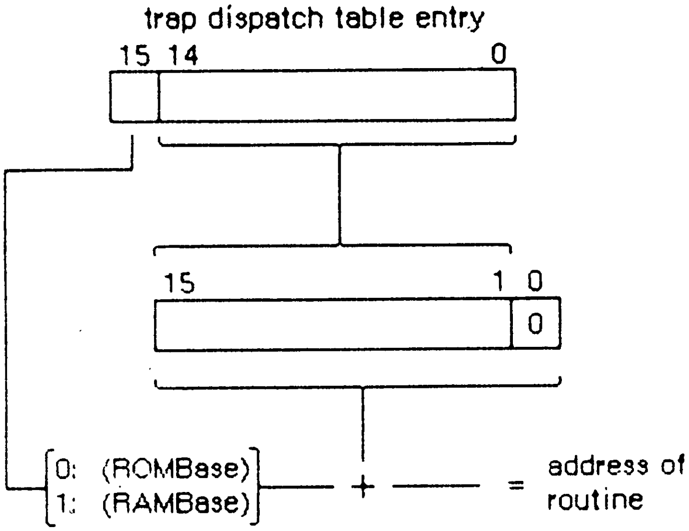


Figure 1. Trap Dispatch Table Entry

Using 15-bit word offsets, the trap 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 slightly more than 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 are used to hold global system data (including the trap 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 trap dispatch table's range. Any such code is inaccessible through the trap dispatch table.

(note)

> This problem is particularly acute on the Macintosh 512K and Macintosh XL. To make sure they lie within range of RAMBase, patches to Toolbox and Operating System routines are typically placed in the system heap rather than the application heap.

THE TRAP MECHANISM

Calls to the Toolbox and Operating System via the trap dispatch table are implemented by means of the MC68000's "1010 emulator" trap. To issue such a call in assembly language, you use one of the *trap macros* defined in the 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 1010); the rest of the word identifies the routine you're calling, along with some additional information pertaining to the call.

(note)

> A list of all Macintosh trap words is given in the appendix of the Operating System Utilities manual.

Instruction words beginning with $A or $F ("A-line" or "F-line" instructions) 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. A-line instructions are reserved for use by Apple; on a Macintosh, they provide access to the Toolbox and Operating System routines. Attempting to execute such an 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 trap dispatch table, and jumps to the routine.

(note)

> F-line instructions are reserved by Motorola for use in future processors.

Format of Trap Words

As noted above, a trap word always contains $A in bits 12-15. Bit 11 determines how the remainder of the word will be interpreted; usually it's Ø for Operating System calls and 1 for Toolbox calls, though there are some exceptions.

Figure 2 shows the Toolbox trap word format. Bits Ø-8 form the trap number (an index into the trap dispatch table), identifying the particular routine being called. Bit 9 isn't used. Bit 1Ø is the "auto-pop" bit; this bit is used by language systems that, rather than directly invoke the trap like Lisa Pascal, do a JSR to the trap word followed immediately by a return to the calling routine. In this case, the return addresses for the both the JSR and the trap get pushed onto the stack, in that order. The auto-pop bit causes the trap dispatcher to pop the trap's return address from the stack and return directly to the calling program.

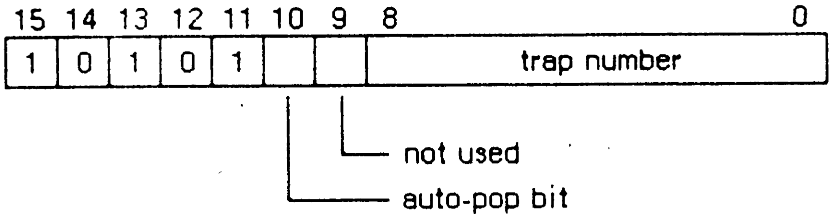


Figure 2. Toolbox Trap Word (Bit 11=1)

For Operating System calls, only the low-order eight bits (bits Ø-7) are used for the trap number (see Figure 3). Thus of the 512 entries in the trap dispatch table, only the first 256 can be used for Operating System traps. Bit 8 of an Operating System 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 routine you're calling, and are covered where relevant in other manuals.

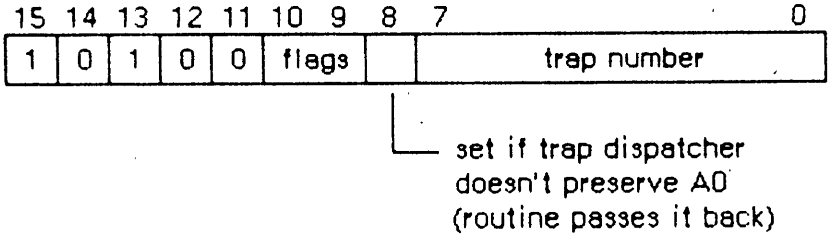

15 14 13 12 11 10 9 8 7 0
1 0 1 0 0
flags
trap number
set if trap dispatcher doesn't preserve A0 (routine passes it back)

Figure 3. Operating System Trap Word (Bit 11=Ø)

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 Toolbox and Operating System documentation. For example, to call the Window Manager routine NewWindow, you would use an instruction with the macro name _NewWindow in the opcode field. There are some exceptions, however, in which the spelling of the macro name differs from the name of the Pascal routine itself; these are noted in the documentation for the individual routines.

(note)

> The reason for the exceptions is that assembler names must be unique to eight characters. Since one character is taken up by the underscore, special macro names must be used for Pascal routines whose names aren't unique to seven characters.

Trap macros for Toolbox calls take no arguments; those for Operating System 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 Routines". 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 Toolbox and Operating System routines fall into two categories: stack-based and register-based. As the terms imply, stack-based routines communicate via the stack, following the same conventions used by the Pascal Compiler for routines written in Lisa Pascal, while register-based routines receive their parameters and return their results in registers. Before calling any Toolbox or Operating System routine, you have to set up the parameters in the way the routine expects.

(note)

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

Stack-Based Routines

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. If the routine you're calling is a function, its result is returned on the stack. The number and types of parameters, and the type of result returned by a function, depend on the routine being called. The number of bytes each parameter or result occupies on the stack depends on its type:

Type of parameter or function result	Size	Contents
INTEGER	2 bytes	Two's complement integer
LONGINT	4 bytes	Two's complement integer
BOOLEAN	2 bytes	Boolean value in bit Ø of high-order byte
CHAR	2 bytes	Extended ASCII code in low-order byte
REAL, DOUBLE, or COMP	4 bytes	Pointer to value converted to EXTENDED
EXTENDED	4 bytes	Pointer to value
STRING[n]	4 bytes	Pointer to string (first byte pointed to is length byte)
SignedByte	2 bytes	Value in low-order byte
Byte	2 bytes	Value in low-order byte
Ptr	4 bytes	Address of data
Handle	4 bytes	Address of master pointer
Record or array	2 or 4 bytes	Contents of structure (padded to word boundary) if <= 4 bytes, otherwise pointer to structure
VAR parameter	4 bytes	Address of variable, regardless of type

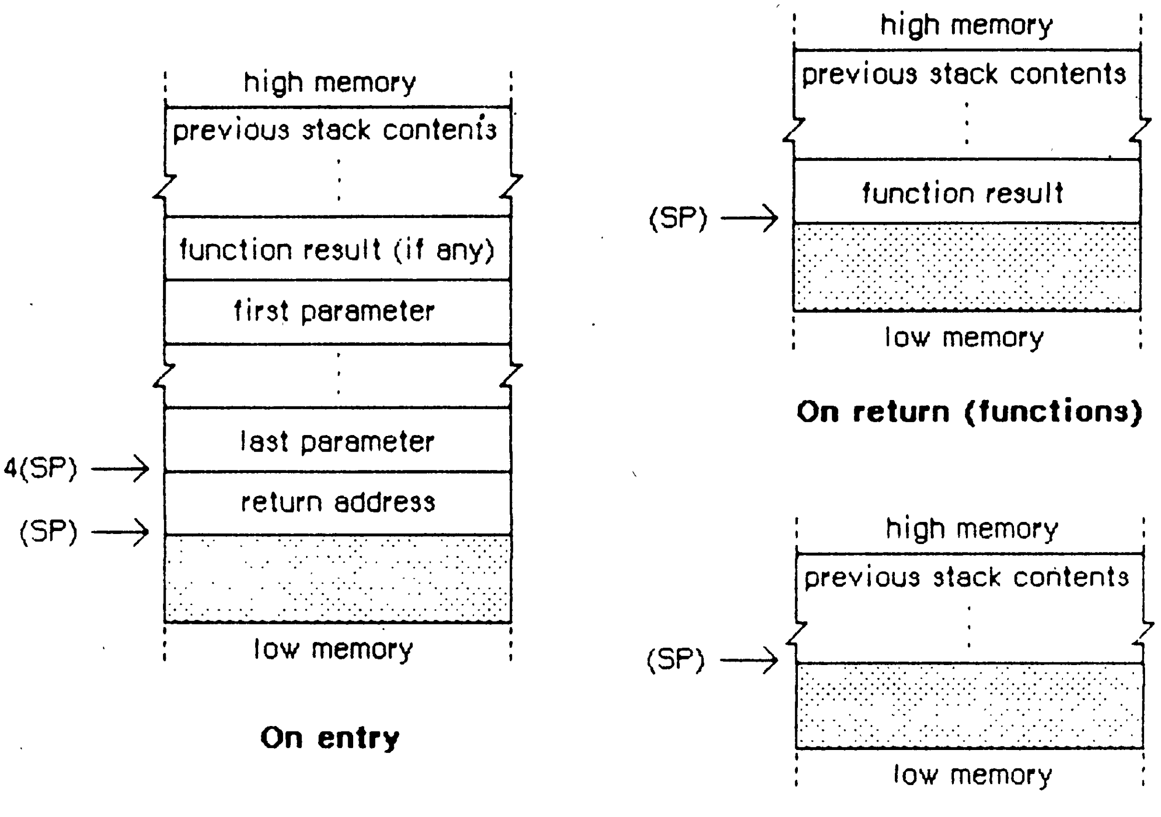
The steps to take to call the routine are as follows:

1. If it's a function, reserve space on the stack for the result.

2. Push the parameters onto the stack in the order they occur in the routine's Pascal definition.

3. 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 4 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 in the space reserved for it; if it's a procedure, it restores the stack to the same state it was in before the call.



On entry

On return (functions)

On return (procedures)

Figure 4. Stack Format for Stack-Based Routines

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
```

Although the MC68ØØØ 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.

Remember that the stack pointer must always be aligned on a word boundary. This is why, for example, a Boolean parameter occupies two bytes; it's actually the Boolean value followed by a byte of padding. Because all Macintosh application code runs in the MC68ØØØ's supervisor mode, an odd stack pointer will cause a "double bus fault": a catastrophic system failure that causes the system to restart.

To keep the stack pointer properly aligned, the MC68ØØØ automatically adjusts the pointer by 2 instead of 1 when you move a byte-length value to or from the stack. This happens only when all of the following three conditions are met:

- A 1-byte value is being transferred.
- Either the source or the destination is specified by predecrement or postincrement addressing.
- The register being decremented or incremented is the stack pointer (A7).

An extra, unused byte will automatically be added in the low-order byte to keep the stack pointer even. (Note that if you need to move a character to or from the stack, you must explicitly use a full word of data, with the character in the low-order byte.)

(warning)

> If you use any other method to manipulate the stack pointer, it's your responsibility to make sure the pointer stays properly aligned.

(note)

> Some Toolbox and Operating System routines accept the address of one of your own routines as a parameter, and call that routine under certain circumstances. In these cases, you must set up your routine to be stack-based.

Register-Based Routines

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 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 AØ, the address of the first destination location in A1, and the number of bytes to be copied in DØ. So you might write something like

```
LEA       src(A5),AØ        ;source address in AØ
LEA       dest(A5),A1       ;destination address in A1
MOVEQ     #2Ø,DØ            ;byte count in DØ
_BlockMove                  ;trap to routine
```

Macro Arguments

The following information applies to the Lisa Assembler. If you're using some other assembler, you should check its documentation to find out whether this information applies.

Many register-based routines expect to find an address of some sort in register AØ. You can 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 AØ. 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 AØ
```

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 AØ will be omitted from the macro expansion. Notice that AØ is loaded with the address denoted by the argument, not the contents of that address.

(note)

> You can use any of the MC68ØØØ'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.

Result Codes

Many register-based routines return a result code in the low-order word of register DØ to report successful completion or failure due to some error condition. A result code of Ø always indicates that the routine was completed successfully. Just before returning from a register-based call, the trap dispatcher tests the low-order word 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
        BEQ     NoError         ;branch if no error
        . . .                   ;handle error
```

(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.

Register-Saving Conventions

All Toolbox and Operating System routines preserve the contents of all registers except AØ, A1, and DØ-D2 (and of course A7, which is the stack pointer). In addition, for register-based routines, the trap dispatcher saves registers A1, D1, and D2 before dispatching to the routine and restores them before returning to the calling program. 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 for a register-based trap depends on the setting of bit 8 of the trap word: If this bit is Ø, the trap dispatcher saves and restores AØ; if it's 1, the routine passes back AØ 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.

Stack-based traps preserve only registers A2-A6 and 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.

(note)

> Any routine in your application that may be called as the result of a Toolbox or Operating System call shouldn't rely on the value of any register except A5, which shouldn't change.

Pascal Interface to the Toolbox and Operating System

When you call a register-based Toolbox or Operating System routine from Pascal, you're actually calling an interface routine that fetches the parameters from the stack where the Pascal calling program left them, puts them in the registers where the routine expects them, and 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 may also move the result code to a global variable, where it can later be accessed.)

For stack-based calls, there's no interface routine; the trap word is inserted directly into the compiled code.

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 a program such as the Linker. For convenience in this discussion, such separately compiled or assembled portions of a program will be called "modules". 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, and must be resolved by a program such as the Linker that 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. For more information, and for details about the actual process of linking the modules together, see the documentation for the development system you're using.

In addition to being able to call your own Pascal routines from assembly language, you can call certain routines in the Toolbox and Operating System that were created expressly for Lisa Pascal programmers and aren't part of the Macintosh ROM. (These routines may also be available to users of other development systems, depending on how the interfaces have been set up on those systems.) They're marked with the notation

[Not in ROM]

*** previously [Pascal only] or [No trap macro] *** in the documentation. There are no trap macros for these routines (though they may call other routines for which there are trap macros). Some of them were created just to allow Pascal programmers access to assembly-language information, and so won't be useful to assembly-language programmers. Others, however, contain code that's executed before a trap macro is invoked, and you may want to perform the operations they provide.

All calls from one language to the other, in either direction, must obey Pascal's stack-based calling conventions (see "Stack-Based Routines", above). To call your own Pascal routine from assembly language, or one of the Toolbox or Operating System routines that aren't in ROM, 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 MC68000'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 it. 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 #0.

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 5). 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.

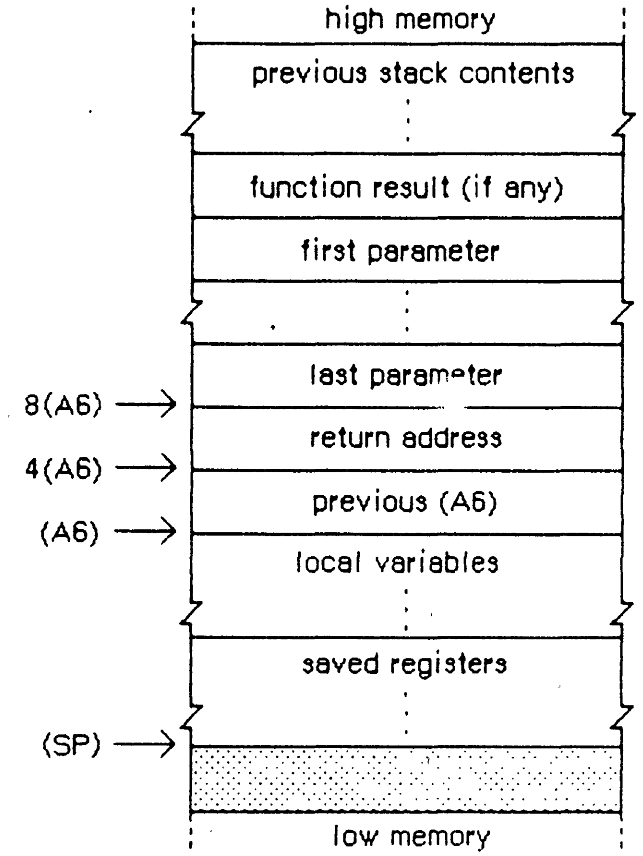


Figure 5. Frame Pointer

Since the saved contents of the frame pointer register occupy a long word (four 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 Routines". 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 it 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 instruction. But if there are any parameters, it's the routine's responsibility to pop 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, and then exit with an indirect jump through the register.

Remember that any routine called from Pascal must observe Pascal register conventions and preserve registers A2-A6 and D3-D7. This is usually done by saving the registers that the routine will be using on the stack with a MOVEM instruction, and then restoring them before returning. Any routine you write that will be accessed via the trap mechanism--for instance, your own version of a Toolbox or Operating System routine that you've patched into the trap 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 an
                                         ; available register
          ADD.W    #pp,SP                ;pop 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)

> When the Segment Loader starts up an application, it 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 global variable CurrentA5.

SUMMARY

Variables

OneOne	$ØØØ1ØØØ1
MinusOne	$FFFFFFFF
Lo3Bytes	$ØØFFFFFF
Scratch2Ø	2Ø-byte scratch area
Scratch8	8-byte scratch area
ToolScratch	8-byte scratch area
ApplScratch	12-byte scratch area reserved for use by applications
CurrentA5	Correct value of A5 (long)

GLOSSARY

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.

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

parameter block: Memory space used to transfer information between applications and certain Operating System routines.

register-based routine: A Toolbox or Operating System routine that receives its parameters and returns its results, if any, in registers.

stack-based routine: A Toolbox or Operating System routine that receives its parameters and returns its results, if any, on the stack.

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

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

trap dispatcher: The part of the Operating System that examines a trap word to determine what operation it stands for, looks up the address of the corresponding routine in the trap dispatch table, and jumps to the routine.

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

trap number: The identifying number of a Toolbox or Operating System routine; an index into the trap dispatch table.

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

unimplemented instruction: An instruction word that doesn't correspond to any valid machine-language instruction but instead causes a trap.

MACINTOSH USER EDUCATION

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

See Also: Macintosh User Interface Guidelines
Inside Macintosh: A Road Map
Macintosh Memory Management: An Introduction
Programming Macintosh Applications in Assembly Language
QuickDraw: A Programmer's Guide
The Control Manager: A Programmer's Guide
The Menu Manager: A Programmer's Guide
The Memory Manager: A Programmer's Guide
The File 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
	Fourth Draft	Caroline Rose & Bob Anders	11/28/84

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.

Summary of significant changes and additions since the last draft:

- A detailed discussion of the specification of resource ID numbers has been added (page 9).

- The concept of "system references" has been moved from the discussion of resource references (page 11) to a separate section (page 37). Since the Finder does not recognize these references to system resources, they aren't particularly useful and have been moved to a section which is essentially "of historical interest only". For this reason, "local references" are now simply called "resource references".

- SizeResource returns a long integer rather than an integer (page 25).

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 the comprehensive Inside Macintosh manual. *** 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 parts of the Toolbox, 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 routine directly.

Like all Toolbox documentation, this manual assumes you're familiar with Lisa Pascal and the information in the following manuals:

- Inside Macintosh: A Road Map

- Macintosh User Interface Guidelines

- Macintosh Memory Management: An Introduction

- Programming Macintosh Applications in Assembly Language, if you're using assembly language

Familiarity with Macintosh files, as described in the File Manager manual, is optional. It's useful if you want a complete understanding of the internal structure of a resource file, but 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 part of the Toolbox that deals with a particular type of resource will tell you what you need to know for that resource.

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 parts of the Toolbox 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).

Resources are grouped logically by function into resource types. You refer to a resource by passing the Resource Manager a resource specification, which consists of the resource type and either an ID number or a name. Any resource type is valid, whether one of those recognized by the Toolbox as referring to standard Macintosh resources (such as 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.

(note)

> The Resource Manager knows nothing about the formats of the individual types of resources. Only the routines in the other parts of the Toolbox 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.

Resource files are not limited to applications; anything stored in a file can have its own resources. For instance, 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.

(note)

> 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).

A number of resource files may be open at one time; the Resource Manager by default 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. 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.)

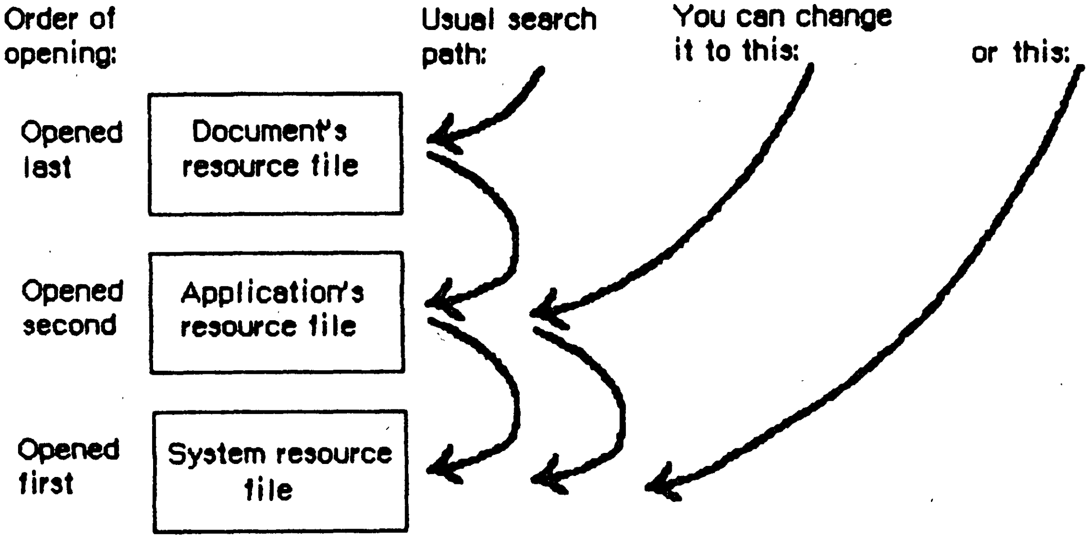


Figure 1. Resource File Searching

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 itself. The code may be divided into different segments, each of which is a resource; this allows various parts of the program to be loaded and purged dynamically. Information is stored in the resource fork via the Resource Manager. The data fork of an application file can contain anything an application wants to store there. Information is stored in the data fork via the File Manager.

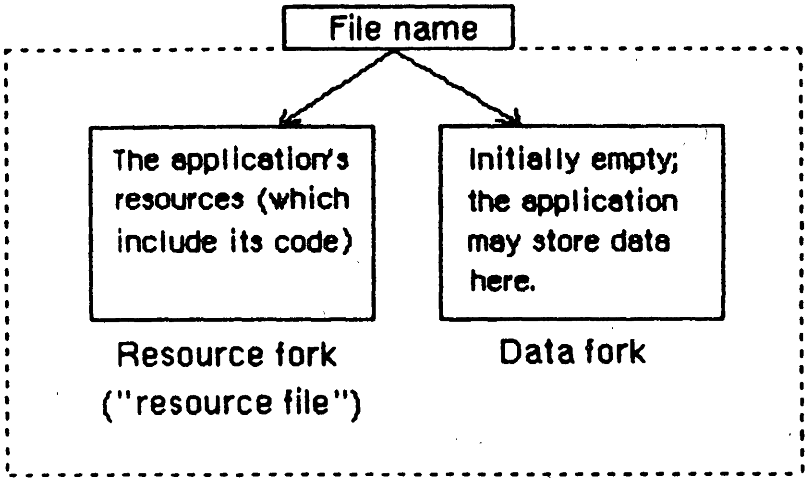

File name
The application's resources (which include its code)
Initially empty; the application may store data here.
Resource fork ("resource file")
Data fork

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 "patches" to the routines in the Macintosh ROM. 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.

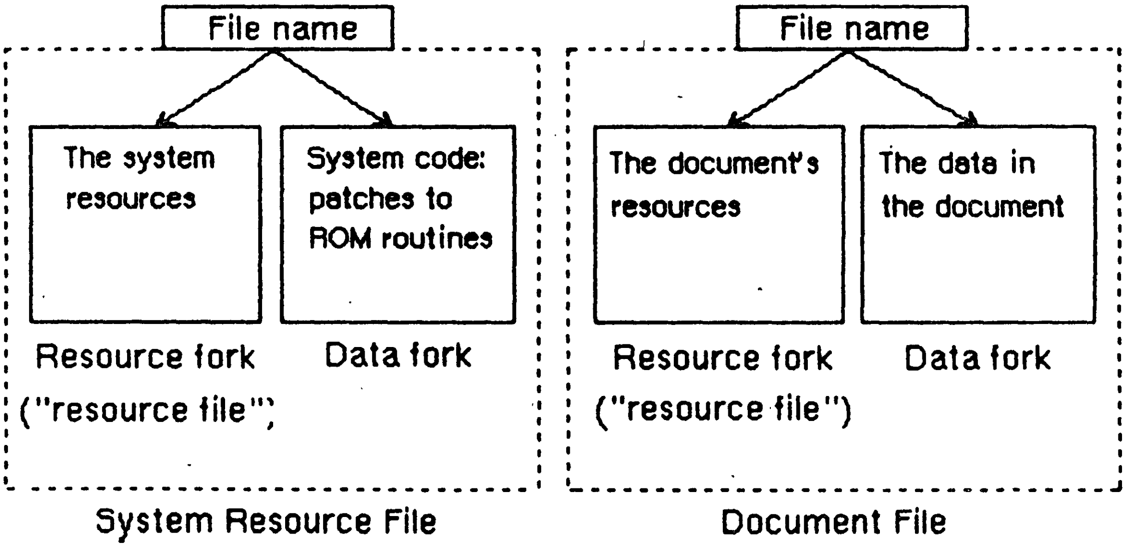


Figure 3. Other Files

To open a resource file, the Resource Manager calls the appropriate File Manager routine and returns the reference number it gets from the File Manager. This is a number greater than Ø by which you can refer to the file when calling other Resource Manager routines.

(note)

> This reference number is actually the path reference number, as described in the File Manager manual.

Most of the Resource Manager routines don't require the resource file's reference number as a parameter. Rather, 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 contains an entry for each resource that provides the location of its resource data. Each entry in the map either gives the offset of the resource data in the file or contains a handle to the data if it's in memory. The resource map is like the index of a book; the Resource Manager looks in it for the resource you specify and determines where its resource data is located.

The resource map is read into memory when the file is opened and remains there until the file is closed. Although for simplicity we say that 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. When read in, resource data is stored in a relocatable block in the heap. Resources are designated in the resource map as being either purgeable or unpurgeable; if purgeable, they may be removed from the heap when space is required by the Memory Manager. Resources consisting of a relatively large amount of data are usually designated as purgeable. Before accessing such a resource through its handle, you ask the Resource Manager to read the resource into memory again if it has been purged.

(note)

> 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 may contain a partial copy of its entry in the file directory, written and used by the Finder, and up to 128 bytes of any data the application wants 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.

Resource Types

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

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

The standard Macintosh resource types are as follows:

Resource type	Meaning
'ALRT'	Alert template
'BNDL'	Bundle
'CDEF'	Control definition function
'CNTL'	Control template
'CODE'	Application code segment
'CURS'	Cursor
'DITL'	Item list in a dialog or alert
'DLOG'	Dialog template
'DRVR'	Desk accessory or other device driver
'DSAT'	System startup alert table
'FKEY'	Command-Shift-number routine
'FONT'	Font
'FREF'	File reference
'FRSV'	Font reserved for system use
'FWID'	Font widths
'ICN#'	Icon list
'ICON'	Icon
'INIT'	Initialization resource
'INTL'	International resource
'KEYC'	Keyboard configuration
'MBAR'	Menu bar
'MDEF'	Menu definition procedure
'MENU'	Menu
'PACK'	Package
'PAT '	Pattern (The space is required.)
'PAT#'	Pattern list
'PDEF'	Printing code
'PICT'	Picture
'PREC'	Print record

'STR '	String (The space is required.)
'STR#'	String list
'WDEF'	Window definition function
'WIND'	Window template

(warning)

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.

Resource ID Numbers

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. If you assign the same resource ID to two resources of the same type, the second assignment of the ID will override the first, thereby making the first resource inaccessible.

(warning)

Certain resources contain the resource IDs of other resources; for instance, a dialog template contains the resource ID of its item list. In order not to duplicate an existing resource ID, a program that copies resources may need to change the resource ID of a resource; such a program may not, however, change the ID where it occurs in other resources. For instance, an item list's resource ID contained in a dialog template may not be changed, even though the actual resource ID of the item list was changed to avoid duplication; this would make it impossible for the template to access the item list. Be sure to verify, and if necessary, correct, the IDs contained within such resources. (For related information, see the section "Resource IDs of Owned Resources" below.)

By convention, the ID numbers are divided into the following ranges:

Range	Description
-32768 through -16385	Reserved; do not use
-16384 through -1	Used for system resources owned by other system resources (explained below)
Ø through 127	Used for other system resources
128 through 32767	Available for your use in whatever way you wish

(note)

The manuals that describe the different types of resources in detail give information about resource types that may be more restrictive about the allowable range for their resource IDs. A device driver, for instance, can't have a resource ID greater than 31.

Resource IDs of Owned Resources

This section is intended for advanced programmers who are involved in writing their own desk accessories (or other drivers), or special types of windows, controls, and menus. It's also useful in understanding the way that resource-copying programs recognize resources that are associated with each other.

Certain types of system resources may have resources of their own in the system resource file; the "owning" resource consists of code that reads the "owned" resource into memory. For example, a desk accessory might have its own pattern and string resources. A special numbering convention is used to associate owned system resources with the resources they belong to. This enables resource-copying programs to recognize which additional resources need to be copied along with an owning resource. An owned system resource has the ID illustrated in Figure 4.

15	14	13 11	10 5	4 0
1	1	type bits	ID of owning resource	variable

Figure 4. Resource ID of an Owned System Resource

Bits 14 and 15 are always 1. Bits 11 through 13 specify the type of the owning resource, as follows:

Type bits	Type
ØØØ	'DRVR'
ØØ1	'WDEF'
Ø1Ø	'MDEF'
Ø11	'CDEF'
1ØØ	'PDEF'
1Ø1	'PACK'
11Ø	Reserved for future use
111	Reserved for future use

Bits 5 through 1Ø contain the resource ID of the owning resource (limited to Ø through 63). Bits Ø through 4 contain any desired value (Ø through 31).

Certain types of resources can't be owned, because their IDs don't conform to the special numbering convention described above. For instance, the resource ID for a resource of type 'WDEF can't be more than 12 bits long (as described in the Window Manager manual). Fonts are also an exception because their IDs include the font size. The manuals describing the different types of resources provide detailed information about such restrictions.

An owned resource may itself contain the ID of a resource associated with it. For instance, a dialog template owned by a desk accessory contains the resource ID of its item list. Though the item list is associated with the dialog template, it's actually owned (indirectly) by the desk accessory. The resource ID of the item list should conform to the same special convention as the ID of the template. For example, if the resource ID of the desk accessory is 17, the IDs of both the template and the item list should contain the value 17 in bits 5 through 1Ø.

As mentioned above, a program that copies resources may need to change the resource ID of a resource in order not to duplicate an existing resource ID. Bits 5 through 1Ø of resources owned, directly or indirectly, by the copied resource will also be changed when those resources are copied. For instance, in the above example, if the desk accessory must be given a new ID, bits 5 through 1Ø of both the template and the item list will also be changed.

(warning)

> Remember that while the ID of an owned resource may be changed by a resource-copying program, the ID may not be changed where it appears in other resources (such as an item list's ID contained in a dialog template).

Resource Names

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 ignores case (but does not ignore diacritical marks in foreign names).

RESOURCE REFERENCES

The entries in the resource map that identify and locate the resources in a resource file are known as resource references. Using the analogy of an index of a book, resource references are like the individual entries in the index.

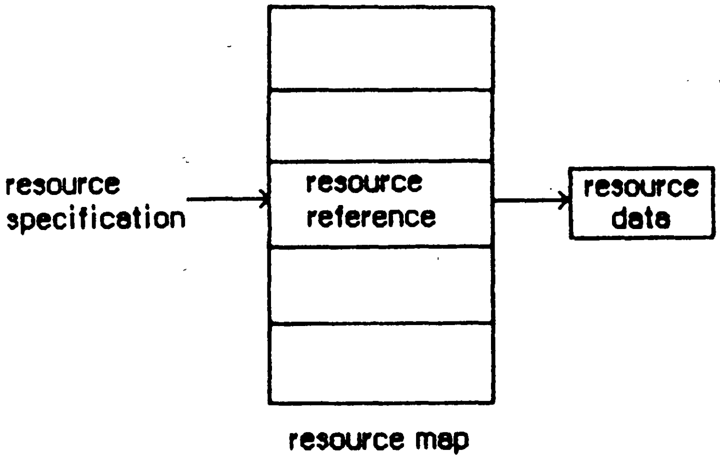


Figure 5. Resource References in Resource Maps

Every resource reference includes the type, ID number, and optional name of the resource. 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 of the current resource file. 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 finds a reference matching the specification, 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 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 contains 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 6.

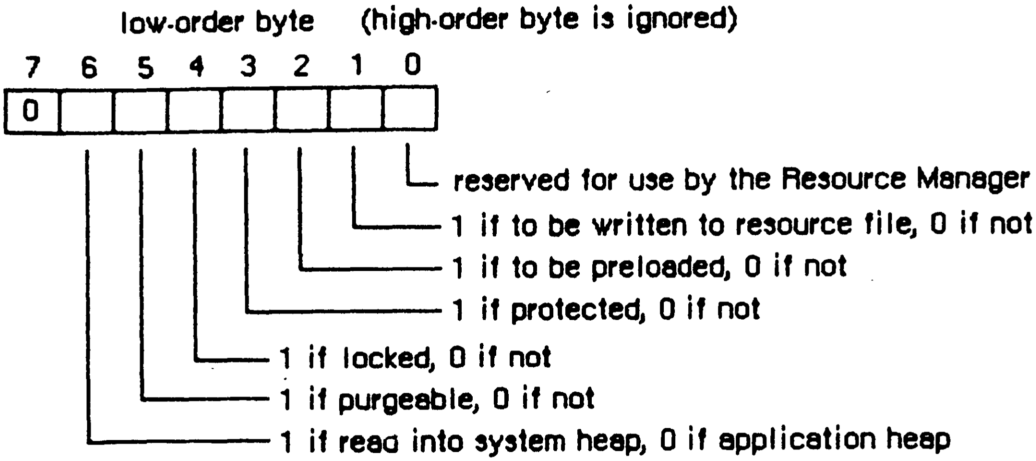

low-order byte
(high-order byte is ignored)
7 6 5 4 3 2 1 0
0
reserved for use by the Resource Manager
1 if to be written to resource file, 0 if not
1 if to be preloaded, 0 if not
1 if protected, 0 if not
1 if locked, 0 if not
1 if purgeable, 0 if not
1 if reao into system heap, 0 if application heap

Figure 6. Resource Attributes

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

```
CONST 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}
```

(warning)

> Your application should not change the setting of bit Ø or 7, 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; it should not be set for your application's resources. If a system 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 change the ID number or name of the resource, modify its contents, or remove the resource from the resource file. 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; it must be Ø in the resource file. It tells the Resource Manager whether this resource has been changed.

USING THE RESOURCE MANAGER

The Resource Manager is initialized 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.

(note)

> These are the only routines you need to know about to use the Resource Manager indirectly through other parts of the Toolbox; 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 (a pointer to a NIL master pointer). You can pass the empty handle to routines that operate only on the resource map (such as the routine that sets resource attributes), since the handle is enough for the Resource Manager to tell what resource you're referring to. Should you later want to access the resource data, you can read it into memory with the LoadResource procedure. Before calling any of the above routines that read the resource data into memory, it's a good idea to call SizeResource to see how much space is needed.

Normally the Resource Manager starts looking for a resource in the most recently opened resource file, and searches other open resource files in the reverse of the order that they were opened. In some situations, you may want to change which file is searched first. You can do this with the UseResFile procedure. One such situation might be when you want a resource to be read from the same file as another resource; in this case, you can find out which resource file the other resource was read from by calling the HomeResFile function.

Once you have a handle to a resource, you can call GetResInfo or GetResAttrs to get the information that's stored for that resource in the resource map, or you can access the resource data through the handle. (If the resource was designated as purgeable, first call LoadResource to ensure that the data is in memory.)

Usually you'll just read resources from previously created resource files with the routines described above. You may, however, want to modify existing resources or even create your own resource file. To create your own resource file, call CreateResFile (followed by OpenResFile to open it). The AddResource procedure lets you add resources to a resource file; to be sure a new resource won't override an existing one, you can call the UniqueID function to get an ID number for it. To make a copy of an existing resource, call DetachResource followed by AddResource (with a new resource ID). There are a number of procedures for modifying existing resources:

- To remove a resource, call RmveResource.

- If you've changed the resource data for a resource and want the changed data to be written to the resource file, call ChangedResource; it signals the Resource Manager to write the data out when the resource file is later updated.

- To change the information stored for a resource in the resource map, call SetResInfo or SetResAttrs. If you want the change to be written to the resource file, call ChangedResource. (Remember that ChangedResource will also cause the resource data itself to be written out.)

These procedures for adding and modifying resources change only the resource map in memory. The changes are written to the resource file when the application terminates (at which time all resource files other than the system resource file are updated and closed) or when one of the following routines is called:

- 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

Assembly-language note: Except for LoadResource, all Resource Manager routines preserve all registers except AØ and DØ. LoadResource preserves AØ and DØ as well.

Initialization

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.

Assembly-language note: The name of the system resource file is stored in the global variable SysResName; the reference number for the file is stored in the global variable SysMap. A handle to the resource map of the system resource file is stored in the variable SysMapHndl.

(note)

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 application heap (that is, whose resSysHeap attribute is Ø), it replaces the handle to that resource in the resource map with NIL.

This lets the Resource Manager know that the resource will have to be read in again (since the previous application heap is no longer around).

Opening and Closing Resource Files

When calling the CreateResFile or OpenResFile routines, described below, you specify a resource file by its file name; the routines assume that the file has a version number of Ø and is on the default volume. (Version numbers and volumes are described in the File Manager manual.)

PROCEDURE CreateResFile (fileName: Str255);

CreateResFile creates a resource file containing no resource data or copy of the file's directory entry. If there's no file at all with the given name, it also creates an empty data fork for the file. If there's already a resource file with the given name (that is, a resource fork that isn't empty), CreateResFile will do nothing and the ResError function will return an appropriate Operating System result code.

(note)

> Before you can work with the resource file, you need to open it with OpenResFile.

FUNCTION OpenResFile (fileName: Str255) : INTEGER;

OpenResFile opens the resource file having the given name and makes it the current resource file. It reads the resource map from the file into memory and returns a reference number for the file. It also reads in every resource whose resPreload attribute is set. If the resource file is already open, it doesn't make it the current resource file; it simply returns the reference number.

(note)

> You don't have to call OpenResFile to open the system resource file or the application's resource file, because they're opened when the system and the application start up, respectively. To get the reference number of the application's resource file, you can call CurResFile after the application starts up (before you open any other resource file).

If the file can't be opened, OpenResFile will return -1 and the ResError function will return an appropriate Operating System result code. For example, an error occurs if there's no resource file with the given name.

Assembly-language note: A handle to the resource map of the most recently opened resource file is stored in the global variable TopMapHndl.

PROCEDURE CloseResFile (refNum: INTEGER);

Given the reference number of a resource file, CloseResFile does the following:

- updates the resource file by calling the UpdateResFile procedure
- for each resource in the resource file, releases the memory it occupies by calling the ReleaseResource procedure
- releases 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 result 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 returns a result code identifying the error, if any. If no error occurred, it returns the result code

```
CONST  noErr = Ø;    {no error}
```

If an error occurred at the Operating System level, it returns an Operating System result code, such as the File Manager "disk I/O" error or the Memory Manager "out of memory" error. (See the File Manager and Memory Manager manuals for a list of the result codes.) If an error happened at the Resource Manager level, ResError returns one of the following result codes:

```
CONST resNotFound  = -192;   {resource not found}
      resFNotFound = -193;   {resource file not found}
      addResFailed = -194;   {AddResource failed}
      rmvResFailed = -196;   {RmveResource 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.

Assembly-language note: The current value of ResError is stored in the global variable ResErr. In addition, you can specify a procedure to be called whenever there's an error by storing a pointer to the procedure in the global variable ResErrProc (which is normally NIL). Before returning a result code other than noErr, the ResError function places that result code in register DØ and calls your procedure.

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.

(note)

If the system resource file is the current resource file, CurResFile returns the actual reference number of the system reference file (found in the global variable SysMap). You needn't worry about this number being used (instead of Ø) in the routines that require a reference number; these routines recognize both Ø and the actual reference number as referring to the system resource file.

Assembly-language note: The reference number of the current resource file is stored in 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 result 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 result code resFNotFound. A refNum of Ø represents the system resource file.

Open resource files are arranged as a linked list; the most recently opened file is at the end of the list and is the first one to be searched. UseResFile lets you start the search with a file opened earlier; the file(s) following it on the list are then left out of the search process. This is best understood with an example. Assume there are four open resource files (RØ through R3); the search order is R3, R2, R1, RØ. If you call UseResFile(R2), the search order becomes R2, R1, RØ; R3 is no longer searched. If you then open a fifth resource file (R4), it's added to the end of the list and the search order becomes R4, R3, R2, R1, RØ.

This procedure is useful 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 leave the application resource file out of the search, causing only the system resource file to be searched.

(warning)

> Early versions of some desk accessories may, upon closing, always set the current resource file to the one opened just prior to the accessory, ignoring any additional resource files that may have been opened while the accessory was in use. To be safe, whenever desk accessories may have been in use, call UseResFile to ensure access to resource files opened after accessories.

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.

(warning)

> If you call SetResLoad(FALSE), be sure to restore the normal state as soon as possible, because other parts of the Toolbox that call the Resource Manager rely on it.

Assembly-language note: The current SetResLoad state is stored in 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).

(warning)

> The handle returned will be an empty handle if you've called SetResLoad(FALSE) (and the data isn't already in memory). The handle 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.

(note)

> 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 and the ResError function will return the result code resNotFound. GetIndResource also returns NIL if the resource is to be read into memory but won't fit; in this case, ResError will return an appropriate Operating System result 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 warning 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 and the ResError function will return the result code resNotFound. GetResource also returns NIL if the resource is to be read into memory but won't fit; in this case, ResError will return an appropriate Operating System result 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 result 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 if 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.

Assembly-language note: LoadResource preserves **all** registers.

PROCEDURE ReleaseResource (theResource: Handle);

Given a handle to a resource, ReleaseResource releases the memory occupied by the resource data, if any, and replaces the handle to that resource in the resource map with NIL. (See Figure 7.) 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.

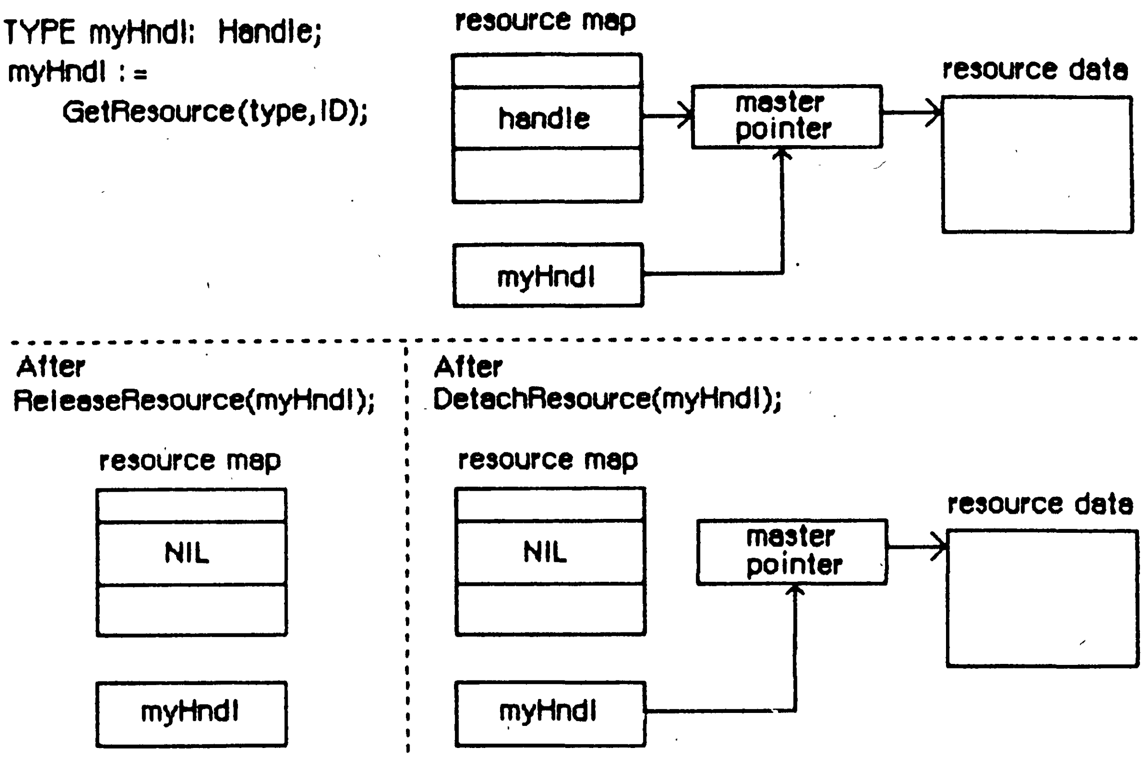


Figure 7. ReleaseResource and DetachResource

If the given handle isn't a handle to a resource, ReleaseResource will do nothing and the ResError function will return the result 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 7 above.) 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. DetachResource is also useful in the unusual case that you don't want a resource to be released when a resource file is closed. To copy a resource, you can call DetachResource followed by AddResource (with a new resource ID).

If the given handle isn't a handle to a resource, DetachResource will do nothing and the ResError function will return the result 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 you won't duplicate a resource ID and override an existing resource.

(warning)
It's possible that UniqueID will return an ID in the range reserved for system resources (Ø to 127). You should check that the ID returned is greater than 127; if it isn't, call UniqueID again.

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 given handle isn't a handle to a resource, GetResInfo will do nothing and the ResError function will return the result 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 above under "Resource References".) If the given handle isn't a handle to a resource, GetResAttrs will do nothing and the ResError function will return the result code resNotFound.

FUNCTION SizeResource (the Resource: Handle) : LONGINT;

Given a handle to a resource, SizeResource returns the size in bytes of the resource in 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 result code resNotFound. It's a good idea to call SizeResource and ensure that sufficient space is available before reading a resource into memory.

Assembly-language note: The macro you invoke to call SizeResource from assembly language is named _SizeRsrc.

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 changes the ID number and name of the resource to the given ID number and name.

> Assembly-language note: If you pass NIL for the name parameter, the name will not be changed.

(warning)
It's a dangerous practice to change the ID number and name of a system resource, because other applications may already access the resource and may no longer work properly.

The change will be written to the resource file when the file is updated if you follow SetResInfo with a call to ChangedResource.

(warning)
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, 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 result 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 result 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 above under "Resource References".) The resProtected attribute takes effect immediately; the others take effect the next time the resource is read in.

(warning)

> 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 result 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

(warning)

> 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 procedures 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 result code resNotFound.

- Sufficient space for the modified resource file can't be reserved on the disk. ResError will return an appropriate Operating System result code.

(warning)

> Be aware that ChangedResource (and not WriteResource) checks to see if there's sufficient disk space to write out the modified file; if there isn't enough space, the resChanged attribute won't be set. This means that when WriteResource is called, it won't know that the resource file has been changed; it won't write out the modified file and no error will be returned. For this reason, always check to see that ChangedResource returns noErr.

```
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 resource 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 result 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 result code.

(warning)

> AddResource doesn't verify whether the resource ID you've passed is already assigned to another resource of the same type; be sure to call UniqueID before adding a resource.

PROCEDURE RmveResource (theResource: Handle);

Given a handle to a resource in the current resource file, RmveResource removes the resource reference to the resource. The resource data will be removed from the resource file when the file is updated.

(note)

> RmveResource doesn't release the memory occupied by the resource data; to do that, call the Memory Manager procedure DisposHandle 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 result code rmvResFailed.

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 (and the call was successful); if you did, the resource data will be written out with WriteResource.

- Compacts the resource file, closing up any empty space created when a resource was removed or 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; the space occupied by the original is then compacted.) 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. 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 result 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 successfully), writes its resource data to the resource file and clears its resChanged attribute.

(warning)

> Be aware that ChangedResource (and not WriteResource) determines if sufficient disk space is available to write out the modified file; if there isn't it will clear the resChanged attribute and WriteResource will be unaware of the modifications. For this reason, always verify that ChangedResource returns noErr.

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 result 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.

(warning)

> The maximum size for resources to be written to a resource file is 32K bytes.

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; for details, refer to the section "Memory Manager Data Structures" in the Memory Manager manual.

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 lowerder 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;   {set if resource file is read-only}
      mapCompact  = 64;    {set to compact file on update}
      mapChanged  = 32;    {set to write map on update}
```

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..

(warning)

> 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.

> Assembly-language note: The current value of the read-only attribute is stored in the global variable ResReadOnly.

The mapCompact attribute causes resource file compaction to occur when the file is updated. It's set by the Resource Manager when a resource 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. 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 result code resFNotFound. A refNum of Ø represents the system resource 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 result code resFNotFound. A refNum of Ø represents the system resource 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 8.)

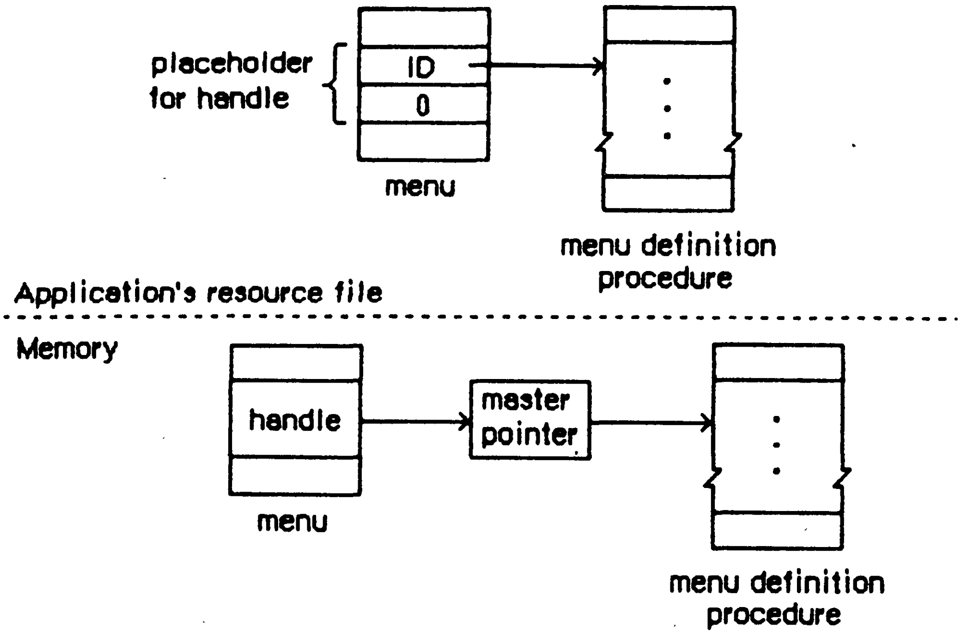

placeholder for handle
ID
0
menu
menu definition procedure
Application's resource file
Memory
handle
master pointer
menu
menu definition procedure

Figure 8. How Resources Point to Resources

(note)

> 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 function 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).

(warning)

> 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

You need to know the exact format of a resource file, described below, only if you're writing a program that will create or modify resource files directly; you don't have to know it to be able to use the Resource Manager routines.

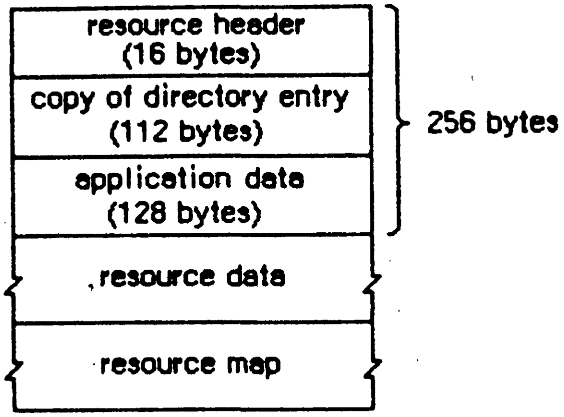


Figure 9. Format of a Resource File

As illustrated in Figure 9, 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

(note)

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 application data

The directory copy is used by the Finder. The application data may be whatever you want.

The resource data follows the application 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 part of the Toolbox 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	Offset from beginning of resouce data to length of data for this resource
4 bytes	Ø (reserved for handle to resource)

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 1Ø shows where the various offsets lead to in a resource file, in general and also specifically for a resource reference.

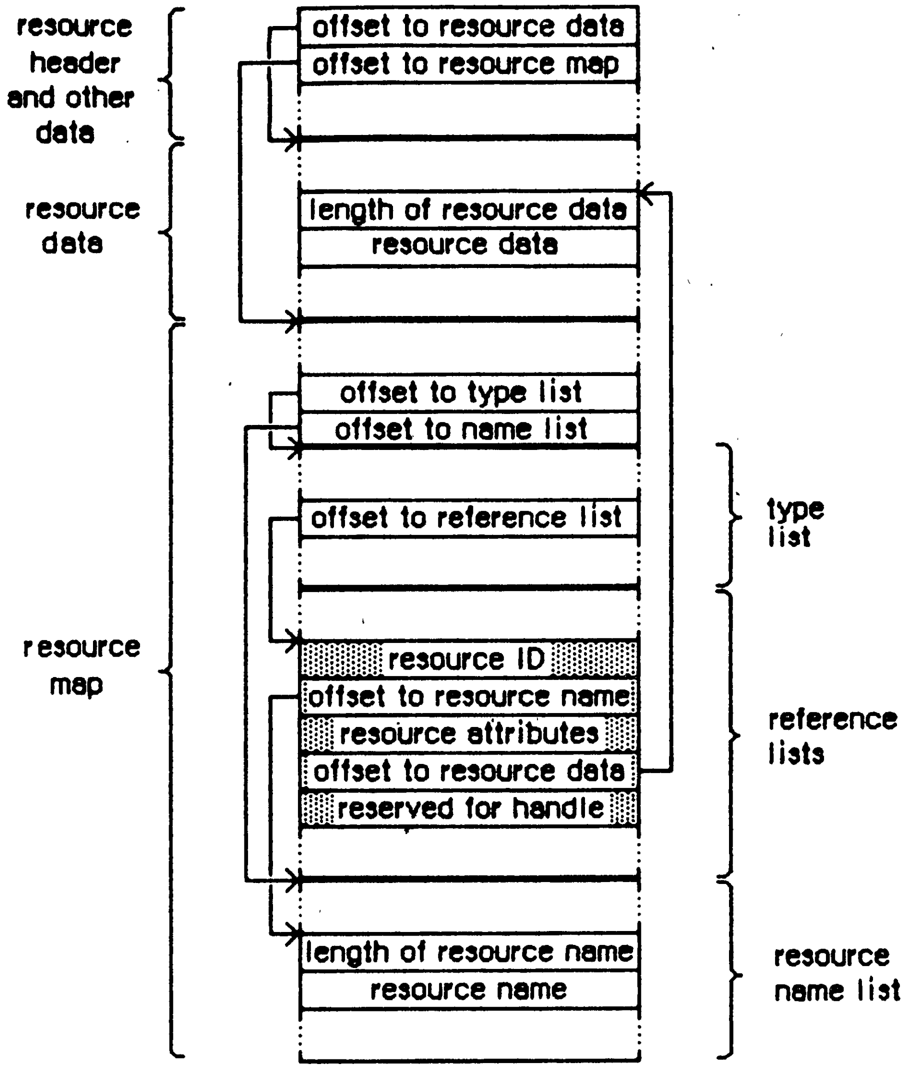


Figure 1Ø. Resource Reference in a Resource File

SYSTEM REFERENCES

This section gives information of historical interest only. It explains another kind of resource reference besides the one explained in the "Resource References" section above. This additional kind of reference, called a system reference, was intended to be used by the Finder, as described below. In fact, the Finder doesn't use system references, so they're not particularly useful.

There are actually two different kinds of resource references, as illustrated in Figure 11:

- Local reference. The term "resource reference", as used earlier in this manual, refers to this type of reference. A local reference is an entry in the resource map that locates the resource data of a resource. If the resource data is already in memory, the local reference provides a handle to the data; otherwise it gives an offset to the resource data in the file.

- System reference. This is also an entry in the resource map but it's a reference to a system resource. It provides 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.

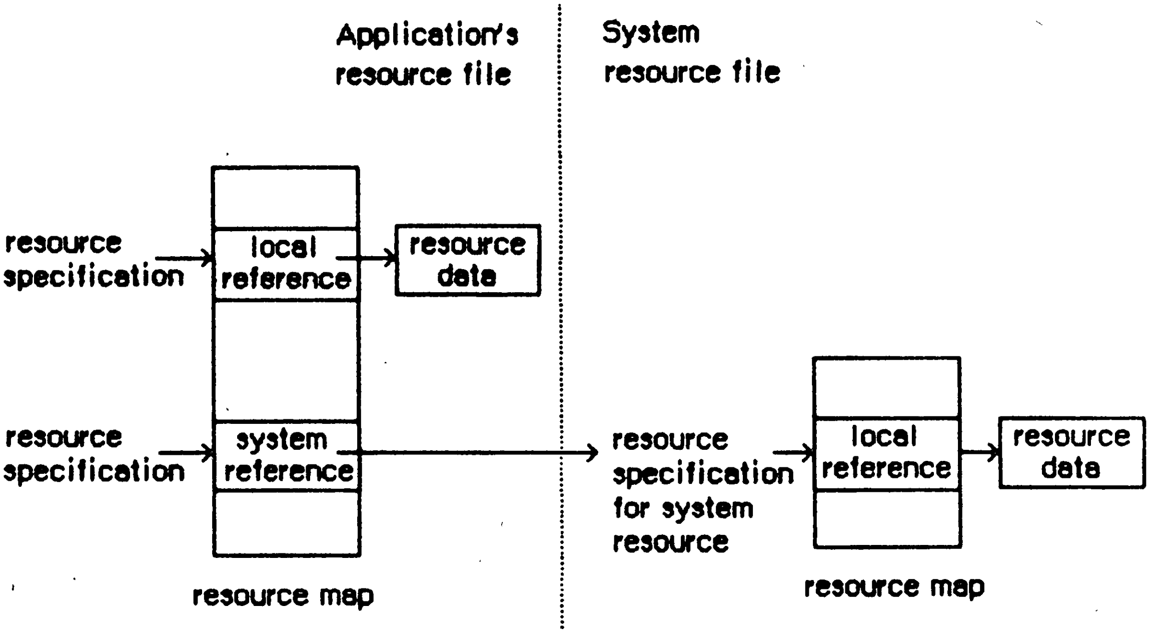


Figure 11. Local and System References

Every resource reference has its own type, ID number, and optional name. In the case of local references, the ID number and name are simply 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 actual resource it refers to in the system resource file.

System references need not be included in an 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. The major reason for having system references was to tell the Finder what system resources an application or document was using. This would ensure that those resources would accompany the application or document should it be copied to a disk having a different system resource file on it. The Finder, however, doesn't recognize system references, which renders them largely ineffectual. (One remaining use for such a reference could be to provide an "alias" for a system resource.)

The remainder of this section explains the use and format of system references, and discusses several routines that work with such references.

Resource Attributes of System References

As stated in the section on resource references, each reference has a set of resource attributes associated with it, and each attribute is specified by a bit in the low-order byte of a word in the resource map. In Figure 6 in that section, bit 7 of the low-order byte is shown as Ø. This bit actually specifies whether or not the reference is a system reference. If you have a system reference in your resource file, this bit should be set. A predefined constant for this attribute is also provided:

```
CONST resSysRef = 128;   {set if system reference}
```

System References in Resource Manager Routines

Some of the previously described Resource Manager routines take special action if the current resource file contains a system reference to the given resource:

- GetResInfo will return the ID number, type, and name of the system reference. The ID number and name may be different from those of the resource itself in the system resource file.

- GetResAttrs will return the attributes of the system reference, which may be different from those of the resource itself in the system resource file.

- SetResInfo will change only the ID number and name of the system reference.

- SetResAttrs will set only the attributes of the system reference.

The following additional procedures can be used to add or remove a system reference.

(note)

> If you've added or removed a system reference, the Resource Manager will write out the entire resource map when it updates the resource file. Also, file compaction will occur during the update if a system reference has been removed.

```
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 result code

  ```
  CONST addRefFailed = -195;   {AddReference failed}
  ```

- 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 result 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.) RmveReference will do nothing and the ResError function will return the result code

```
CONST rmvRefFailed = -197;   {RmveReference failed}
```

if any of the following are true:

- The resProtected attribute for the resource is set.
- There's no system reference to the resource in the current resource file.
- The given handle isn't a handle to a system resource.

Format of System References

In the section "Format of a Resource File", the format of a resource list actually covered only the case of a local reference; the format of a reference list containing either local or system references is outlined below:

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

SUMMARY OF THE RESOURCE MANAGER

Constants

```
CONST { Resource attributes }

      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}

      { Resource Manager result codes }

      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}

      { Resource file attributes }

      mapReadOnly = 128;     {set if file is read-only}
      mapCompact  = 64;      {set to compact file on update}
      mapChanged  = 32;      {set to write map on update}
```

Data Types

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

Routines

Initialization

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

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 UpdateResFile   (refNum: INTEGER);
PROCEDURE WriteResource   (theResource: Handle);
PROCEDURE SetResPurge     (install: BOOLEAN);
```

Advanced Routines

```
FUNCTION  GetResFileAttrs (refNum: INTEGER) : INTEGER;
PROCEDURE SetResFileAttrs (refNum: INTEGER; attrs: INTEGER);
```

Modifying System References

```
PROCEDURE AddReference  (theResource: Handle; theID: INTEGER; name:
                          Str255);
PROCEDURE RmveReference (theResource: Handle);
```

Assembly-Language Information

Constants

```
; Resource attributes

resSysRef        .EQU   7    ;set if system reference
resSysHeap       .EQU   6    ;set if read into system heap
resPurgeable     .EQU   5    ;set if purgeable
resLocked        .EQU   4    ;set if locked
resProtected     .EQU   3    ;set if protected
resPreload       .EQU   2    ;set if to be preloaded
resChanged       .EQU   1    ;set if to be written to resource file

; Resource Manager result codes

resNotFound      .EQU   -192    ;resource not found
resFNotFound     .EQU   -193    ;resource file not found
addResFailed     .EQU   -194    ;AddResource failed
addRefFailed     .EQU   -195    ;AddReference failed
rmvResFailed     .EQU   -196    ;RmveResource failed
rmvRefFailed     .EQU   -197    ;RmveReference failed

; Resource file attributes

mapReadOnly      .EQU    7    ;set if resource file is read-only
mapCompact       .EQU    6    ;set to compact file on update
mapChanged       .EQU    5    ;set to write map on update
```

Variables

Name	Size	Contents
TopMapHndl	4 bytes	Handle to resource map of most recently opened resource file
SysMapHndl	4 bytes	Handle to map of system resource file

SysMap	2 bytes	Reference number of system resource file
CurMap	2 bytes	Reference number of current resource file
ResReadOnly	2 bytes	Current value of mapReadOnly attribute
ResLoad	2 bytes	Current value of SetResLoad
ResErr	2 bytes	Current value of ResError
ResErrProc	4 bytes	Pointer to resource error procedure
SysResName	2Ø bytes	Name of system resource file (beginning with one-byte length)

Special Macro Name

Routine name	Macro name
SizeResource	_SizeRsrc

SUMMARY OF THE RESOURCE FILE FORMAT

(note)

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	Application 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	Offset to length of resource data
	4 bytes	Reserved for handle to resource
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.

data fork: The part of the file that contains data accessed via the File Manager.

empty handle: A pointer to a NIL master pointer.

fork: One of two parts of a file; see data fork and resource fork.

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 fork: The part of the file that contains the resources used by an application (such as menus, fonts, and icons) and also the application code itself; usually accessed via the Resource Manager.

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, an entry that identifies a resource and contains either an offset to its resource data in the resource file or a handle to the data if it's already been read into memory.

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 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.

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.

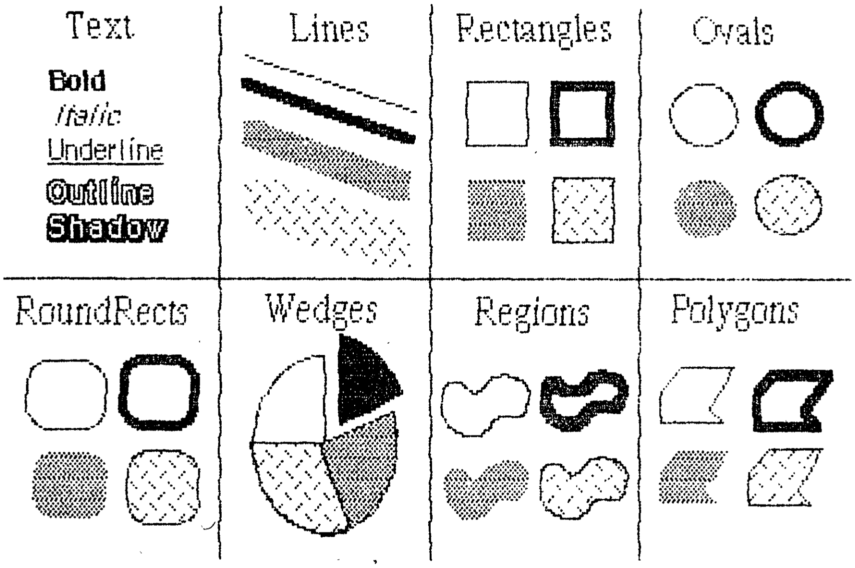


Figure 1. Samples of QuickDraw´s Abilities

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.

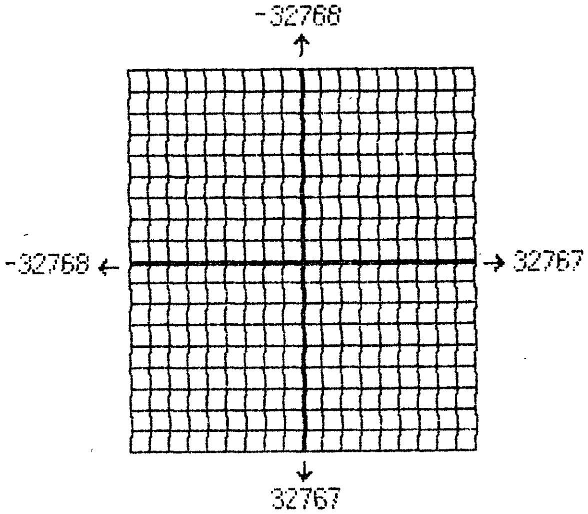


Figure 2. The Coordinate Plane

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).

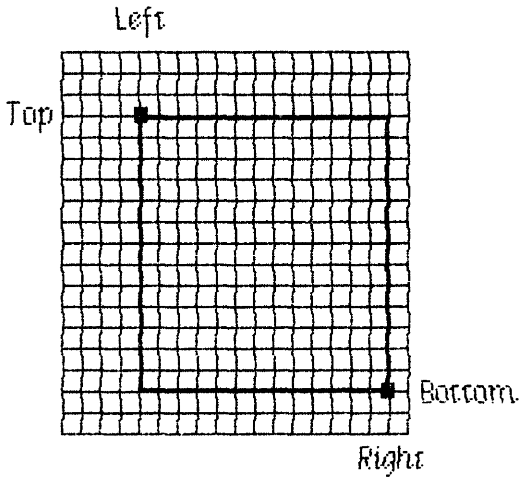


Figure 3. A Rectangle

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.

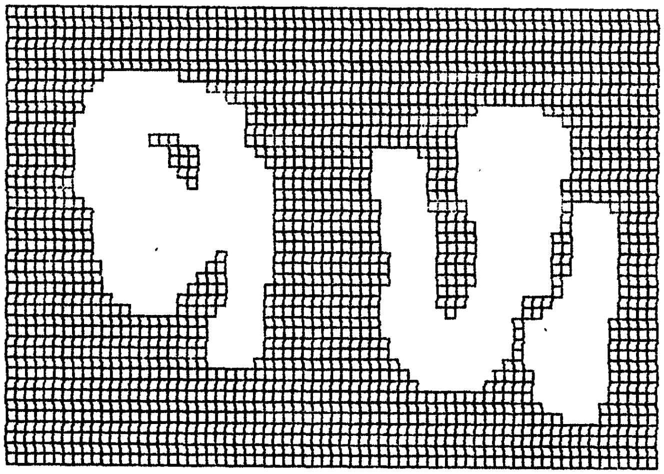


Figure 4. Regions

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 1Ø.

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.

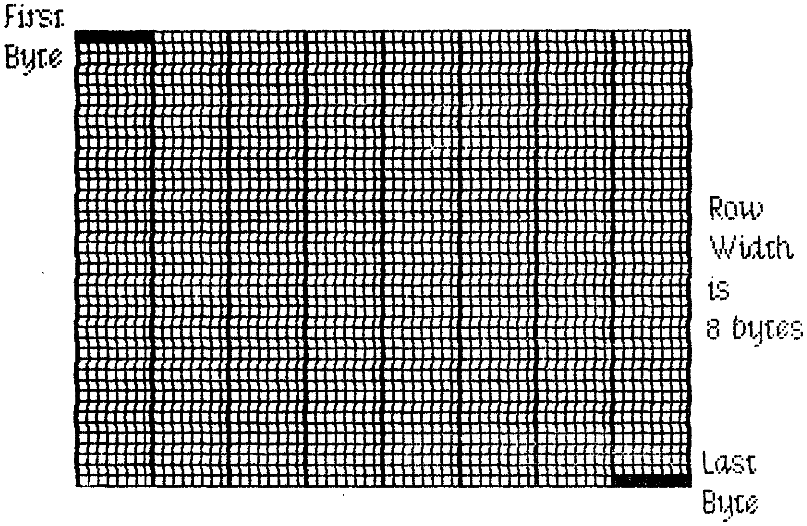


Figure 5. A Bit Image

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;
```

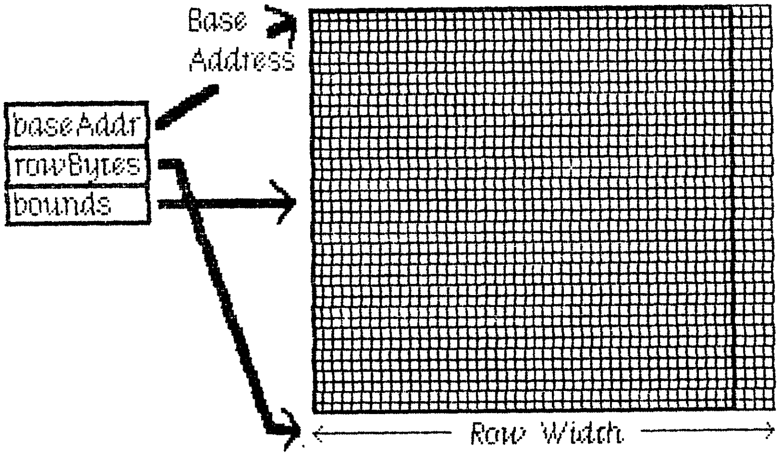


Figure 6. A BitMap

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 (1Ø,-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).

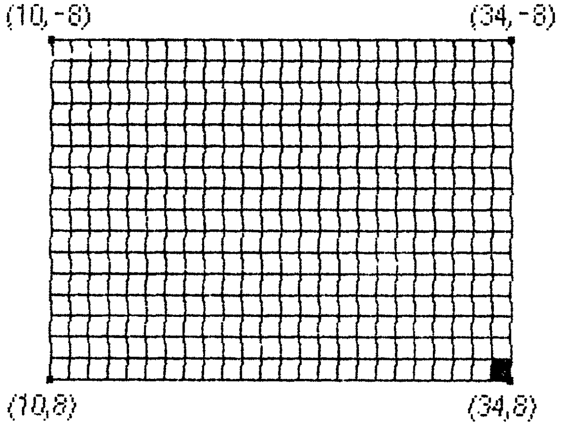


Figure 7. Coordinates and BitMaps

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.

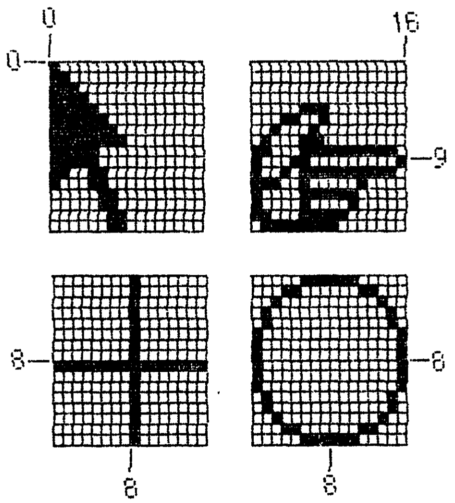


Figure 8. Cursors

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 (∅,∅) 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 (∅,∅) is at the top left of the image. For the arrow in Figure 8 to point to the mouse position, (∅,∅) 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.

Table of Contents

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.

Chapter 1

Introduction

The purpose of the software package described in Part III of this manual is to provide the features of the Standard Apple Numeric Environment (SANE) to assembly-language programmers using Apple's 68000-based systems. SANE—described in detail in Part I—fully supports the IEEE Standard (754) for Binary Floating-Point Arithmetic, and augments the Standard to provide greater utility for applications in accounting, finance, science, and engineering. The IEEE Standard and SANE offer a combination of quality, predictability, and portability heretofore unknown for numerical software.

A functionally equivalent 6502 assembly-language SANE engine is available for Apple's 6502-based systems. Thus numerical algorithms coded in assembly language for an Apple 68000-based system can be readily recoded for an Apple 6502-based system. We have chosen macros for accessing the 6502 and 68000 engines to make it easier to port algorithms from one system to the other.

Part III of this manual describes the use of the 68000 assembly-language SANE engine, but does not describe SANE itself. For example, Part III explains how to call the SANE remainder function from 68000 assembly language, but does not discuss what this function does. See Part I for information about the semantics of SANE.

See Appendix A for information about accessing the 68000 SANE engine from the Apple 68000-based systems.

Chapter 2

Basics

The following code illustrates a typical invocation of the SANE engine, FP68K.

```
PEA     A_ADR    ; Push address of A (single format)
PEA     B_ADR    ; Push address of B (extended format)
FSUBS            ; Floating-point SUBtract Single: B ← B - A
```

FSUBS is an assembly-language macro taken from the file listed in Appendix B. The form of the operation in the example (B ← B - A, where A is a numeric type and B is extended) is similar to the forms for most FP68K operations. Also, this example is typical of SANE engine calls because operands are passed to FP68K by pushing the addresses of the operands onto the stack prior to the call. Details of SANE engine access are given later in this chapter.

The SANE elementary functions are provided in Elems68K. Access to Elems68K is similar to access to FP86K; details are given in Chapter 9.

Operation Forms

The example above illustrates the form of an FP68K binary operation. Forms for other FP68K operations are described in this section. Examples and further details are given in subsequent chapters.

Arithmetic and Auxiliary Operations

Most numeric operations are either unary (one operand), like square root and negation, or binary (two operands), like addition and multiplication.

The 68000 assembly-language SANE engine, FP68K, provides unary operations in a one-address form:

DST ← <op> DST ... for example, B ← sqrt(B)

The operation <op> is applied to (or operates on) the operand DST and the result is returned to DST, overwriting the previous value. DST is called the destination operand.

FP68K provides binary operations in a two-address form:

DST ← DST <op> SRC ... for example, B ← B / A

The operation <op> is applied to the operands DST and SRC and the result is returned to DST, overwriting the previous value. SRC is called the source operand.

In order to store the result of an operation (unary or binary), the location of the operand DST must be known to FP68K, so DST is passed by address to FP68K. In general all operands, source and destination, are passed by address to FP68K.

For most operations the storage format for a source operand (SRC) can be one of the SANE numeric formats (single, double, extended, or comp). To support the extended-based SANE arithmetic, a destination operand (DST) must be in the extended format.

The forms for the copysign next-after functions are unusual and are discussed in Chapter 4.

Conversions

FP68K provides conversions between the extended format and other SANE formats, between extended and 16- or 32-bit integers, and between extended and decimal records. Conversions between binary formats (single, double, extended, comp, and integer) and conversions from decimal to binary have the form

DST ← SRC

Conversions from binary to decimal have the form

DST ← SRC according to SRC2

where SRC2 is a DecForm record specifying the decimal format for the conversion of SRC to DST.

Comparisons

Comparisons have the form

<relation> ← SRC, DST

where DST is extended and SRC is single, double, comp, or extended, and where <relation> is less, equal, greater, or unordered according as

DST <relation> SRC

Here the result <relation> is indicated by setting the 68000 CCR flags.

Other Operations

FP68K provides inquiries for determining the class and sign of an operand and operations for accessing the floating-point environment word and the halt address. Forms for these operations vary and are given as the operations are introduced.

External Access

The SANE engine, FP68K, is reentrant, position-independent code, which may be shared in multiprocess environments. It is accessed through one entry point, labeled FP68K. Each user process has a static state area consisting of one word of mode bits and error flags, and a two-word halt vector. The package allows for different access to the state word in single and multiprocess environments.

The package preserves all 68000 registers across invocations, except that REMAINDER modifies D0. The package modifies the 68000 CCR flags. Except for binary-decimal conversions, it uses little more stack area than is required to save the sixteen 32-bit 68000 registers. Because the binary-decimal conversions themselves call the package (to perform multiplies and divides), they use about twice the stack space of the regular operations.

The access constraints described in this section also apply to Elems68K.

Calling Sequence

A typical invocation of the engine consists of a sequence of PEA's to push operand addresses followed by one of the Appendix B macros:

```
PEA      <source address>
PEA      <destination address>
<FOPMACRO>
```

PEA's for source operands always precede those for destination operands. <FOPMACRO> represents a typical operation macro defined as

```
MOVE.W  <opword>,-(SP)          ; Push op code.
JSRFP
```

The macro JSRFP in turn generates a call to FP68K; for Macintosh™, it expands to an A-line trap, whereas for Lisa® it expands to an intrinsic unit subroutine call

```
JSR      FP68K .
```

The Opword

The opword is the logical OR of an operand format code and an operation code.

The operand format code specifies the format (extended, double, single, integer, or comp) of one of the operands. The operand format code typically gives the format for the source operand (SRC). At most one operand format need be specified, because other operands' formats are implied.

The operation code specifies the operation to be performed by FP68K.

(Opwords are listed in Appendix C; operand format codes and operation codes are listed in Appendix B.)

Example

The format code for single is 1000 (hex). The operation code for divide is 0006 (hex). Hence the opword 1006 (hex) indicates divide by a value of type single.

Assembly-Language Macros

The macro file in Appendix B provides macros for

```
MOVE.W  <opword>,-(SP)
JSRFP
```

for most common <opword> calls to FP68K.

Example 1

Add a single-format operand A to an extended-format operand B.

```
PEA     A_ADR   ; Push address of A
PEA     B_ADR   ; Push address of B
FADDS           ; Floating-point ADD Single: B ← B + A
```

Example 2

Compute B ← sqrt(A), where A and B are extended. The value of A should be preserved.

```
PEA     A_ADR   ; Push address of A
PEA     B_ADR   ; Push address of B
FX2X            ; Floating-point eXtended to eXtended: B ← A
PEA     B_ADR   ; Push address of B
FSQRTX          ; Floating SQuare RooT eXtended: B ← sqrt(B)
```

Example 3

Compute C ← A - B, where A, B, and C are in the double format. Because destinations are extended, a temporary extended variable T is required.

```
PEA     A_ADR   ; Push address of A
PEA     T_ADR   ; Push address of 10-byte temporary variable
FD2X            ; Fl-pt convert Double to eXtended: T ← A
PEA     B_ADR   ; Push address of B
PEA     T_ADR   ; Push address of temporary
FSUBD           ; Fl-pt SUBtract Double: T ← T - B
PEA     T_ADR   ; Push address of temporary
PEA     C_ADR   ; Push address of C
FX2D            ; Fl-pt convert eXtended to Double: C ← T
```

Arithmetic Abuse

FP68K is designed to be as robust as possible, but it is not bullet-proof. Passing the wrong number of operands to the engine damages the stack. Using UNDEFINED opword parameters or passing incorrect addresses produces undefined results.

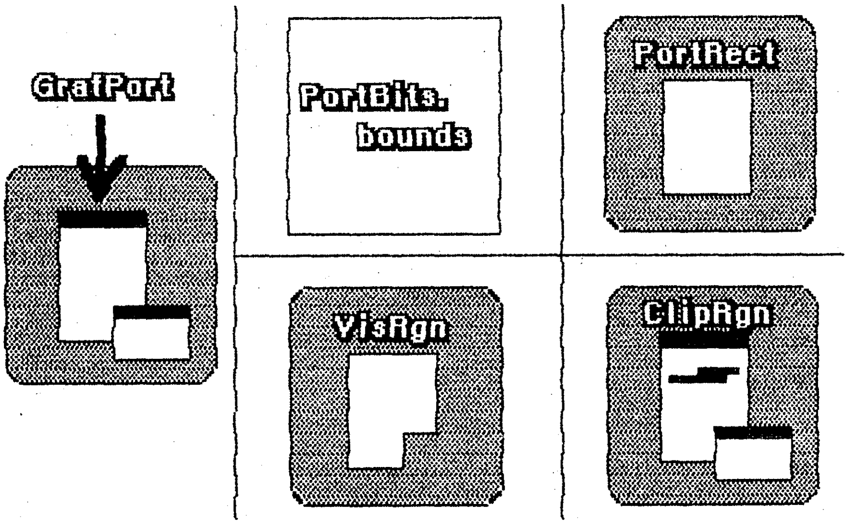


Figure 9. GrafPort Regions

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

Chapter 3

Data Types

FP68K fully supports the SANE data types

single	—	32-bit floating-point
double	—	64-bit floating-point
comp	—	64-bit integer
extended	—	80-bit floating-point

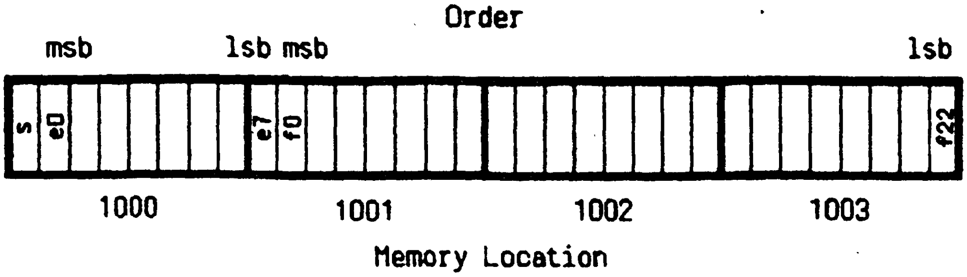
and the 68000-specific types

integer	—	16-bit two's complement integer
longint	—	32-bit two's complement integer

The 68000 engine uses the convention that least-significant bytes are stored in high memory. For example, let us take a variable of type single with bits

s	—	sign
e0 ... e7	—	exponent (msb...lsb)
f0 ... f22	—	significand fraction (msb...lsb)

The logical structure of this four-byte variable is shown below.



If this variable is assigned the address 1000, then its bits are distributed to the locations 1000 to 1003 as shown.

The other SANE formats (see Chapter 2 in Part I) are represented in memory in similar fashion.

Chapter 4

Arithmetic Operations and Auxiliary Routines

The operations covered in this chapter follow the access schemes described in Chapter 2.

Unary operations follow the one-address form:
DST ← <op> DST. They use the calling sequence

```
PEA      <DST address>
<FOPMACRO>
```

Binary operations follow the two-address form:
DST ← DST <op> SRC. They use the calling sequence

```
PEA      <SRC address>
PEA      <DST address>
<FOPMACRO>
```

The destination operand (DST) for these operations is passed by address and is generally in the extended format. The source operand (SRC) is also passed by address and may be single, double, comp, or extended. Some operations are distinguished by requiring some specific type for SRC, by using a nonextended destination, or by returning auxiliary information in the D0 register and in the processor CCR status bits. In this section, operations so distinguished are noted. The examples employ the macros in Appendix B.

Add, Subtract, Multiply, and Divide

These are binary operations and follow the two-address form.

Example

B ← B / A , where A is double and B is extended.

```
PEA     A_ADR    ; push address of A
PEA     B_ADR    ; push address of B
FDIVD            ; divide with source operand of type double
```

Square Root

This is a unary operation and follows the one-address form.

Example

B ← sqrt(B) , where B is extended.

```
PEA     B_ADR    ; push address of B
FSQRTX           ; square root (operand is always extended)
```

Round-to-Integer, Truncate-to-Integer

These are unary operations and follow the one-address form.

Round-to-integer rounds (according to the current rounding direction) to an integral value in the extended format. Truncate-to-integer rounds toward zero (regardless of the current rounding direction) to an integral value in the extended format. The calling sequence is the usual one for unary operators, illustrated above for square root.

Remainder

This is a binary operation and follows the two-address form.

Remainder returns auxiliary information: the low-order integer quotient (between -127 and +127) in D0.W. The high half of D0.L is undefined. This intrusion into the register file is extremely valuable in argument reduction—the principal use of the remainder function. The state of D0 after an invalid remainder is undefined.

Example

B ← B rem A , where A is single and B is extended.

```
PEA     A_ADR    ; push address of A
PEA     B_ADR    ; push address of B
FREMS            ; remainder with source operand of type single
```

Logb, Scalb

Logb is a unary operation and follows the one-address form.

Scalb is a binary operation and follows the two-address form. Its source operand is a 16-bit integer.

Example

B ← B $*$ 2^I, where B is extended.

```
PEA      I_ADR    ; push address of I
PEA      B_ADR    ; push address of B
FSCALBX           ; scalb
```

Negate, Absolute Value, Copy-Sign

Negate and absolute value are unary operations and follow the one-address form.

Copy-sign uses the calling sequence

```
PEA      <SRC address>
PEA      <DST address>
FCPYSGNX
```

to copy the sign of DST onto the sign of SRC. Note that copy-sign differs from most two-address operations in that it changes the SRC value rather than the DST value. The formats of the operands of FCPYSGNX can be single, double, or extended. (For efficiency, the 68000 assembly-language programmer should copy signs directly rather than calling FP68K.)

Example

Copy the sign of B (single, double, or extended) into the sign of A (single, double, or extended).

```
PEA      A_ADR    ; push address of A
PEA      B_ADR    ; push address of B
FCPYSGNX          ; copy-sign
```

Next-After

Both source and destination operands must be of the same floating-point type (single, double, or extended). The next-after operations use the calling sequence

```
PEA      <SRC address>
PEA      <DST address>
<next-after macro>
```

to effect SRC ← next value, in the format indicated by the macro, after SRC in the direction of DST. Note that next-after operations differ from most two-address operations in that they change SRC values rather than DST values.

Example

A ← next-after(A) in the direction of B, where A and B are double (so *next-after* means *next-double-after*).

```
PEA      A_ADR    ; push address of A
PEA      B_ADR    ; push address of B
FNEXTD            ; next-after in double format
```

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 1∅, the pen has four characteristics: a location, a size, a drawing mode, and a drawing pattern.

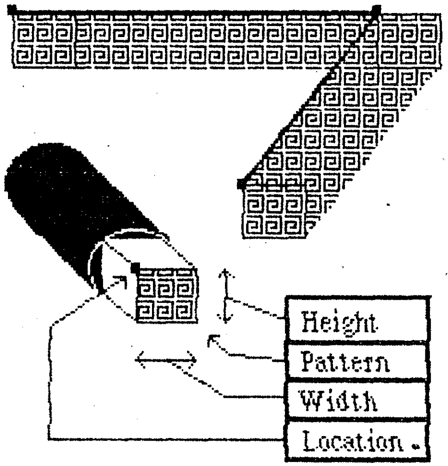


Figure 1∅. A Graphics Pen

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 (∅,∅) 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.

Chapter 5

Conversions

This chapter discusses conversions between binary formats and conversions between binary and decimal formats.

Conversions Between Binary Formats

FP68K provides conversions between the extended type and the SANE types single, double, and comp, as well as the 16- and 32-bit integer types.

Conversions to Extended

FP68K provides conversions of a source, of type single, double, comp, extended, or integer, to an extended destination.

```
                    single
                    double
extended     ←      comp
                    extended
                    integer
```

All operands, even integer ones, are passed by address. The following example illustrates the calling sequence.

Example

Convert A to B, where A is of type comp and B is extended.

```
PEA     A_ADR   ; push address of A
PEA     B_ADR   ; push address of B
FC2X            ; convert comp to extended
```

Conversions From Extended

FP68K provides conversions of an extended source to a destination of type single, double, comp, extended, or integer.

single
double
comp ← extended
extended
integer

(Note that conversion to a narrower format may alter values.) Contrary to the usual scheme, the destination for these conversions need not be of type extended. All operands are passed by address. The following example illustrates the calling sequence.

Example

Convert A to B where A is extended and B is double.

```
PEA     A_ADR    ; push address of A
PEA     B_ADR    ; push address of B
FX2D             ; convert extended to double
```

Binary-Decimal Conversions

FP68K provides conversions between the binary types (single, double, comp, extended, and integer) and the decimal record type.

Decimal records and decform records (used to specify the form of decimal representations) are described in Chapter 4 of Part I. For FP68K, the maximum length of the sig digits field of a decimal record is 20. (The value 20 is specific to this implementation: algorithms intended to port to other SANE implementations should use no more than 18 digits in sig.)

Binary to Decimal

The calling sequence for a conversion from a binary format to a decimal record passes the address of a decform record, the address of a binary source operand, and the address of a decimal-record destination. The maximum number of significant digits that will be returned is 19..

Example

Convert a comp-format value A to a decimal record D according to the decform record F.

```
PEA     F_ADR    ; push address of F
PEA     A_ADR    ; push address of A
PEA     D_ADR    ; push address of D
FC2DEC           ; convert comp to decimal
```

Fixed-Format "Overflow"

If a number is too large for a chosen fixed style, then FP68K returns the string '?' in the sig field of the decimal record.

Decimal to Binary

The calling sequence for a conversion from decimal to binary passes the address of a decimal-record source operand and the address of a binary destination operand.

The maximum number of digits in sig is 19. If the length of sig is 20, then sig represents its first 19 digits plus one or more additional nonzero digits after the 19th. The exponent corresponds to the 19-digit integer represented by the first 19 digits of sig.

Example

Convert the decimal record D to a double-format value B.

```
PEA     D_ADR    ; push address of D
PEA     B_ADR    ; push address of B
FDEC2D           ; convert decimal to double
```

Techniques for Maximum Accuracy

The following techniques apply to FP68K; other SANE implementations require other techniques.

For maximum accuracy, insert or delete trailing zeros for the sig field of a decimal record in order to minimize the magnitude of the exp field. For example, for 1.0E60 set sig to '1000000000000000000000000000' (17 zeros) and exp to 43, and for 300E-43 set sig to '3' and exp to -41.

If you are writing a parser and must handle a number with more than 19 significant digits, follow these rules:

- Place the implicit decimal point to the right of the 19 most significant digits.
- If any of the discarded digits to the right of the implicit decimal point are nonzero, then concatenate the digit '1' to sig.

Chapter 6

Comparisons and Inquiries

Comparisons

FP68K offers two comparison operations: FCPX (which signals invalid if its operands compare unordered) and FCMP (which does not). Each compares a source operand (which may be single, double, extended, or comp) with a destination operand (which must be extended). The result of a comparison is the relation (less, greater, equal, or unordered) for which

DST <relation> SRC

is true. The result is delivered in the X, N, Z, V, and C status bits:

Result	**Status Bits**				
	X	**N**	**Z**	**V**	**C**
greater	0	0	0	0	0
less	1	1	0	0	1
equal	0	0	1	0	0
unordered	0	0	0	1	0

These status bit encodings reflect that floating-point comparisons have four possible results, unlike the more familiar integer comparisons with three possible results. You need not learn these encodings, however; simply use the FBxxx series of macros for branching after FCMP and FCPX.

FCMP and FCPX are both provided to facilitate implementation of relational operators defined by higher level languages that do not contemplate unordered comparisons. The IEEE standard specifies that the invalid exception shall be signaled whenever necessary to alert users of such languages that an unordered comparison may have adversely affected their program's logic.

Example 1

Test B < = A, where B is extended and A is single; if TRUE branch to LOC; signal if unordered.

```
PEA       A_ADR     ; push address of A
PEA       B_ADR     ; push address of B
FCPXS               ; compare using source of type single,
                    ; signal invalid if unordered
FBLE      LOC       ; branch if B <= A
```

Example 2

Test B not-equal A, where B is extended and A is double; if TRUE branch to LOC. (Note that not-equal is equivalent to less, greater, or unordered, so invalid should not be signaled on unordered.)

```
PEA       A_ADR     ; push address of A
PEA       B_ADR     ; push address of B
FCMPD               ; compare using source of type double,
                    ; do not signal invalid if unordered
FBNE      LOC       ; branch if B not-equal A
```

Inquiries

The classify operation provides both class and sign inquiries. This operation takes one source operand (single, double, or extended), which is passed by address, and places the result in a 16-bit integer destination.

The sign of the result is the sign of the source; the magnitude of the result is

1	signaling NaN
2	quiet NaN
3	infinite
4	zero
5	normal
6	denormal

Example

Set C to sign and class of A.

```
PEA      A_ADR    ; push address of A
PEA      C_ADR    ; push address of result
FCLASSS           ; classify single
```

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.

Chapter 7

Environmental Control

The Environment Word

The floating-point environment is encoded in the 16-bit integer format as shown below in hexadecimal:

msb															lsb
–	r	r	x	d	o	u	i	–	R	R	X	D	O	U	I
	rounding direction		exception flags						rounding precision			halts enabled			

rounding direction, bits 6000			rr
0000	—	to-nearest	
2000	—	upward	
4000	—	downward	
6000	—	toward-zero	
exception flags, bits 1F00			
0100	—	invalid	i
0200	—	underflow	u
0400	—	overflow	o
0800	—	division-by-zero	d
1000	—	inexact	x

rounding precision, bits 0060			RR
0000	—	extended	
0020	—	double	
0040	—	single	
0060	—	UNDEFINED	

halts enabled, bits 001F			
0001	—	invalid	I
0002	—	underflow	U
0004	—	overflow	O
0008	—	division-by-zero	D
0010	—	inexact	X

Bits 8000 and 0080 are undefined.

Note that the default environment is represented by the integer value zero.

Example

With rounding toward-zero, inexact and underflow exception flags raised, extended rounding precision, and halt on invalid, overflow, and division-by-zero, the most significant byte of the environment is 72 and the least significant byte is 0D.

Access to the environment is via the operations get-environment, set-environment, test-exception, set-exception, procedure-entry, and procedure-exit.

Get-Environment and Set-Environment

Get-environment takes one input operand: the address of a 16-bit integer destination. The environment word is returned in the destination.

Set-environment has one input operand: the address of a 16-bit integer, which is to be interpreted as an environment word.

Example

Set rounding direction to toward-zero.

```
PEA       A_ADR
FGETENV
LEA       A_ADR,AO            ; AO gets address of A
MOVE.W    (AO),DO             ; DO gets environment
OR.W      #$6000,DO           ; set rounding toward-zero
MOVE.W    DO,(AO)             ; restore A
PEA       A_ADR
FSETENV
```

Test-Exception and Set-Exception

Test– exception takes one operand: the address of a 16– bit integer destination. On input the destination contains a bit index:

0 -- invalid
1 -- underflow
2 -- overflow
3 -- divide– by– zero
4 -- inexact

If the corresponding exception flag is set, then test– exception returns the value 1 in the high byte of the destination; otherwise it returns zero.

Example

Branch to XLOC if underflow is set.

```
MOVE.W   #FBUFLOW,-(SP)   ; underflow bit index
PEA      (SP)
FTESTXCP
TST.B    (SP)+                    ; test byte, pop word
BNE      XLOC
```

Set– exception takes one source operand, the address of a16– bit integer which encodes an exception in the manner described above for test– exception. Set– exception stimulates the indicated exception.

Procedure-Entry and Procedure-Exit

Procedure-entry saves the current floating-point environment (16-bit integer) at the address passed as the sole operand, and sets the operative environment to the default state.

Procedure-exit saves (temporarily) the exception flags, sets the environment passed as the sole operand, and then stimulates the saved exceptions.

Example

Here is a procedure that appears to its callers as an atomic operation.

```
ATOMICPROC
        PEA     E_ADR    ; push address to store environment
        FPROCENTRY       ; procedure entry

        ...body of routine...

        PEA     E_ADR    ; push address of environment
        FPROCEXIT        ; procedure exit
        RTS
```

Chapter 8

Halts

FP68K lets you transfer program control when selected floating-point exceptions occur. Because this facility will be used to implement halts in high-level languages, we refer to it as a halt mechanism. The assembly-language programmer can write a *halt handler* routine to cause special actions for floating-point exceptions. The FP68K halting mechanism differs from the traps that are an optional part of the IEEE Standard.

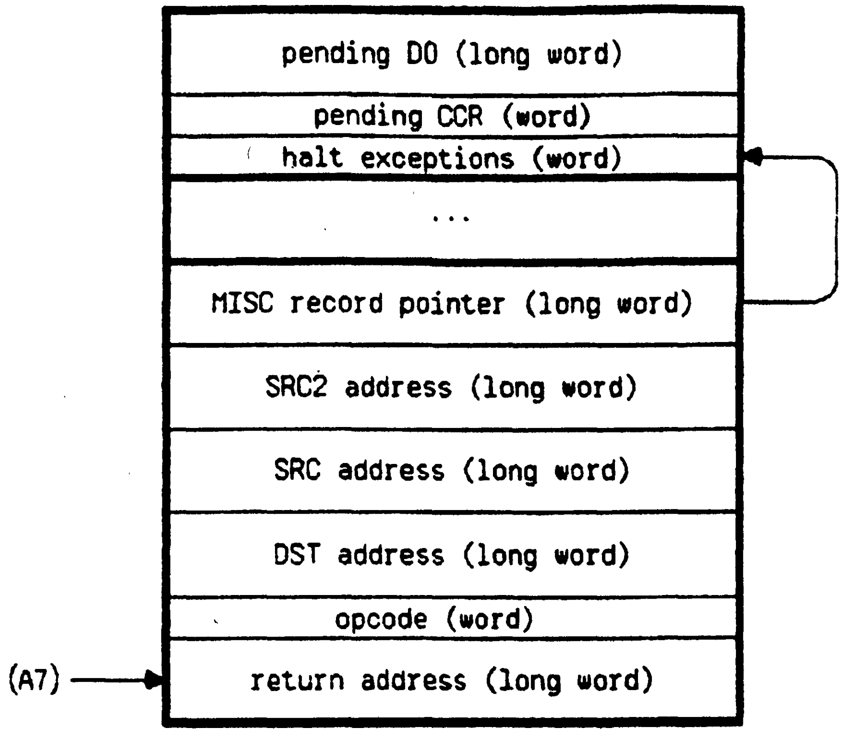
Conditions for a Halt

Any floating-point exception can, under the appropriate conditions, trigger a halt. The halt for a particular exception is enabled when the user has set the halt-enable bit corresponding to that exception.

The Halt Mechanism

If the halt for a given exception is enabled, FP68K does these things when that exception occurs:

1. FP68K delivers the same result to the destination address that it would return if the halt were not enabled.
2. It sets up the following stack frame:



The first word of the record MISC contains in its five low-order bits the AND of the halt-enable bits with the exceptions that occurred in the operation just completing. If halts were not enabled, then (upon return from FP68K) CCR and D0 would have the values given in MISC.

3. It passes control by JSR through the halt vector previously set by FSETHV, pushing another long word containing a return address in FP68K. If execution is to continue, the halt procedure must clear 18 bytes from the stack to remove the opword and the DST, SRC, SRC2, and MISC addresses.

Set-halt-vector has one input operand: the address of a 32-bit integer, which is interpreted as the halt vector (that is, the address to jump to in case a halt occurs).

Get-halt-vector has one input operand: the address of a 32-bit integer, which receives the halt vector.

Using the Halt Mechanism

This example illustrates the use of the halt mechanism. The user must set the halt vector to the starting address of a halt handler routine. This particular halt handler returns control to FP68K, which will continue as if no halt had occurred, returning to the next instruction in the user's program.

```
        LEA      HROUTINE,AO        ; AO gets address of halt routine
        MOVE.L   AO,H_ADR           ; H_ADR gets same
        PEA      H_ADR              ;
        FSETHV                      ; set halt vector to HROUTINE
        . . .
        PEA                         ; floating-point operand here
        <FOPMACRO>                  ; a floating-point call here
        . . .
HROUTINE                            ; called by FP68K
        MOVE.L   (SP)+,AO           ; AO saves return address in FP68K
        ADD.L    #18,SP             ; increment stack past arguments
        JMP      (AO)               ; return to FP68K
```

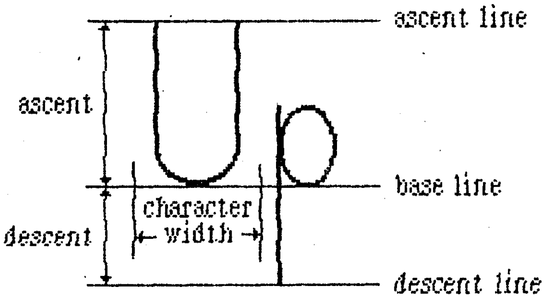


Figure 11. QuickDraw Characters

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.

The FP68K halt machanism is designed so that a halt procedure may be written in Lisa Pascal. This is the form of a Pascal equivalent to HROUTINE:

```
type    miscrec = record
                halterrors : integer ;
                ccrpending : integer ;
                DOpending : longint ;
        end {record} ;

procedure haltroutine
        ( var misc : miscrec ;
          src2, src, dst : longint ;
          opcode : integer ) ;

begin {haltroutine}
end {haltroutine} ;
```

Like HROUTINE, haltroutine merely continues execution as if no halt had occurred.

Chapter 9

Elementary Functions

The elementary functions that are specified by the Standard Apple Numeric Environment are made available to the 68000 assembly-language programmer in ELEMS68K. Also included are two functions that compute $\log_2(1 + x)$ and $2^x - 1$ accurately. ELEMS68K calls the SANE engine (FP68K) for its basic arithmetic. The access schemes for FP68K (described in Chapter 2) and ELEMS68K are similar. Opwords and sample macros are included at the end of the file listed in Appendix B. (These macros are used freely in the examples below.)

One-Argument Functions

The SANE elementary functions $\log_2(x)$, ln(x), ln1(x) = ln(1 + x), 2^x, e^x, exp1(x) = $e^x - 1$, cos(x), sin(x), tan(x), atan(x), and random(x), together with log21(x) = $\log_2(1 + x)$ and exp21(x) = $2^x - 1$, each have one extended argument, passed by address. These functions use the one-address calling sequence

```
PEA      DST
<EOPMACRO>
```

to effect

DST ← <op> DST

<EOPMACRO> is one of the macros in Appendix B that generate code to push an opword and invoke ELEMS68K. This calling sequence follows the FP68K access scheme for unary operations, such as square root and negate.

Example

B ← sin(B), where B is of extended type.

```
PEA      B_ADR    ; push address of B
FSINX             ; B ←  sin(B)
```

Two-Argument Functions

General exponentiation (x^y) has two extended arguments, both passed by address. The result is returned in x.

Integer exponentiation (x^i) also has two arguments. The extended argument x, passed by address, receives the result. The 16-bit integer argument i is also passed by address.

Both exponentiation functions use the calling sequence for binary operations

```
PEA      SRC address      ; push exponent address first
PEA      DST address      ; push base address second
<EOPMACRO>
```

to effect

DST ← DST^{SRC}

Example

B ← B^K, where the type of B is extended.

```
PEA      K_ADR    ; push address of K
PEA      B_ADR    ; push address of B
FXPWRI            ; integer exponentiation
```

Three-Argument Functions

Compound and annuity use the calling sequence

```
PEA     SRC2 address    ; push address of rate first
PEA     SRC  address    ; push address of number of periods second
PEA     DST  address    ; push address of destination third
<EOPMACRO>
```

to effect

DST ← <op> (SRC2, SRC)

where <op> is compound or annuity, SRC2 is the rate, and SRC is the number of periods. All arguments SRC2, SRC, and DST must be of the extended type.

Example

C ← $(1 + R)^N$, where C, R, and N are of type extended.

```
PEA     R_ADR   ; push address of R
PEA     N_ADR   ; push address of N
PEA     C_ADR   ; push address of C
FCOMPOUND       ; compound
```

Appendix A

68000 SANE Access

In your assemblies include the file TLASM/SANEMACS.TEXT, which contains the macros mentioned in this manual. The standard version is for Macintosh. For programs that will run on Lisa, redefine the symbol FPBYTRAP as follows:

```
FPBYTRAP .EQU 0
```

On Macintosh, the object code for FP68K and ELEMS68K is automatically loaded as needed by the Package Manager. On Lisa, it suffices to link your assembled code with the intrinsic unit file IOSFPLIB.OBJ.

Appendix B

68000 SANE Macros

```
 ;------------------------------------------------------------
 ;
 ; FILE: SANEMACS.TEXT
 ;
 ;  These macros and equates give assembly-language access to
 ;  the 68K floating-point arithmetic routines.
 ;------------------------------------------------------------

 ;------------------------------------------------------------
 ; WARNING: set FPBYTRAP for your system.
 ;------------------------------------------------------------
FPBYTRAP        .EQU    1       ;0 for Lisa, 1 for Macintosh

        .MACRO  JSRFP
            .IF     FPBYTRAP
                _FP68K          ;defined in TOOLMACS
            .ELSE
                .REF    FP68K
                JSR    FP68K
            .ENDC
        .ENDM

        .MACRO  JSRELEMS
            .IF    FPBYTRAP
                _ELEMS68K      ;defined in TOOLMACS
            .ELSE
                .REF    ELEMS68K
                JSR    ELEMS68K
            .ENDC
        .ENDM
```

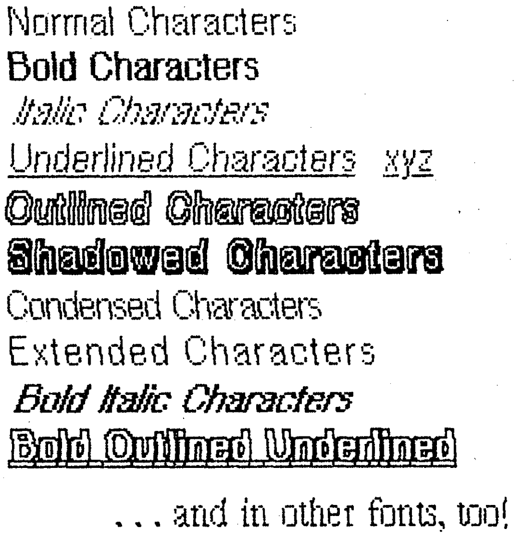


Figure 12. Character Styles

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.

```
 ;-----------------------------------------------------------
 ; Operation code masks.
 ;-----------------------------------------------------------
FOADD          .EQU    $0000  ; add
FOSUB          .EQU    $0002  ; subtract
FOMUL          .EQU    $0004  ; multiply
FODIV          .EQU    $0006  ; divide
FOCMP          .EQU    $0008  ; compare, no exception from unordered
FOCPX          .EQU    $000A  ; compare, signal invalid if unordered
FOREM          .EQU    $000C  ; remainder
FOZ2X          .EQU    $000E  ; convert to extended
FOX2Z          .EQU    $0010  ; convert from extended
FOSQRT         .EQU    $0012  ; square root
FORTI          .EQU    $0014  ; round to integral value
FOTTI          .EQU    $0016  ; truncate to integral value
FOSCALB        .EQU    $0018  ; binary scale
FOLOGB         .EQU    $001A  ; binary log
FOCLASS        .EQU    $001C  ; classify
 ; UNDEFINED     .EQU    $001E

FOSETENV      .EQU    $0001  ; set environment
FOGETENV       .EQU    $0003  ; get environment
FOSETHV        .EQU    $0005  ; set halt vector
FOGETHV        .EQU    $0007  ; get halt vector
FOD2B          .EQU    $0009  ; convert decimal to binary
FOB2D          .EQU    $000B  ; convert binary to decimal
FONEG          .EQU    $000D  ; negate
FOABS          .EQU    $000F  ; absolute
FOCPYSGN       .EQU    $0011  ; copy sign
FONEXT         .EQU    $0013  ; next-after
FOSETXCP       .EQU    $0015  ; set exception
FOPROCENTRY    .EQU    $0017  ; procedure entry
FOPROCEXIT     .EQU    $0019  ; procedure exit
FOTESTXCP      .EQU    $001B  ; test exception
 ; UNDEFINED     .EQU    $001D
 ; UNDEFINED     .EQU    $001F
 ;-----------------------------------------------------------
 ; Operand format masks.
 ;-----------------------------------------------------------
FFEXT          .EQU    $0000  ; extended  80-bit float
FFDBL          .EQU    $0800  ; double   64-bit float
FFSGL          .EQU    $1000  ; single   32-bit float
FFINT          .EQU    $2000  ; integer   16-bit integer
FFLNG          .EQU    $2800  ; long int  32-bit integer
FFCOMP         .EQU    $3000  ; comp     64-bit integer

 ;-----------------------------------------------------------
 ; Precision code masks: forces a floating point output
 ; value to be coerced to the range and precision specified.
 ;-----------------------------------------------------------
FCEXT          .EQU    $0000  ; extended
FCDBL          .EQU    $4000  ; double
FCSGL          .EQU    $8000  ; single

 ;-----------------------------------------------------------
 ; Operation macros: operand addresses should already be on
 ; the stack, with the destination address on top.  The
 ; suffix X, D, S, C, I, or L  determines the format of the
 ; source operand  extended, double, single, comp,
 ; integer, or long integer, respectively; the destination
 ; operand is always extended.
 ;-----------------------------------------------------------

 ;-----------------------------------------------------------
 ; Addition.
 ;-----------------------------------------------------------
        .MACRO  FADDX
        MOVE.W  #FFEXT+FOADD,-(SP)
        JSRFP
        .ENDM

        .MACRO  FADDD
        MOVE.W  #FFDBL+FOADD,-(SP)
        JSRFP
        .ENDM

        .MACRO  FADDS
        MOVE.W  #FFSGL+FOADD,-(SP)
        JSRFP
        .ENDM
        .MACRO   FADDC
        MOVE.W   #FFCOMP+FOADD,-(SP)
        JSRFP
        .ENDM

        .MACRO   FADDI
        MOVE.W   #FFINT+FOADD,-(SP)
        JSRFP
        .ENDM

        .MACRO   FADDL
        MOVE.W   #FFLNG+FOADD,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------
; Subtraction.
;-----------------------------------------------------------
        .MACRO   FSUBX
        MOVE.W   #FFEXT+FOSUB,-(SP)
        JSRFP
        .ENDM

        .MACRO   FSUBD
        MOVE.W   #FFDBL+FOSUB,-(SP)
        JSRFP
        .ENDM

        .MACRO   FSUBS
        MOVE.W   #FFSGL+FOSUB,-(SP)
        JSRFP
        .ENDM

        .MACRO   FSUBC
        MOVE.W   #FFCOMP+FOSUB,-(SP)
        JSRFP
        .ENDM

        .MACRO   FSUBI
        MOVE.W   #FFINT+FOSUB,-(SP)
        JSRFP
        .ENDM

        .MACRO   FSUBL
        MOVE.W   #FFLNG+FOSUB,-(SP)
        JSRFP
        .ENDM
```

```
;-----------------------------------------------------------
; Multiplication.
;-----------------------------------------------------------
        .MACRO   FMULX
        MOVE.W   #FFEXT+FOMUL,-(SP)
        JSRFP
        .ENDM

        .MACRO   FMULD
        MOVE.W   #FFDBL+FOMUL,-(SP)
        JSRFP
        .ENDM

        .MACRO   FMULS
        MOVE.W   #FFSGL+FOMUL,-(SP)
        JSRFP
        .ENDM

        .MACRO   FMULC
        MOVE.W   #FFCOMP+FOMUL,-(SP)
        JSRFP
        .ENDM

        .MACRO   FMULI
        MOVE.W   #FFINT+FOMUL,-(SP)
        JSRFP
        .ENDM

        .MACRO   FMULL
        MOVE.W   #FFLNG+FOMUL,-(SP)
        JSRFP
        .ENDM
;-----------------------------------------------------------
; Division.
;-----------------------------------------------------------
        .MACRO   FDIVX
        MOVE.W   #FFEXT+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO   FDIVD
        MOVE.W   #FFDBL+FODIV,-(SP)
        JSRFP
        .ENDM
        .MACRO   FDIVS
        MOVE.W   #FFSGL+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO   FDIVC
        MOVE.W   #FFCOMP+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO   FDIVI
        MOVE.W   #FFINT+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO   FDIVL
        MOVE.W   #FFLNG+FODIV,-(SP)
        JSRFP
        .ENDM
;-----------------------------------------------------------
; Square root.
;-----------------------------------------------------------
        .MACRO   FSQRTX
        MOVE.W   #FOSQRT,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------
; Round to integer, according to the current rounding mode.
;-----------------------------------------------------------
        .MACRO   FRINTX
        MOVE.W   #FORTI,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------
; Truncate to integer, using round toward zero.
;-----------------------------------------------------------
        .MACRO   FTINTX
        MOVE.W   #FOTTI,-(SP)
        JSRFP
        .ENDM
```

```
;------------------------------------------------------------
; Remainder.
;------------------------------------------------------------
        .MACRO   FREMX
        MOVE.W   #FFEXT+FOREM,-(SP)
        JSRFP
        .ENDM

        .MACRO   FREMD
        MOVE.W   #FFDBL+FOREM,-(SP)
        JSRFP
        .ENDM

        .MACRO   FREMS
        MOVE.W   #FFSGL+FOREM,-(SP)
        JSRFP
        .ENDM

        .MACRO   FREMC
        MOVE.W   #FFCOMP+FOREM,-(SP)
        JSRFP
        .ENDM

        .MACRO   FREMI
        MOVE.W   #FFINT+FOREM,-(SP)
        JSRFP
        .ENDM

        .MACRO   FREML
        MOVE.W   #FFLNG+FOREM,-(SP)
        JSRFP
        .ENDM

;------------------------------------------------------------
; Logb.
;------------------------------------------------------------
        .MACRO   FLOGBX
        MOVE.W   #FOLOGB,-(SP)
        JSRFP
        .ENDM
```

```
;-----------------------------------------------------------
; Scalb.
;-----------------------------------------------------------
       .MACRO  FSCALBX
       MOVE.W  #FFINT+FOSCALB,-(SP)
       JSRFP
       .ENDM

;-----------------------------------------------------------
; Copy-sign.
;-----------------------------------------------------------
       .MACRO  FCPYSGNX
       MOVE.W  #FOCPYSGN,-(SP)
       JSRFP
       .ENDM

;-----------------------------------------------------------
; Negate.
;-----------------------------------------------------------
       .MACRO  FNEGX
       MOVE.W  #FONEG,-(SP)
       JSRFP
       .ENDM

;-----------------------------------------------------------
; Absolute value.
;-----------------------------------------------------------
       .MACRO  FABSX
       MOVE.W  #FOABS,-(SP)
       JSRFP
       .ENDM

;-----------------------------------------------------------
; Next-after.  NOTE: both operands are of the same
; format, as specified by the usual suffix.
;-----------------------------------------------------------
       .MACRO  FNEXTS
       MOVE.W  #FFSGL+FONEXT,-(SP)
       JSRFP
       .ENDM

       .MACRO  FNEXTD
       MOVE.W  #FFDBL+FONEXT,-(SP)
       JSRFP
       .ENDM
```

```
        .MACRO   FNEXTX
        MOVE.W   #FFEXT+FONEXT,-(SP)
        JSRFP
        .ENDM
;------------------------------------------------------------
; Conversion to extended.
;------------------------------------------------------------
        .MACRO   FX2X
        MOVE.W   #FFEXT+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO   FD2X
        MOVE.W   #FFDBL+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO   FS2X
        MOVE.W   #FFSGL+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO   FI2X
        MOVE.W   #FFINT+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO   FL2X
        MOVE.W   #FFLNG+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO   FC2X
        MOVE.W   #FFCOMP+FOZ2X,-(SP)
        JSRFP
        .ENDM
```

```
;----------------------------------------------------------
; Conversion from extended.
;----------------------------------------------------------
       .MACRO   FX2D
       MOVE.W   #FFDBL+FOX2Z,-(SP)
       JSRFP
       .ENDM

       .MACRO   FX2S
       MOVE.W   #FFSGL+FOX2Z,-(SP)
       JSRFP
       .ENDM

       .MACRO   FX2I
       MOVE.W   #FFINT+FOX2Z,-(SP)
       JSRFP
       .ENDM

       .MACRO   FX2L
       MOVE.W   #FFLNG+FOX2Z,-(SP)
       JSRFP
       .ENDM

       .MACRO   FX2C
       MOVE.W   #FFCOMP+FOX2Z,-(SP)
       JSRFP
       .ENDM

;----------------------------------------------------------
; Binary to decimal conversion.
;----------------------------------------------------------
       .MACRO   FX2DEC
       MOVE.W   #FFEXT+FOB2D,-(SP)
       JSRFP
       .ENDM

       .MACRO   FD2DEC
       MOVE.W   #FFDBL+FOB2D,-(SP)
       JSRFP
       .ENDM

       .MACRO   FS2DEC
       MOVE.W   #FFSGL+FOB2D,-(SP)
       JSRFP
       .ENDM
```

```
        .MACRO  FC2DEC
        MOVE.W  #FFCOMP+FOB2D,-(SP)
        JSRFP
        .ENDM

        .MACRO  FI2DEC
        MOVE.W  #FFINT+FOB2D,-(SP)
        JSRFP
        .ENDM

        .MACRO  FL2DEC
        MOVE.W  #FFLNG+FOB2D,-(SP)
        JSRFP
        .ENDM

;------------------------------------------------------------
; Decimal to binary conversion.
;------------------------------------------------------------
        .MACRO  FDEC2X
        MOVE.W  #FFEXT+FOD2B,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDEC2D
        MOVE.W  #FFDBL+FOD2B,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDEC2S
        MOVE.W  #FFSGL+FOD2B,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDEC2C
        MOVE.W  #FFCOMP+FOD2B,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDEC2I
        MOVE.W  #FFINT+FOD2B,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDEC2L
        MOVE.W  #FFLNG+FOD2B,-(SP)
        JSRFP
        .ENDM
```

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

```
;-----------------------------------------------------------
; Compare, not signaling invalid on unordered.
;-----------------------------------------------------------
        .MACRO   FCMPX
        MOVE.W   #FFEXT+FOCMP,-(SP)
        JSRFP
        .ENDM

        .MACRO   FCMPD
        MOVE.W   #FFDBL+FOCMP,-(SP)
        JSRFP
        .ENDM

        .MACRO   FCMPS
        MOVE.W   #FFSGL+FOCMP,-(SP)
        JSRFP
        .ENDM

        .MACRO   FCMPC
        MOVE.W   #FFCOMP+FOCMP,-(SP)
        JSRFP
        .ENDM

        .MACRO   FCMPI
        MOVE.W   #FFINT+FOCMP,-(SP)
        JSRFP
        .ENDM

        .MACRO   FCMPL
        MOVE.W   #FFLNG+FOCMP,-(SP)
        JSRFP
        .ENDM
;-----------------------------------------------------------
; Compare, signaling invalid on unordered.
;-----------------------------------------------------------
        .MACRO   FCPXX
        MOVE.W   #FFEXT+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO   FCPXD
        MOVE.W   #FFDBL+FOCPX,-(SP)
        JSRFP
        .ENDM
        .MACRO   FCPXS
        MOVE.W   #FFSGL+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO   FCPXC
        MOVE.W   #FFCOMP+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO   FCPXI
        MOVE.W   #FFINT+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO   FCPXL
        MOVE.W   #FFLNG+FOCPX,-(SP)
        JSRFP
        .ENDM
;-----------------------------------------------------------
; The following macros define a set of so-called floating
; branches.  They presume that the appropriate compare
; operation, macro FCMPz or FCPXz, precedes.
;-----------------------------------------------------------
        .MACRO   FBEQ
        BEQ      %1
        .ENDM

        .MACRO   FBLT
        BCS      %1
        .ENDM

        .MACRO   FBLE
        BLS      %1
        .ENDM

        .MACRO   FBGT
        BGT      %1
        .ENDM

        .MACRO   FBGE
        BGE      %1
        .ENDM
        .MACRO  FBULT
        BLT     %1
        .ENDM

        .MACRO  FBULE
        BLE     %1
        .ENDM

        .MACRO  FBUGT
        BHI     %1
        .ENDM

        .MACRO  FBUGE
        BCC     %1
        .ENDM

        .MACRO  FBU
        BVS     %1
        .ENDM

        .MACRO  FBO
        BVC     %1
        .ENDM

        .MACRO  FBNE
        BNE     %1
        .ENDM

        .MACRO  FBUE
        BEQ     %1
        BVS     %1
        .ENDM

        .MACRO  FBLG
        BNE     %1
        BVC     %1
        .ENDM
;------------------------------------------------------------
; Short branch versions.
;------------------------------------------------------------
        .MACRO  FBEQS
        BEQ.S   %1
        .ENDM

        .MACRO  FBLTS
        BCS.S     %1
        .ENDM
```

```
.MACRO  FBLES
BLS.S    %1
.ENDM

.MACRO  FBGTS
BGT.S    %1
.ENDM

.MACRO  FBGES
BGE.S    %1
.ENDM

.MACRO  FBULTS
BLT.S    %1
.ENDM

.MACRO  FBULES
BLE.S    %1
.ENDM

.MACRO  FBUGTS
BHI.S    %1
.ENDM

.MACRO  FBUGES
BCC.S    %1
.ENDM

.MACRO  FBUS
BVS.S    %1
.ENDM

.MACRO  FBOS
BVC.S    %1
.ENDM

.MACRO  FBNES
BNE.S    %1
.ENDM

.MACRO  FBUES
BEQ.S    %1
BVS.S    %1
.ENDM
```

```
        .MACRO   FBLGS
        BNE.S     %1
        BVC.S     %1
        .ENDM

 ;---------------------------------------------------------
 ; Class and sign inquiries.
 ;---------------------------------------------------------
FCSNAN          .EQU    1       ; signaling NAN
FCQNAN          .EQU    2       ; quiet NAN
FCINF           .EQU    3       ; infinity
FCZERO          .EQU    4       ; zero
FCNORM          .EQU    5       ; normal number
FCDENORM        .EQU    6       ; denormal number

        .MACRO   FCLASSS
        MOVE.W   #FFSGL+FOCLASS,-(SP)
        JSRFP
        .ENDM

        .MACRO   FCLASSD
        MOVE.W   #FFDBL+FOCLASS,-(SP)
        JSRFP
        .ENDM

        .MACRO   FCLASSX
        MOVE.W   #FFEXT+FOCLASS,-(SP)
        JSRFP
        .ENDM
 ;-----------------------------------------------------------
 ; Bit indexes for bytes of floating point environment word.
 ;-----------------------------------------------------------
FBINVALID       .EQU    0       ; invalid operation
FBUFLOW         .EQU    1       ; underflow
FBOFLOW         .EQU    2       ; overflow
FBDIVZER        .EQU    3       ; division by zero
FBINEXACT       .EQU    4       ; inexact
FBRNDLO         .EQU    5       ; low bit of rounding mode
FBRNDHI         .EQU    6       ; high bit of rounding mode
FBLSTRND        .EQU    7       ; last round result bit
FBDBL           .EQU    5       ; double precision control
FBSGL           .EQU    6       ; single precision control

 ;-----------------------------------------------------------
 ; Get and set environment.
 ;-----------------------------------------------------------
        .MACRO  FGETENV
        MOVE.W  #FOGETENV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FSETENV
        MOVE.W  #FOSETENV,-(SP)
        JSRFP
        .ENDM

 ;-----------------------------------------------------------
 ; Test and set exception.
 ;-----------------------------------------------------------
        .MACRO  FTESTXCP
        MOVE.W  #FOTESTXCP,-(SP)
        JSRFP
        .ENDM

        .MACRO  FSETXCP
        MOVE.W  #FOSETXCP,-(SP)
        JSRFP
        .ENDM
```

```
;-----------------------------------------------------------
; Procedure entry and exit.
;-----------------------------------------------------------
       .MACRO  FPROCENTRY
       MOVE.W  #FOPROCENTRY,-(SP)
       JSRFP
       .ENDM

       .MACRO  FPROCEXIT
       MOVE.W  #FOPROCEXIT,-(SP)
       JSRFP
       .ENDM

;-----------------------------------------------------------
; Get and set halt vector.
;-----------------------------------------------------------
       .MACRO  FGETHV
       MOVE.W  #FOGETHV,-(SP)
       JSRFP
       .ENDM

       .MACRO  FSETHV
       MOVE.W  #FOSETHV,-(SP)
       JSRFP
       .ENDM
```

```
;-----------------------------------------------------------
; Elementary function operation code masks.
;-----------------------------------------------------------
FOLNX           .EQU     $0000   ; base-e log
FOLOG2X         .EQU     $0002   ; base-2 log
FOLN1X          .EQU     $0004   ; ln (1 + x)
FOLOG21X        .EQU     $0006   ; log2 (1 + x)

FOEXPX          .EQU     $0008   ; base-e exponential
FOEXP2X         .EQU     $000A   ; base-2 exponential
FOEXP1X         .EQU     $000C   ; exp (x) - 1
FOEXP21X        .EQU     $000E   ; exp2 (x) - 1

FOXPWRI         .EQU     $8010   ; integer exponentiation
FOXPWRY         .EQU     $8012   ; general exponentiation
FOCOMPOUND      .EQU     $C014   ; compound
FOANNUITY       .EQU     $C016   ; annuity

FOSINX          .EQU     $0018   ; sine
FOCOSX          .EQU     $001A   ; cosine
FOTANX          .EQU     $001C   ; tangent
FOATANX         .EQU     $001E   ; arctangent
FORAND  X       .EQU     $0020   ; random
```

```
;------------------------------------------------------------
; Elementary function macros.
;------------------------------------------------------------
       .MACRO   FLNX             ; base-e log
       MOVE.W   #FOLNX,-(SP)
       JSRELEMS
       .ENDM

       .MACRO   FLOG2X           ; base-2 log
       MOVE.W   #FOLOG2X,-(SP)
       JSRELEMS
       .ENDM

       .MACRO   FLN1X           ; ln (1 + x)
       MOVE.W   #FOLN1X,-(SP)
       JSRELEMS
       .ENDM

       .MACRO   FLOG21X         ; log2 (1 + x)
       MOVE.W   #FOLOG21X,-(SP)
       JSRELEMS
       .ENDM

       .MACRO   FEXPX           ; base-e exponential
       MOVE.W   #FOEXPX,-(SP)
       JSRELEMS
       .ENDM

       .MACRO   FEXP2X           ; base-2 exponential
       MOVE.W   #FOEXP2X,-(SP)
       JSRELEMS
       .ENDM

       .MACRO   FEXP1X           ; exp (x) - 1
       MOVE.W   #FOEXP1X,-(SP)
       JSRELEMS
       .ENDM

       .MACRO   FEXP21X         ; exp2 (x) - 1
       MOVE.W   #FOEXP21X,-(SP)
       JSRELEMS
       .ENDM
```

```
.MACRO   FXPWRI          ; integer exponential
MOVE.W   #FOXPWRI,-(SP)
JSRELEMS
.ENDM

.MACRO   FXPWRY          ; general exponential
MOVE.W   #FOXPWRY,-(SP)
JSRELEMS
.ENDM

.MACRO   FCOMPOUND       ; compound
MOVE.W   #FOCOMPOUND ,-(SP)
JSRELEMS
.ENDM

.MACRO   FANNUITY        ; annuity
MOVE.W   #FOANNUITY ,-(SP)
JSRELEMS
.ENDM

.MACRO   FSINX           ; sine
MOVE.W   #FOSINX,-(SP)
JSRELEMS
.ENDM

.MACRO   FCOSX           ; cosine
MOVE.W   #FOCOSX,-(SP)
JSRELEMS
.ENDM

.MACRO   FTANX           ; tangent
MOVE.W   #FOTANX,-(SP)
JSRELEMS
.ENDM

.MACRO   FATANX          ; arctangent
MOVE.W   #FOATANX,-(SP)
JSRELEMS
.ENDM

.MACRO   FRAND  IX       ; random number generator
MOVE.W   #FORANDI IX,-(SP)
JSRELEMS
.ENDM
```

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).

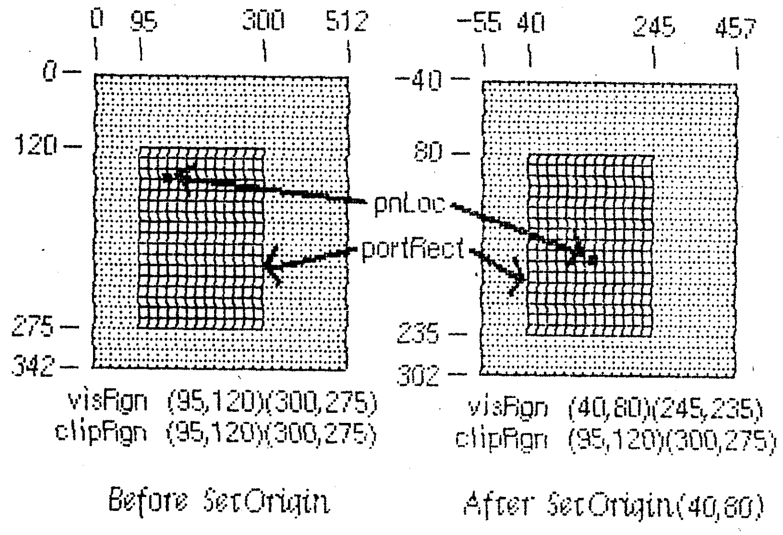


Figure 13. Changing Local Coordinates

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

```
 ;-------------------------------------------------------
 ; NaN codes.
 ;-------------------------------------------------------
NANSQRT .EQU     1     ; Invalid square root such as sqrt(-1).
NANADD  .EQU     2     ; Invalid addition such as +INF - +INF.
NANDIV  .EQU     4     ; Invalid division such as 0/0.
NANMUL  .EQU     8     ; Invalid multiply such as 0 * INF.
NANREM  .EQU     9     ; Invalid remainder or mod such as x REM 0.
NANASCBIN .EQU  17      ; Attempt to convert invalid ASCII string.
NANCOMP .EQU    20      ; Result of converting comp NaN to floating.
NANZERO .EQU    21      ; Attempt to create a NaN with a zero code.
NANTRIG .EQU    33      ; Invalid argument to trig routine.
NANINVTRIG .EQU 34      ; Invalid argument to inverse trig routine.
NANLOG  .EQU    36      ; Invalid argument to log routine.
NANPOWER .EQU  37      ; Invalid argument to x^i or x^y routine.
NANFINAN .EQU  38      ; Invalid argument to financial function.
NANINIT .EQU   255      ; Uninitialized storage.
 ;-------------------------------------------------------
```

Appendix C

68000 SANE Quick Reference Guide

This guide contains diagrams of the SANE data formats and the 68K SANE operations and environment word.

Formats of SANE Types

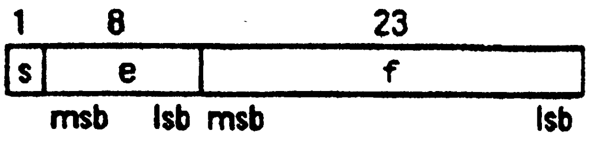
Each of the diagrams below is followed by the rules for evaluating the number v.

In each field of each diagram, the leftmost bit is the msb and the rightmost is the lsb.

Table C-1. *Format Diagram Symbols*

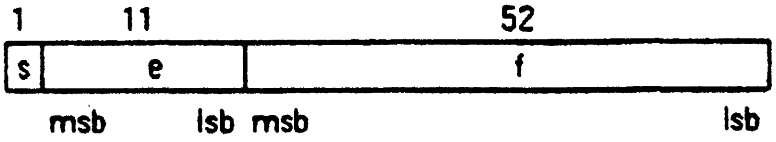
v	value of number
s	sign bit
e	biased exponent
i	explicit one's-bit (extended type only)
f	fraction

Single: 32 Bits



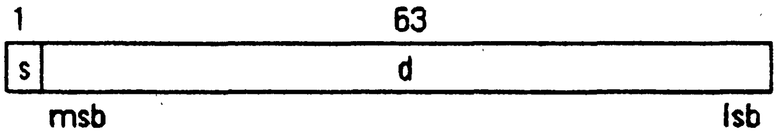
If $0 < e < 255$, then $v = (-1)^s * 2^{(e-127)} * (1.f)$.
If $e = 0$ and $f \neq 0$, then $v = (-1)^s * 2^{(-126)} * (0.f)$.
If $e = 0$ and $f = 0$, then $v = (-1)^s * 0$.
If $e = 255$ and $f = 0$, then $v = (-1)^s * \infty$.
If $e = 255$ and $f \neq 0$, then v is a NaN.

Double: 64 Bits



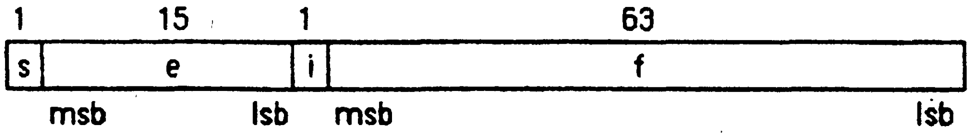
If $0 < e < 2047$, then $v = (-1)^s * 2^{(e-1023)} * (1.f)$.
If $e = 0$ and $f \neq 0$, then $v = (-1)^s * 2^{(-1022)} * (0.f)$.
If $e = 0$ and $f = 0$, then $v = (-1)^s * 0$.
If $e = 2047$ and $f = 0$, then $v = (-1)^s * \infty$.
If $e = 2047$ and $f \neq 0$, then v is a NaN.

Comp: 64 Bits



If s = 1 and d = 0, then v is the unique comp NaN.
Otherwise, v is the two's-complement value of the 64-bit representation.

Extended: 80 Bits



If 0 < = e < 32767, then v = $(-1)^{s} * 2^{(e-16383)} * (i.f)$.
If e = 32767 and f = 0, then v = $(-1)^{s} * \infty$, regardless of i.
If e = 32767 and f ≠ 0, then v is a NaN, regardless of i.

Operations

In the operations below, the operation's mnemonic is followed by the opword in parentheses: the first byte is the operation code; the second is the operand format code. For some operations, the first byte of the opword (xx) is ignored.

Abbreviations and Symbols

The symbols and abbreviations in this section closely parallel those in the text, although some are shortened. In some cases, the same symbol has various meanings, depending on context.

Operands

DST	destination operand (passed by address)
SRC	source operand (passed by address), pushed before DST
SRC2	second source operand (passed by address), pushed before SRC

Data Types

X	extended (80 bits)
D	double (64 bits)
S	single (32 bits)
I	integer (16 bits)
L	longint (32 bits)
C	comp (64 bits)
Dec	decimal Record
Decform	decform Record

68000 Processor Registers

D0	data register 0
X	extend bit of processor status register
N	negative bit of processor status register
Z	zero bit of processor status register
V	overflow bit of processor status register
C	carry bit of processor status register

Exceptions

I	invalid operation
U	underflow
O	overflow
D	divide-by-zero
X	inexact

For each operation, an exception marked with x indicates that the operation will signal the exception for some input.

Environment and Halts

EnWrd	SANE environment word (16-bit integer)
HltVctr	SANE halt vector (32-bit longint)

Arithmetic Operations and Auxiliary Routines (Entry Point FP68K)

Operation	Operands and Data Types					Exceptions				
ADD	DST	←	DST	+	SRC	I	U	O	D	X
FADDX (0000)	X		X		X	x	-	x	-	x
FADDD (0800)	X		X		D	x	-	x	-	x
FADDS (1000)	X		X		S	x	-	x	-	x
FADDC (3000)	X		X		C	x	-	x	-	x
FADDI (2000)	X		X		I	x	-	x	-	x
FADDL (2800)	X		X		L	x	-	x	-	x
SUBTRACT	DST	←	DST	-	SRC	I	U	O	D	X
FSUBX (0002)	X		X		X	x	-	x	-	x
FSUBD (0802)	X		X		D	x	-	x	-	x
FSUBS (1002)	X		X		S	x	-	x	-	x
FSUBC (3002)	X		X		C	x	-	x	-	x
FSUBI (2002)	X		X		I	x	-	x	-	x
FSUBL (2802)	X		X		L	x	-	x	-	x
MULTIPLY	DST	←	DST	*	SRC	I	U	O	D	X
FMULX (0004)	X		X		X	x	x	x	-	x
FMULD (0804)	X		X		D	x	x	x	-	x
FMULS (1004)	X		X		S	x	x	x	-	x
FMULC (3004)	X		X		C	x	-	x	-	x
FMULI (2004)	X		X		I	x	-	x	-	x
FMULL (2804)	X		X		L	x	-	x	-	x

Operation	Operands and Data Types					Exceptions				
DIVIDE	DST	←	DST	/	SRC	I	U	O	D	X
FDIVX (0006)	X		X		X	x	x	x	x	x
FDIVD (0806)	X		X		D	x	x	x	x	x
FDIVS (1006)	X		X		S	x	x	x	x	x
FDIVC (3006)	X		X		C	x	x	-	x	x
FDIVI (2006)	X		X		I	x	x	-	x	x
FDIVL (2806)	X		X		L	x	x	-	x	x
SQUARE ROOT	DST	←	sqrt(DST)			I	U	O	D	X
FSQRTX (0012)	X		X			x	-	-	-	x
ROUND TO INT	DST	←	rnd(DST)			I	U	O	D	X
FRINTX (0014)	X		X			x	-	-	-	x
TRUNC TO INT	DST	←	chop(DST)			I	U	O	D	X
FTINTX (0016)	X		X			x	-	-	-	x
REMAINDER	DST	←	DST	REM	SRC	I	U	O	D	X
FREMX (000C)	X		X		X	x	-	-	-	-
FREMD (080C)	X		X		D	x	-	-	-	-
FREMS (100C)	X		X		S	x	-	-	-	-
FREMC (300C)	X		X		C	x	-	-	-	-
FREMI (200C)	X		X		I	x	-	-	-	-
FREML (280C)	X		X		L	x	-	-	-	-
	D0	←	integer quotient DST/SRC, between -127 and +127							
LOG BINARY	DST	←	logb(DST)			I	U	O	D	X
FLOGBX (001A)	X		X			x	-	-	x	-
SCALE BINARY	DST	←	DST * 2^SRC			I	U	O	D	X
FSCALBX (0018)	X		X		I	x	x	x	-	x
NEGATE	DST	←	-DST			I	U	O	D	X
FNEGX (000D)	X		X			-	-	-	-	-
ABSOLUTE VALUE	DST	←	|DST|			I	U	O	D	X
FABSX (000F)	X		X			-	-	-	-	-
COPY-SIGN	SRC	←	SRC with		DST's sign	I	U	O	D	X
FCPYSGNX (0011)	X, D, or S		X, D, or S		X, D, or S	-	-	-	-	-
NEXT-AFTER	SRC	←	next after SRC toward DST			I	U	O	D	X
FNEXTX (0013)	X		X		X	x	x	x	-	x
FNEXTD (0813)	D		D		D	x	x	x	-	x
FNEXTS (1013)	S		S		S	x	x	x	-	x

Conversions (Entry Point FP68K)

Operation	Operands and Data Types				Exceptions				
CONVERT									
Bin to Bin	DST	←	SRC		I	U	O	D	X
FX2X (0010)	X		X		x	-	-	-	-
FX2D (0810)	D		X		x	x	x	-	x
FX2S (1010)	S		X		x	x	x	-	x
FX2C (3010)	C		X		x	-	-	-	x
FX2I (2010)	I		X		x	-	-	-	x
FX2L (2810)	L		X		x	-	-	-	x
FD2X (080E)	X		D		x	-	-	-	-
FS2X (100E)	X		S		x	-	-	-	-
FC2X (300E)	X		C		-	-	-	-	-
FI2X (200E)	X		I		-	-	-	-	-
FL2X (280E)	X		L		-	-	-	-	-
Bin to Dec	DST	←	SRC according to	SRC2	I	U	O	D	X
FX2DEC (000B)	Dec		X	Decform	x	-	-	-	x
FD2DEC (080B)	Dec		D	Decform	x	-	-	-	x
FS2DEC (100B)	Dec		S	Decform	x	-	-	-	x
FC2DEC (300B)	Dec		C	Decform	-	-	-	-	x
FI2DEC (200B)	Dec		I	Decform	-	-	-	-	x
FL2DEC (280B)	Dec		L	Decform	-	-	-	-	x

(First SRC2 is pushed, then SRC, then DST.)

Operation	DST	←	SRC	I	U	O	D	X
Dec to Bin	DST	←	SRC	I	U	O	D	X
FDEC2X (0009)	X		Dec	-	x	x	-	x
FDEC2D (0809)	D		Dec	-	x	x	-	x
FDEC2S (1009)	S		Dec	-	x	x	-	x
FDEC2C (3009)	C		Dec	x	-	-	-	x
FDEC2I (2009)	I		Dec	x	-	-	-	x
FDEC2L (2809)	L		Dec	x	-	-	-	x

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).

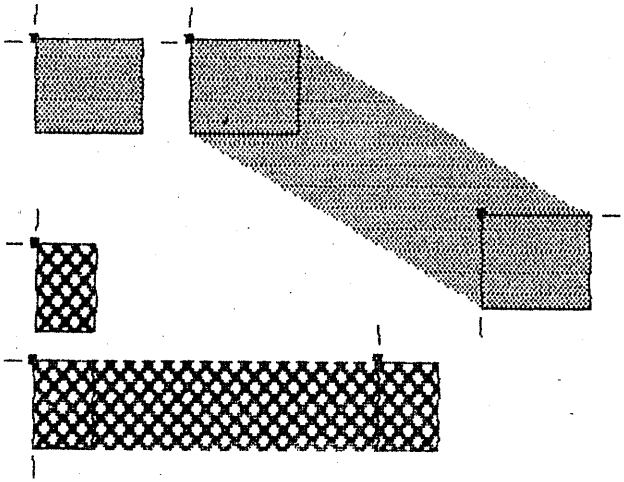


Figure 14. Drawing Lines

Compare and Classify (Entry Point FP68K)

Operation	**Operands and Data Types**		**Exceptions**				
COMPARE							
No invalid	Status Bits ← <relation>		I	U	O	D	X
for unordered	where DST <relation>	SRC					
FCMPX (0008)	X	X	x	-	-	-	-
FCMPD (0808)	X	D	x	-	-	-	-
FCMPS (1008)	X	S	x	-	-	-	-
FCMPC (3008)	X	C	x	-	-	-	-
FCMPI (2008)	X	I	x	-	-	-	-
FCMPL (2808)	X	L	x	-	-	-	-

(Invalid only for signaling NaN inputs.)

Signal invalid	Status Bits ← <relation>		I	U	O	D	X
if unordered	where DST <relation>	SRC					
FCPXX (000A)	X	X	x	-	-	-	-
FCPXD (080A)	X	D	x	-	-	-	-
FCPXS (100A)	X	S	x	-	-	-	-
FCPXC (300A)	X	C	x	-	-	-	-
FCPXI (200A)	X	I	x	-	-	-	-
FCPXL (280A)	X	L	x	-	-	-	-

<relation>	**Status Bits**				
	X	**N**	**Z**	**V**	**C**
DST > SRC	0	0	0	0	0
DST < SRC	1	1	0	0	1
DST = SRC	0	0	1	0	0
DST & SRC unordered	0	0	0	1	0

CLASSIFY	<class>	←	class of	SRC	I	U	O	D	X
	<sign>	←	sign of	SRC					
	DST	←	(-1) ^ <sign> * <class>						
FCLASSX (001C)	I			X	-	-	-	-	-
FCLASSD (081C)	I			D	-	-	-	-	-
FCLASSS (101C)	I			S	-	-	-	-	-

SRC	<class>	SRC	<sign>
signaling NaN	1	positive	0
quiet NaN	2	negative	1
infinite	3		
zero	4		
normalized	5		
denormalized	6		

Environmental Control (Entry Point FP68K)

Operation	Operands and Data Types	Exceptions
GET ENVIRONMENT FGETENV (0003)	DST ← EnvWrd I	I U O D X - - - - -
SET ENVIRONMENT FSETENV (0001)	EnvWrd ← SRC I	I U O D X x x x x x

(Exceptions set by set-environment cannot cause halts.)

Operation	Operands and Data Types	Exceptions
TEST EXCEPTION FTESTXCP (001B)	DST high byte <-- DST Xcp set I I	I U O D X - - - - -
SET EXCEPTION FSETXCP (0015)	EnvWrd ← EnvWrd AND SRC I	I U O D X x x x x x
PROCEDURE ENTRY FPROCENTRY (0017)	DST ← EnvWrd, EnvWrd ← 0 I	I U O D X x x x x x
PROCEDURE EXIT FPROCEXIT (0019)	EnvWrd ← SRC OR current Xcps I	I U O D X x x x x x

Halt Control (Entry Point FP68K)

Operation	Operands and Data Types	Exceptions
SET HALT VECTOR FSETHV (xx05)	HltVctr ← SRC L	I U O D X - - - - -
GET HALT VECTOR FGETHV (0007)	DST ← HltVctr L	I U O D X - - - - -

Elementary Functions (Entry Point ELEMS68K)

Operation	**Operands and Data Types**	**Exceptions**
BASE-E LOGARITHM	DST ← ln(DST)	I U O D X
FLNX (0000)	X X	x - - x x
BASE-2 LOGARITHM	DST ← log2(DST)	I U O D X
FLOG2X (0002)	X X	x - - x x
BASE-E LOG1 (LN1)	DST ← ln(1 + DST)	I U O D X
FLN1X (0004)	X X	x x - x x
BASE-2 LOG1	DST ← log2(1 + DST)	I U O D X
FLOG21X (0006)	X X	x x - x x
BASE-E EXPONENTIAL	DST ← e ^ DST	I U O D X
FEXPX (0008)	X X	x x x - x
BASE-2 EXPONENTIAL	DST ← 2 ^ DST	I U O D X
FEXP2X (000A)	X X	x x x - x
BASE-E EXP1	DST ← e ^ DST - 1	I U O D X
FEXP1X (000C)	X X	x x x - x
BASE-2 EXP1	DST ← 2 ^ DST - 1	I U O D X
FEXP21X (000E)	X X	x x x - x
INTEGER EXPONENTIATION	DST ← DST ^ SRC	I U O D X
FXPWRI (8010)	X X I	x x x x x
GENERAL EXPONENTIATION	DST ← DST ^ SRC	I U O D X
FXPWRY (8012)	X X X	x x x x x
COMPOUND INTEREST	DST ← compound(SRC2,SRC)	I U O D X
FCOMPOUND (C014)	X X X	x x x x x

(SRC2 is the rate; SRC is the number of periods.)

ANNUITY FACTOR	DST ← annuity(SRC2,SRC)	I U O D X
FANNUITY (C016)	X X X	x x x x x

(SRC2 is the rate; SRC is the number of periods.)

SINE	DST ← sin(DST)	I U O D X
FSINX (0018)	X X	x x - - x
COSINE	DST ← cos(DST)	I U O D X
FCOSX (001A)	X X	x x - - x

TANGENT	DST	←	tan(DST)	I	U	O	D	X
FTANX (001C)	X		X	x	x	-	x	x
ARCTANGENT	DST	←	atan(DST)	I	U	O	D	X
FATANX (001E)	X		X	x	x	-	-	x
RANDOM	DST	←	random(DST)	I	U	O	D	X
FRANDX (0020)	X		X	x	x	x	-	x

Environment Word

The floating-point environment is encoded in the 16-bit integer format as shown below in hexadecimal:

msb															lsb
–	r	r	x	d	o	u	i	–	R	R	X	D	O	U	I
	rounding direction		exception flags						rounding precision		halts enabled				

rounding direction, bits 6000			rr
0000	–	to-nearest	
2000	–	upward	
4000	–	downward	
6000	–	toward-zero	
exception flags, bits 1F00			
0100	–	invalid	i
0200	–	underflow	u
0400	–	overflow	o
0800	–	division-by-zero	d
1000	–	inexact	x

(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).

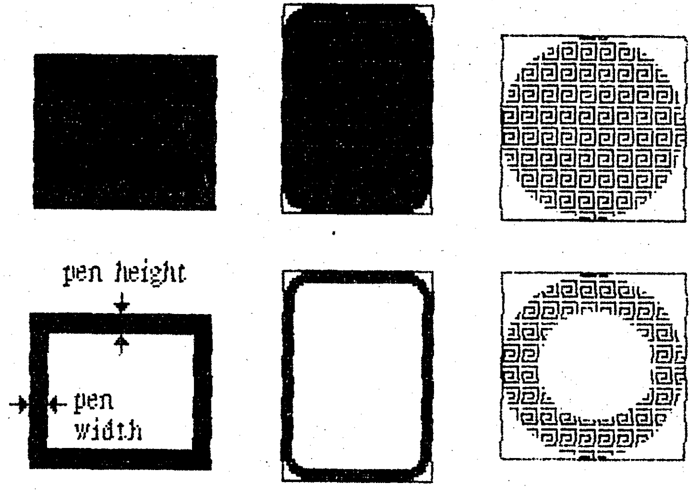


Figure 15. Solid Shapes and Framed Shapes

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).

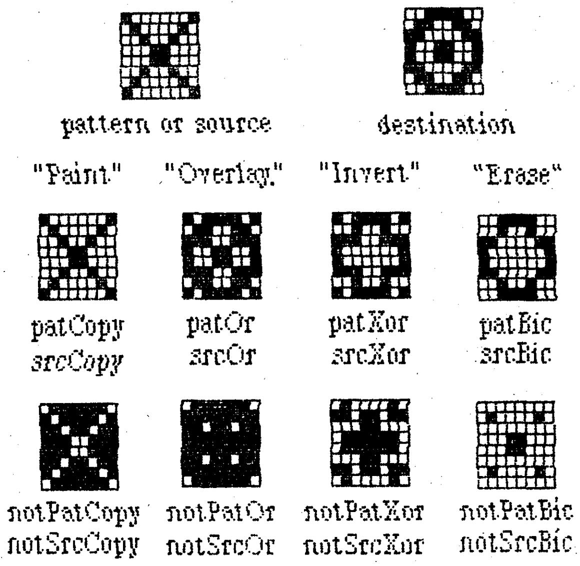


Figure 16. Transfer Modes

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 = Ø;
             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.

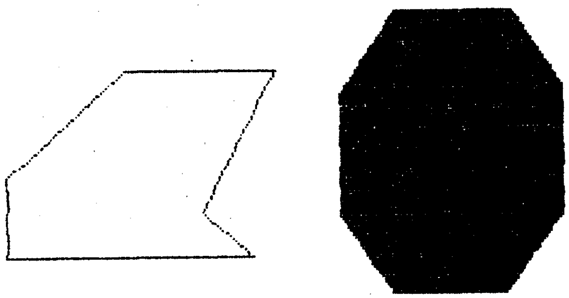


Figure 17. Polygons

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	5Ø% gray pattern
ltGray	Pattern	25% gray pattern
dkGray	Pattern	75% gray pattern
arrow	Cursor	pointing arrow cursor
screenBits	BitMap	Macintosh screen, (Ø,Ø,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).

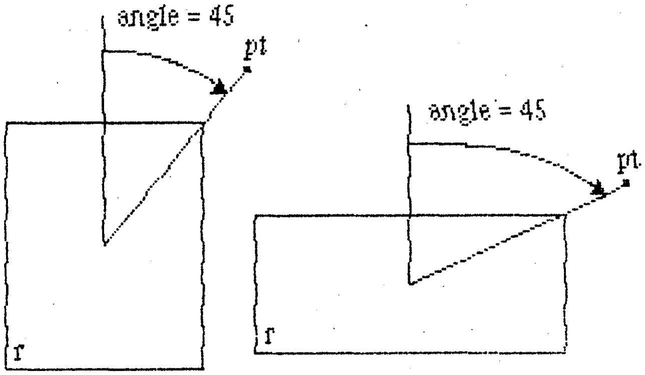


Figure 18. PtToAngle

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.

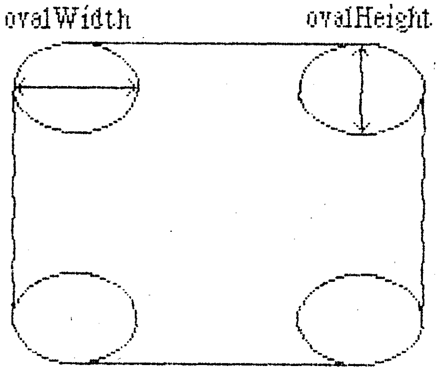


Figure 19. Rounded-Corner Rectangle

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 36Ø. 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, 9Ø (or -27Ø) is at 3 o'clock, 18Ø (or -18Ø) is at 6 o'clock, and 27Ø (or -9Ø) 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 2Ø).

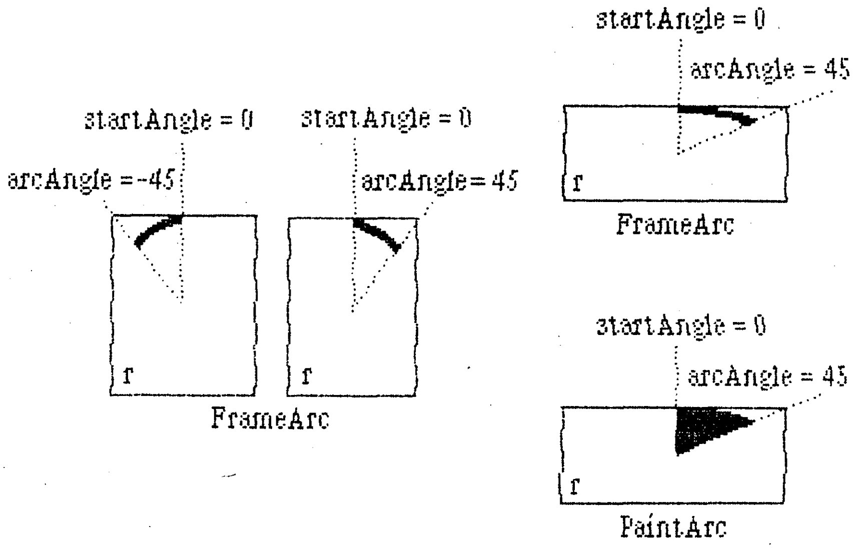


Figure 2Ø. Operations on Arcs and Wedges

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).

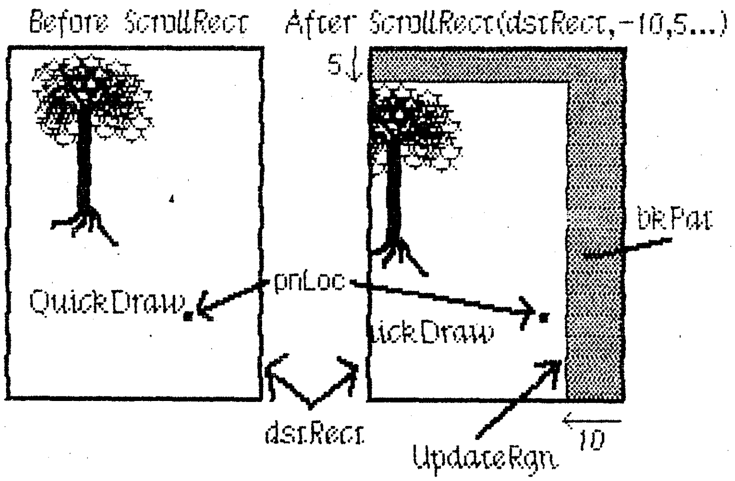


Figure 21. Scrolling

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).

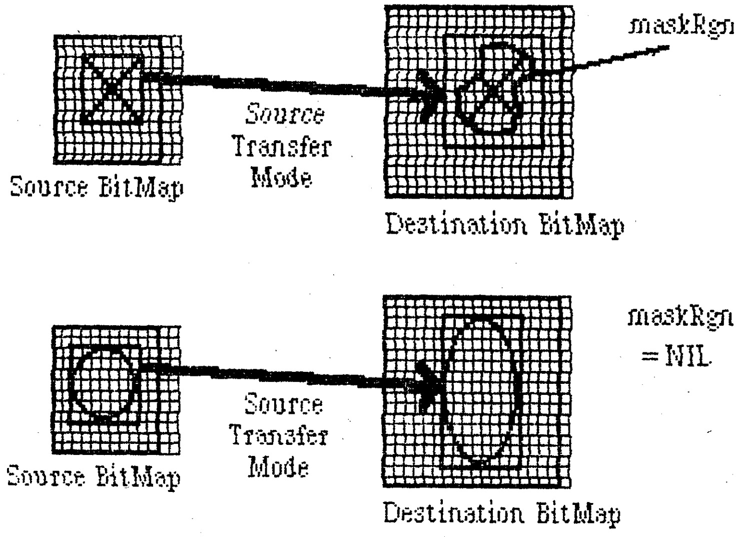


Figure 22. Operation of CopyBits

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(300,100);          { move to first point and }
  LineTo(400,200);          {         form            }
  LineTo(200,200);          {          the            }
  LineTo(300,100);          {        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.

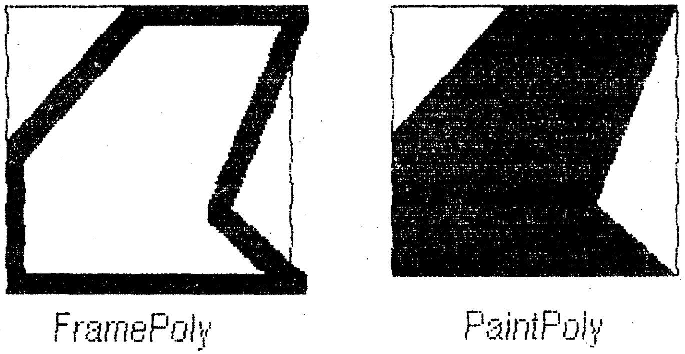


Figure 23. Drawing Polygons

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!}
```

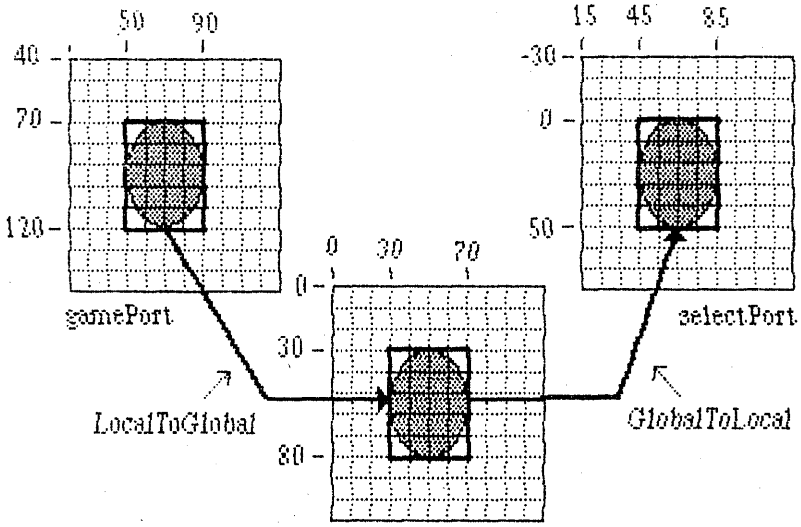


Figure 24. Converting between Coordinate Systems

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´)
```

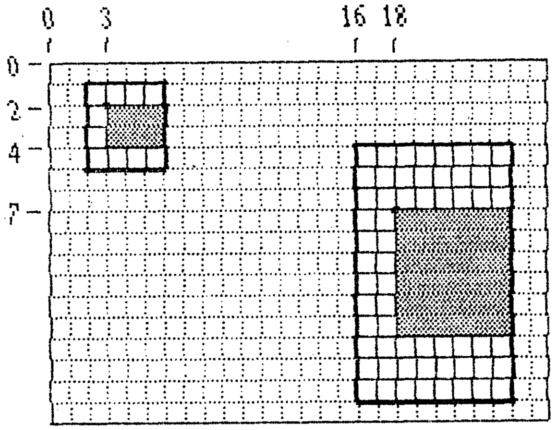
places a striped pattern into the pattern variable stripes.

(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.



ScalePt scales pen size (3,2) to (6,6).
MapPt maps point (3,2) to (18,7).

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 80 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.

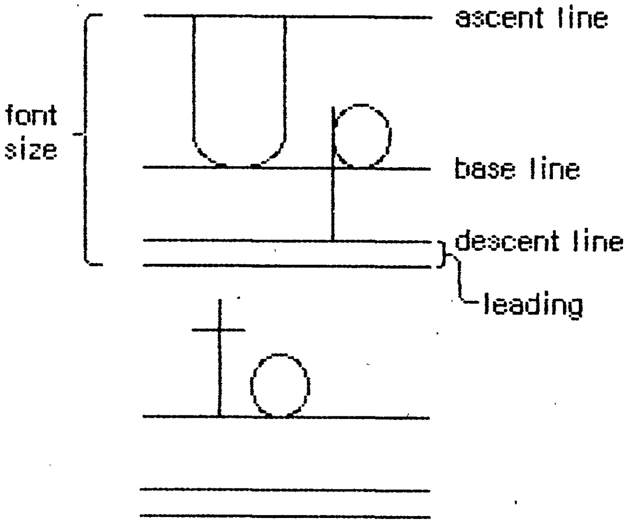


Figure 1. Font Size

(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	¶	∂	∆	÷		
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).

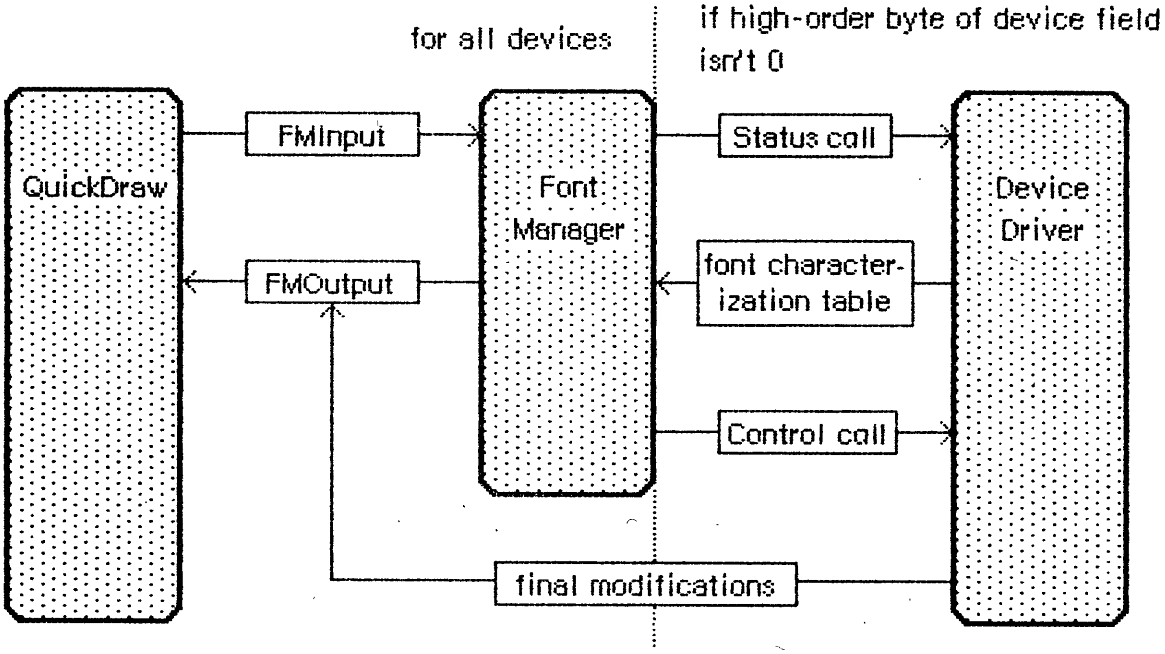


Figure 3. Communication About Fonts

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.

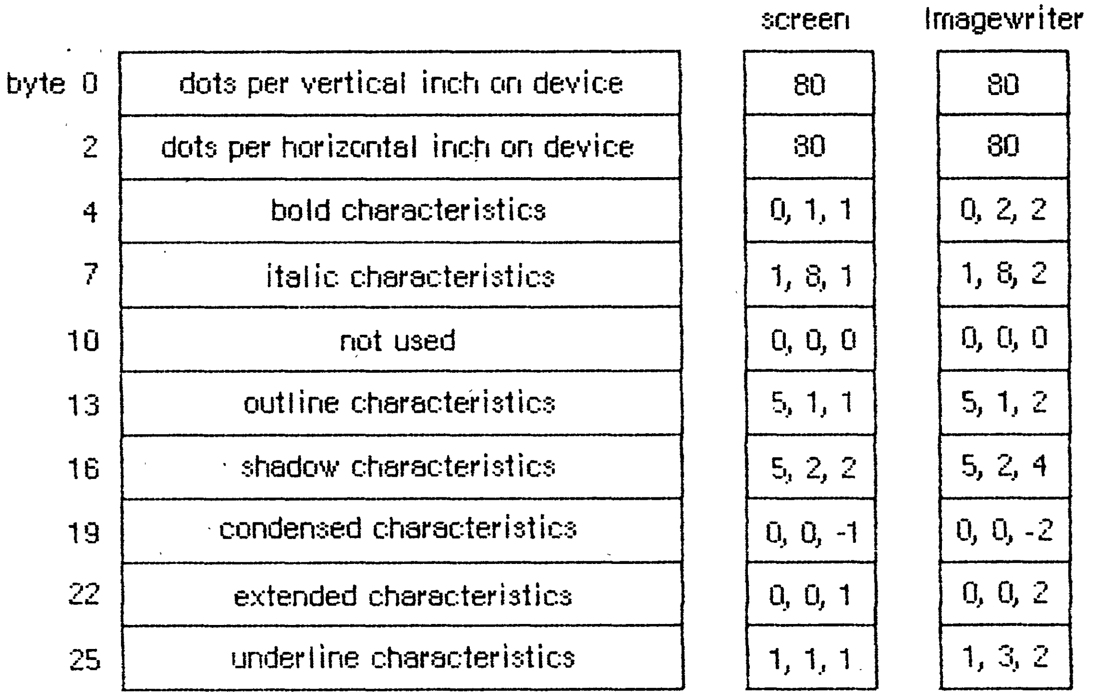
The Font Manager takes the FMInput record and asks the Resource Manager for the font. If the requested size isn't available, the Font Manager scales another size to match (as described previously).

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.



byte		screen	Imagewriter
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.

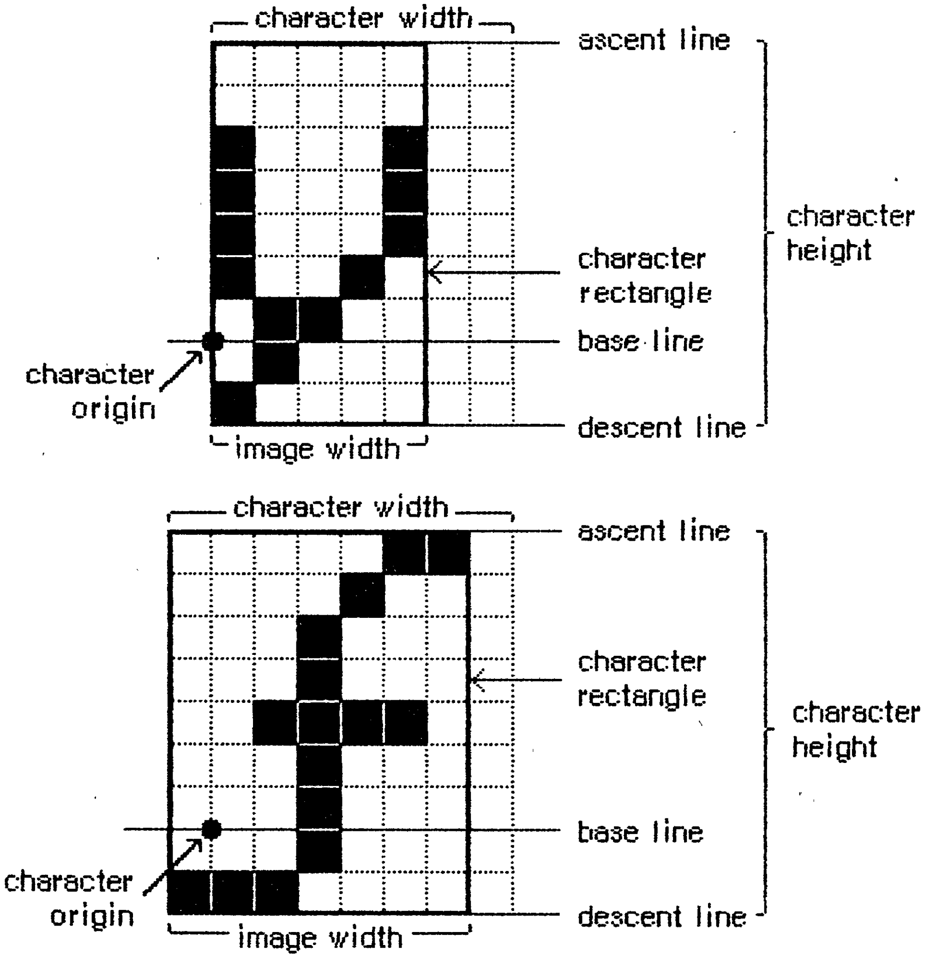


Figure 5. Character Images

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).

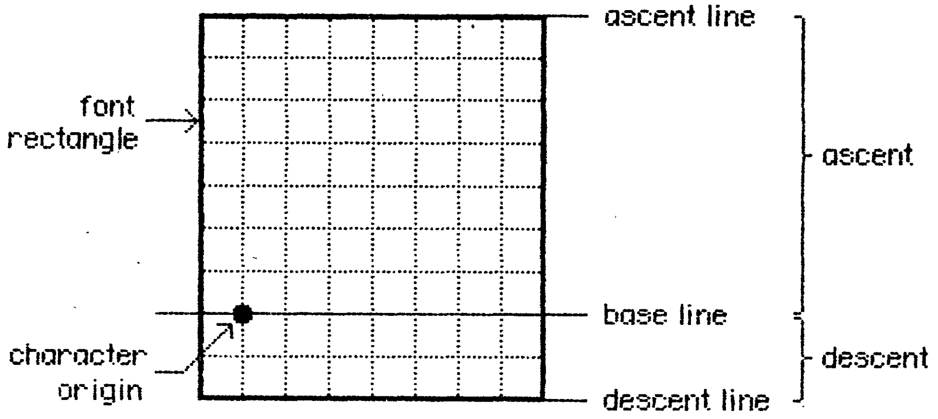


Figure 6. Features of Fonts

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.

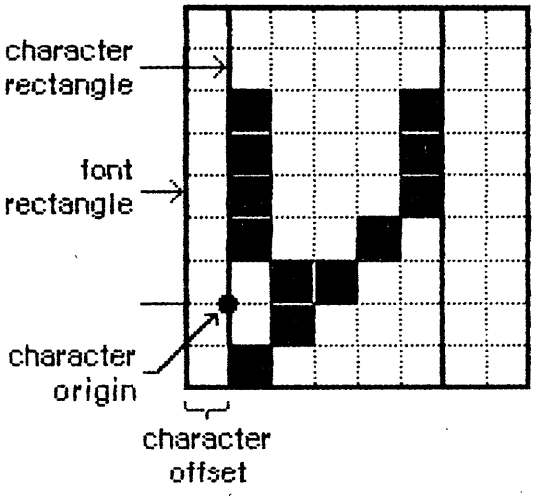


Figure 7. Character Offset

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.

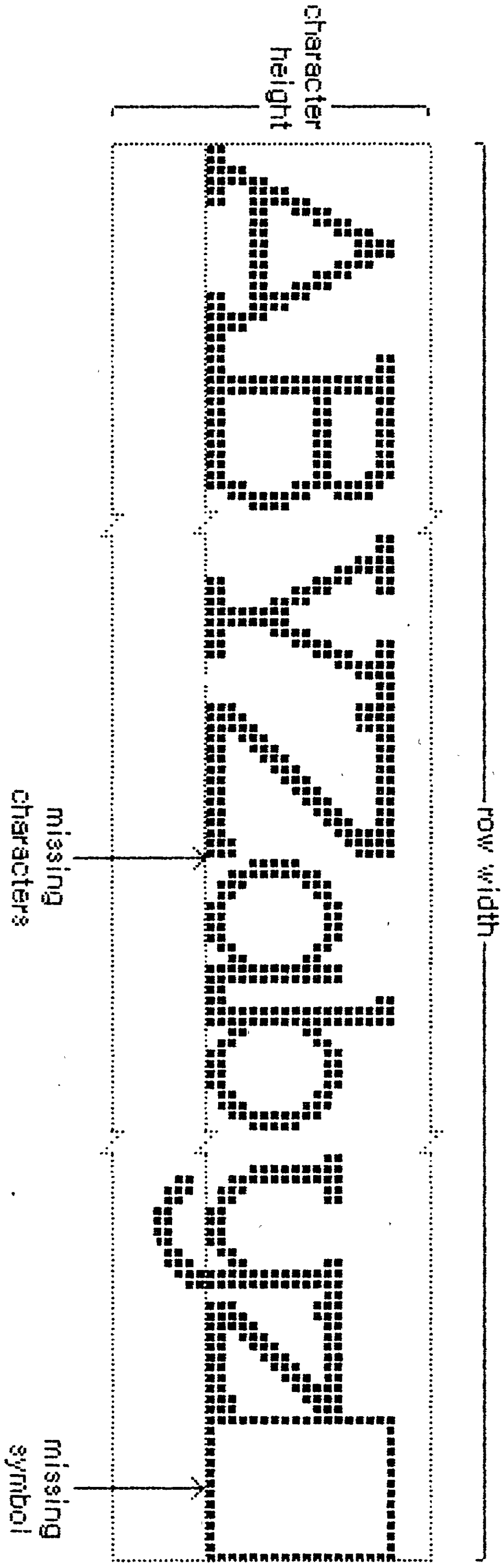


Figure 8. Partial Bit Image for a Font

Font Records

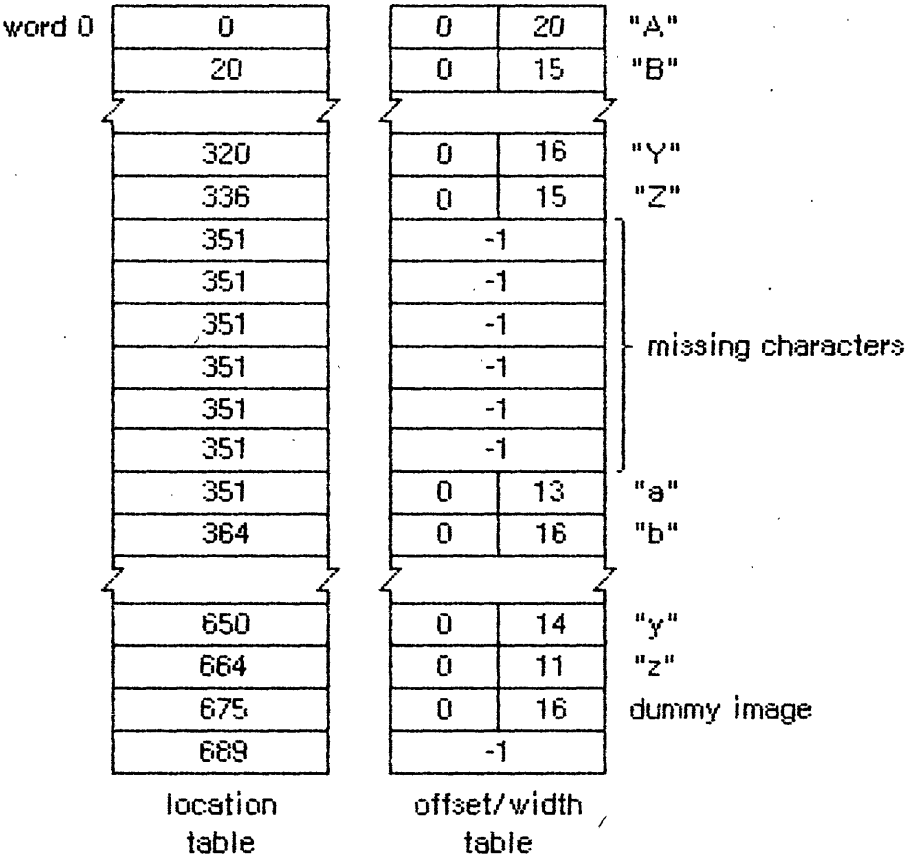
The information describing a font is contained in a data structure called a font record, which contains the following:

- 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.



	location table	offset/width table		
word 0	0	0	20	"A"
	20	0	15	"B"
	320	0	16	"Y"
	336	0	15	"Z"
	351	-1		missing characters
	351	-1		
	351	-1		
	351	-1		
	351	-1		
	351	-1		
	351	0	13	"a"
	364	0	16	"b"
	650	0	14	"y"
	664	0	11	"z"
	675	0	16	dummy image
	689	-1		

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

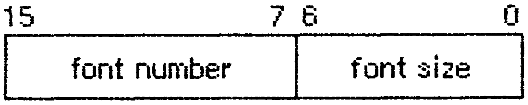


Figure 1Ø. Resource ID for a Font

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
romFont∅	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 Toolbox Event Manager: A Programmer's Guide /EMGR/EVENTS

See Also: Inside Macintosh: A Road Map
Macintosh Memory Management: An Introduction
Macintosh User Interface Guidelines
Programming Macintosh Applications in Assembly Language
The Resource Manager: A Programmer's Guide
QuickDraw: A Programmer's Guide
The Desk Manager: A Programmer's Guide
Macintosh Packages: A Programmer's Guide
The Operating System Event Manager: A Programmer's Guide
The File Manager: A Programmer's Guide
The Device Manager: A Programmer's Guide

Modification History:			
	First Draft (ROM 4)	Steve Chernicoff	6/2Ø/83
	Second Draft (ROM 7)	Caroline Rose & Brent Davis	11/19/84

ABSTRACT

This manual describes the Event Manager, the part of the Macintosh User Interface Toolbox that allows your application to monitor the user's actions, such as those involving the mouse, keyboard, and keypad. The Event Manager is also used by other parts of the Toolbox; for instance, the Window Manager uses events to coordinate the ordering and display of windows on the screen.

Summary of significant changes and additions since last draft:

- PostEvent, FlushEvents, and SetEventMask have been moved to the Operating System Event Manager manual. The section on defining a nonstandard keyboard configuration was incorrect and has been removed; an accurate version of it will be included in the next draft of the Operating System Event Manager manual.

- The changeFlag bit of the modifiers field of an event record is no longer documented; it's unreliable and should not be used.

- Functions GetDblTime and GetCaretTime have been added (page 24).

- Details have been added to GetNextEvent (page 21), GetKeys (page 24) and the sections on keyboard events (page 7), event records (page 1Ø), using the Event Manager (page 15), and journaling (page 24).

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Event Manager, the part of the Macintosh User Interface Toolbox that allows your application to monitor the user's actions, such as those involving the mouse, keyboard, and keypad. *** Eventually it will become part of the comprehensive Inside Macintosh manual. *** The Event Manager is also used by other parts of the Toolbox; for instance, the Window Manager uses events to coordinate the ordering and display of windows on the screen.

There are actually two Event Managers: one in the Operating System and one in the Toolbox. The Toolbox Event Manager calls the Operating System Event Manager and serves as an interface between it and your application; 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 the one your application will ordinarily deal 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 Operating System Event Manager manual.

(note)

> Most of the constants and data types presented in this manual are actually defined in the Operating System Event Manager; they're explained here because they're essential to understanding the Toolbox Event Manager.

Like all Toolbox documentation, this manual assumes you're familiar with Lisa Pascal and the information in the following manuals:

- Inside Macintosh: A Road Map
- Macintosh User Interface Guidelines
- Macintosh Memory Management: An Introduction
- Programming Macintosh Applications in Assembly Language, if you're using assembly language

You should also be familiar with:

- resources, as documented in the Resource Manager manual
- the basic concepts and data structures behind QuickDraw

ABOUT THE TOOLBOX EVENT MANAGER

The Toolbox Event Manager is your application's link to its user. Whenever the user presses the mouse button, types on the keyboard or keypad, or inserts a disk in a disk drive, your application is notified by means of an event. 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.

Although the Event Manager's primary purpose is to monitor the user's actions and pass them to your application in an orderly way, it also serves as a convenient mechanism for sending signals from one part of your application 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 screen. You can also define your own types of events and use them in any way you wish.

Most events waiting to be processed are kept in the event queue, where they were stored (posted) by the Operating System Event Manager. The Toolbox Event Manager retrieves events from this queue for your application and also reports other events that aren't kept in the queue, such as those related to windows. In general, events are collected from a variety of sources and reported to your application on demand, one at a time. Events aren't necessarily reported in the order they occurred; some have a higher priority than others.

There are several different types of events. You can restrict some Event Manager routines to apply only to certain event types, in effect disabling the other types.

Other operations your application can perform with Event Manager routines include:

- directly reading the current state of the keyboard, keypad, and mouse button

- monitoring the location of the mouse

- finding out how much time has elapsed since the system was last started up

The Event Manager also provides a journaling mechanism, which enables events to be fed to the Event Manager from a source other than the user.

EVENT TYPES

Events are of various types, depending on their origin and meaning. Some report actions by the user; others are generated by the Window Manager, by device drivers, or by your application itself for its own purposes. Some events are handled by the system before your application ever sees them; others are left for your application to handle in its own way.

The most important event types 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. Auto-key events are generated when the user 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 or takes any other action that requires a volume to be mounted (as described in the File Manager manual). For example, a hard disk that contains several volumes may also post a disk-inserted event.

(note)

> Mere movements of the mouse are not reported as events. If necessary, your application can keep track of them by periodically asking the Event Manager for the current location of the mouse.

The following event types are generated 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 all or part of a window's contents need to be drawn or redrawn, usually as a result of the user's opening, closing, activating, or moving a window.

Another event type (device driver event) may be generated by device drivers in certain situations; for example, a driver might be set up to report an event when its transmission of data is interrupted. The documentation for the individual drivers will tell you about any specific device driver events that may occur.

A network event may be generated by the AppleBus Manager; for details, see the AppleBus Manager manual *** (doesn't yet exist) ***.

In addition, your application can define as many as four event types of its own and use them for any desired purpose.

(note)

> You place application-defined events in the event queue with the Operating System Event Manager procedure PostEvent. See the Operating System Event Manager manual for details.

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

The event queue is a FIFO (first-in-first-out) list--that is, events are retrieved from the queue in the order they were originally posted. However, the way that various types of events are generated and detected causes some events to have higher priority than others. (Remember, not all events are kept in the event queue.) Furthermore, when you ask the Event Manager for an event, you can specify particular types that are of interest; doing so can cause some events to be passed over in favor of others that were actually posted later. The following discussion is 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 types. 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, network, device driver, application-defined (all in FIFO order)

3. auto-key

4. update (in front-to-back order of windows)

5. null

Activate events take priority over all others; they're 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; at most there will be one for a window becoming inactive followed by another for a window becoming active.

Category 2 includes most of the event types. Within this category, events are retrieved from the queue in the order they were posted.

If no event is available in categories 1 and 2, the Event Manager reports an auto-key event if the appropriate conditions hold for one. (These conditions are described in detail in the next section.)

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 drawn. 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.

(note)

> The event queue has a capacity of 2Ø events. If the queue should become full, the Operating System Event Manager will begin discarding old events to make room for new ones as they're posted. The events discarded are always the oldest ones in the queue. Advanced programmers can configure the capacity of the event queue in the system startup information stored on a volume; for more information, see the section "Data Organization on Volumes" in the File Manager manual. *** No more information is given there yet, but it will be included in the next draft of that manual. ***

KEYBOARD EVENTS

The character keys on the Macintosh keyboard and optional keypad generate key-down and key-up events when pressed and released; this includes all keys except Shift, Caps Lock, Command, and Option, which are called modifier keys. (Modifier keys are treated specially, as described below, and generate no keyboard events of their own.) In addition, an auto-key event is posted whenever all of the following conditions apply:

- Auto-key events haven't been disabled. (This is discussed further under "Event Masks" below.)
- No higher-priority event is available.
- The user is currently holding down a character key.
- The appropriate time interval (see below) has elapsed since the last key-down or auto-key event.

Two different time intervals are associated with auto-key events. The first auto-key event is generated after a certain initial delay has elapsed since the original key-down event (that is, since the key was originally pressed); this is called the auto-key threshold. Subsequent auto-key events are then generated each time a certain repeat interval has elapsed since the last such event; this is called the auto-key rate. The default values are 16 ticks (sixtieths of a second) for the auto-key threshold and four ticks for the auto-key rate. The user can change these values with the Control Panel desk accessory, by adjusting the keyboard touch and the rate of repeating keys.

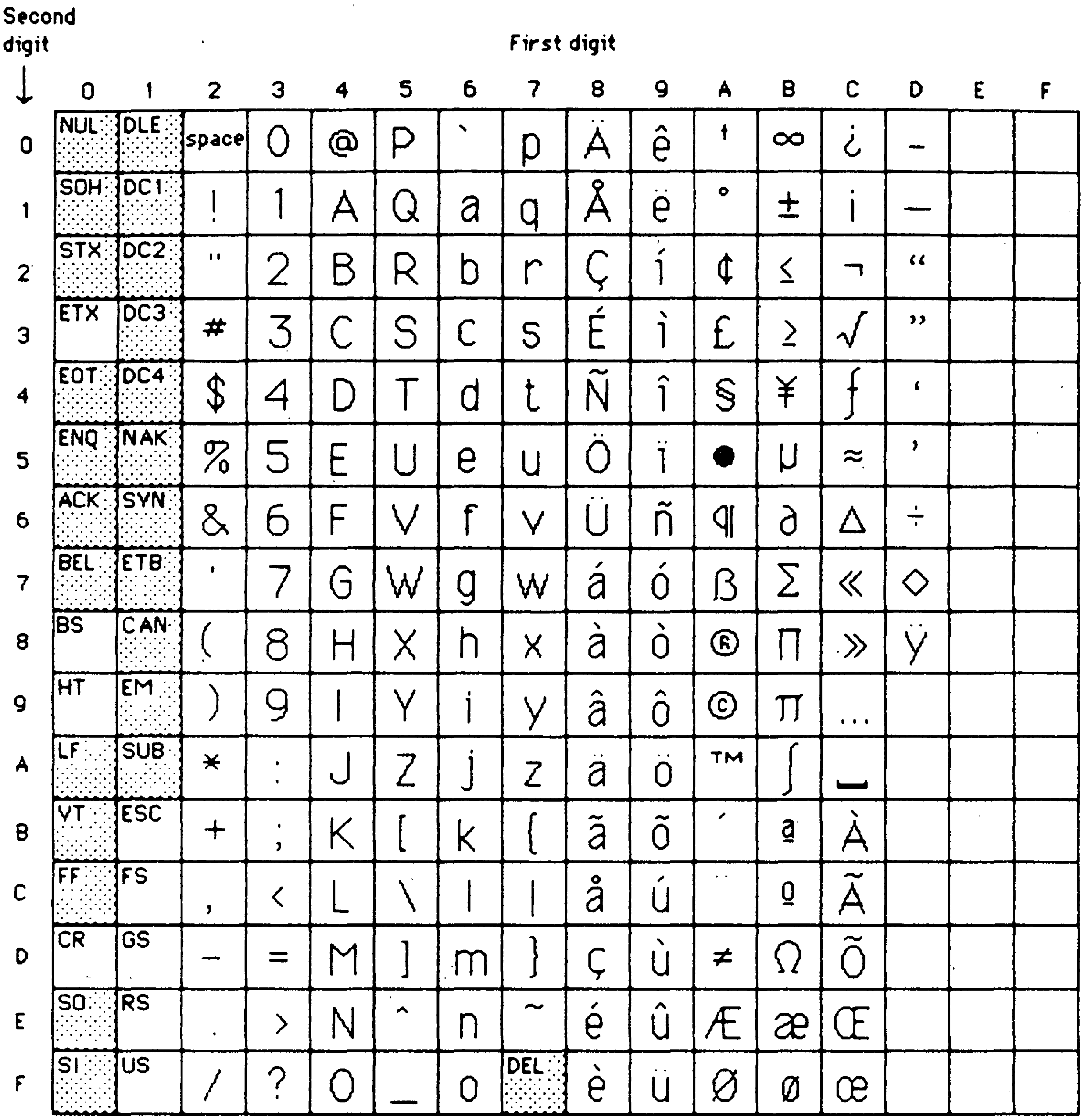
Assembly-language note: The current values for the auto-key threshold and rate are stored in the global variables KeyThresh and KeyRepThresh, respectively.

When the user presses, holds down, or releases a character key, the character generated by that key is identified internally with a *character code*. Character codes are given in the extended version of ASCII (the American Standard Code for Information Interchange) used by the Macintosh. A table showing the character codes for the standard Macintosh character set appears in Figure 1 on the following page. All character codes are given in hexadecimal in this table. The first digit of a character's hexadecimal value is shown at the top of the table, the second down the left side. For example, character code $47 stands for "G", which appears in the table at the intersection of column 4 and row 7.

Macintosh, the owner's guide, describes the method of generating the printing characters (codes $20 through $D8) shown in Figure 1. Notice that in addition to the regular space character ($20) there's a nonbreaking space ($CA), which is generated by pressing the space bar with the Option key down.

Nonprinting or "control" characters ($00 through $1F, as well as $7F) are identified in the table by their traditional ASCII abbreviations; those that are shaded have no special meaning on the Macintosh and cannot normally be generated from the Macintosh keyboard or keypad. Those that can be generated are listed below along with the method of generating them:

Code	Abbreviation	Key
$03	ETX	Enter key on keyboard or keypad
$08	BS	Backspace key on keyboard
$09	HT	Tab key on keyboard
$0D	CR	Return key on keyboard
$1B	ESC	Clear key on keypad
$1C	FS	Left arrow key on keypad
$1D	GS	Right arrow key on keypad
$1E	RS	Up arrow key on keypad
$1F	US	Down arrow key on keypad



First digit

Second digit ↓	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	space	0	@	P	`	p	Ä	ê	†	∞	¿	–		
1	SOH	DC1	!	1	A	Q	a	q	Å	ë	°	±	¡	—		
2	STX	DC2	"	2	B	R	b	r	Ç	í	¢	≤	¬	“		
3	ETX	DC3	#	3	C	S	c	s	É	ì	£	≥	√	”		
4	EOT	DC4	$	4	D	T	d	t	Ñ	î	§	¥	ƒ	‘		
5	ENQ	NAK	%	5	E	U	e	u	Ö	ï	•	µ	≈	’		
6	ACK	SYN	&	6	F	V	f	v	Ü	ñ	¶	∂	∆	÷		
7	BEL	ETB	'	7	G	W	g	w	á	ó	ß	∑	«	◇		
8	BS	CAN	(	8	H	X	h	x	à	ò	®	∏	»	ÿ		
9	HT	EM	)	9	I	Y	i	y	â	ô	©	π	…			
A	LF	SUB	*	:	J	Z	j	z	ä	ö	™	∫	⎵			
B	VT	ESC	+	;	K	[	k	{	ã	õ	´	ª	À			
C	FF	FS	,	<	L	\	l	|	å	ú	¨	º	Ã			
D	CR	GS	–	=	M	]	m	}	ç	ù	≠	Ω	Õ			
E	SO	RS	.	>	N	^	n	~	é	û	Æ	æ	Œ			
F	SI	US	/	?	O	_	o	DEL	è	ü	Ø	ø	œ			

⎵ stands for a nonbreaking space, the same width as a digit.

The shaded characters cannot normally be generated from the Macintosh keyboard or keypad.

Figure 1. Macintosh Character Set

The association between characters and keys on the keyboard or the keypad is defined by a keyboard configuration, which is a resource stored in a resource file. The particular character that's generated by a character key depends on three things:

- the character key being pressed
- which, if any, of the modifier keys were held down when the character key was pressed
- the keyboard configuration currently in effect

The modifier keys, instead of generating keyboard events themselves, modify the meaning of the character keys by changing the character codes that those keys generate. For example, under the standard U.S. keyboard configuration, the "C" key generates any of the following, depending on which modifier keys are held down:

Key(s) pressed	Character generated
"C" by itself	Lowercase c
"C" with Shift or Caps Lock down	Capital C
"C" with Option down	Lowercase c with a cedilla (ç), used in foreign languages
"C" with Option and Shift down, or with Option and Caps Lock down	Capital C with a cedilla (Ç)

The state of each of the modifier keys is also reported in a field of the event record (see next section), where your application can examine it directly.

(note)

> As described in the owner's guide, some accented characters are generated by pressing Option along with another key for the accent, and then typing the character to be accented. In these cases, a single key-down event occurs for the accented character; there's no event corresponding to the typing of the accent.

Under the standard keyboard configuration only the Shift, Caps Lock, and Option keys actually modify the character code generated by a character key on the keyboard; the Command key has no effect on the character code generated. 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 event record field containing the state of the modifier keys.

Normally you'll just want to use the standard keyboard configuration, which is read from the system resource file every time the system is started up. Other keyboard configurations can be used for nonstandard layouts. In rare cases, you may want to define your own keyboard configuration to suit your application's special needs. You can make the Command key affect the character code generated (or, when a key is pressed on the keypad, the Caps Lock or Option key). For information on how to install an alternate keyboard configuration or define one of your own, see the Operating System Event Manager manual *** (the information isn't yet in that manual; it will be in the next draft) ***.

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 (in ticks since system startup)
- the location of the mouse at the time the event was posted (in global coordinates)
- 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

Every event has an event record containing this information--even null events.

Event records are defined as follows:

```
TYPE EventRecord = RECORD
                     what:       INTEGER;   {event code}
                     message:    LONGINT;   {event message}
                     when:       LONGINT;   {ticks since startup}
                     where:      Point;     {mouse location}
                     modifiers:  INTEGER    {modifier flags}
                   END;
```

The when field contains the number of ticks since the system was last started up, and the where field gives the location of the mouse, in global coordinates, at the time the event was posted. The other three fields are described below.

Event Code

The what field of an event record contains an event code identifying the type of the event. The event codes are available 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}
      networkEvt  = 1Ø; {network}
      driverEvt   = 11; {device driver}
      app1Evt     = 12; {application-defined}
      app2Evt     = 13; {application-defined}
      app3Evt     = 14; {application-defined}
      app4Evt     = 15; {application-defined}
```

Event Message

The message field of an event record contains the event message, which conveys additional important information about the event. The nature of this information depends on the event type, as summarized in the following table and described below.

Event type	Event message
Keyboard	Character code and key code in low-order word
Activate, update.	Pointer to window
Disk-inserted	Drive number in low-order word, File Manager result code in high-order word
Mouse-down, mouse-up, null	Meaningless
Network	See AppleBus Manager manual
Device driver	See driver documentation
Application-defined	Whatever you wish

For keyboard events, only the low-order word of the event message is used, as shown in Figure 2. The low-order byte of this word contains the ASCII character code generated by the key or combination of keys that was pressed or released; usually this is all you'll need.

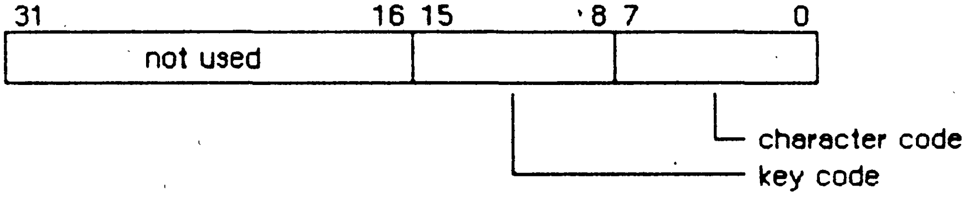
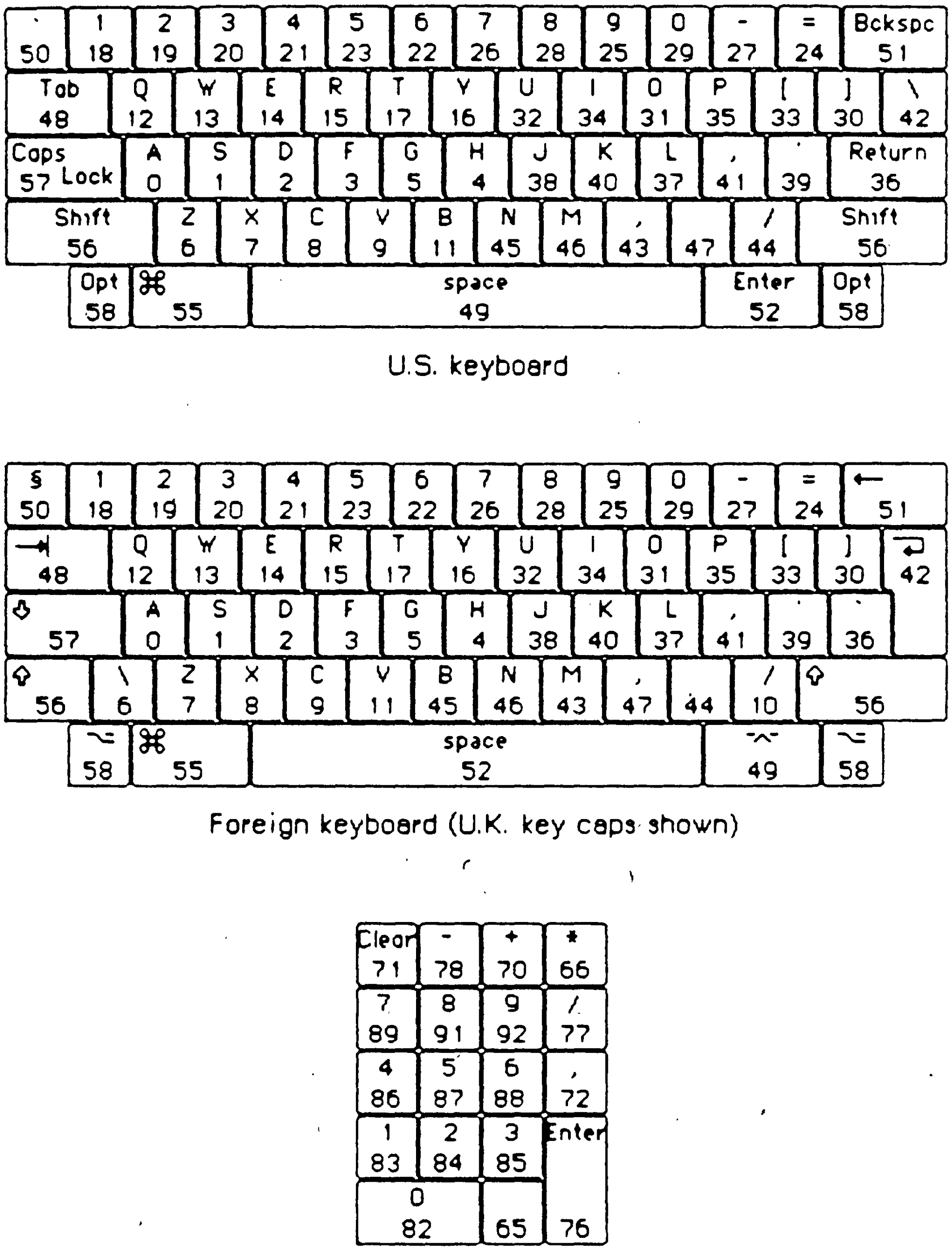


Figure 2. Event Message for Keyboard Events

The key code in the event message for a keyboard event is an integer representing the character key that was pressed or released; this value is always the same for any given character key, regardless of the modifier keys pressed along with it. Key codes are useful in special cases--in a music generator, for example--where you want to treat the keyboard as a set of keys unrelated to characters. Figure 3 gives the key codes for all the keys on the keyboard and keypad. (Key codes are shown for modifier keys here because they're meaningful in other contexts, as explained later.) Both the U.S. and foreign keyboards are shown; in some cases the codes are quite different (for example, space and Enter are reversed).



U.S. keyboard

Foreign keyboard (U.K. key caps shown)

Keypad (both U.S. and foreign)

Figure 3. Key Codes

The following predefined constants are available to help you access the character code and key code:

```
      CONST charCodeMask = $000000FF;   {character code}
            keyCodeMask  = $0000FF00;   {key code}
```

(note)

> You can use the Toolbox Utility function BitAnd with these constants; for instance, to access the character code, use
>
> ```
> charCode := BitAnd(myEvent.message,charCodeMask)
> ```

For activate and update events, the event message is a pointer to the window affected. (If the event is an activate event, additional important information about the event can be found in the modifiers field of the event record, as described below.)

For disk-inserted events, the low-order word of the event message contains the drive number of the disk drive into which the disk was inserted: 1 for the Macintosh's built-in drive, and 2 for the external drive, if any. Numbers greater than 2 denote additional disk drives connected through one of the two serial ports. By the time your application receives a disk-inserted event, the system will already have attempted to mount the volume on the disk by calling the File Manager function MountVol; the high-order word of the event message will contain the result code returned by MountVol.

For mouse-down, mouse-up, and null events, the event message is meaningless and should be ignored. For network and device driver events, the contents of the event message depend on the situation under which the event was generated; the documentation describing those situations will give the details. Finally, you can use the event message however you wish for application-defined event types.

Modifier Flags

As stated above, the modifiers field of an event record contains further information about activate events and the state of the modifier keys and mouse button at the time the event was posted (see Figure 4). You might look at this field to find out, for instance, whether the Command key was down when a mouse-down event was posted (which in many applications affects the way objects are selected) or when a key-down event was posted (which could mean the user is choosing a menu item by typing its keyboard equivalent).

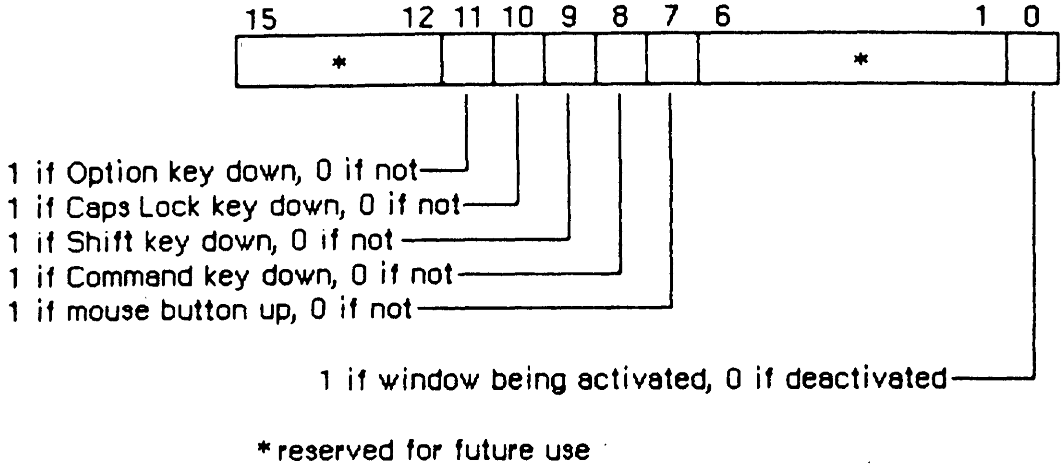

15
12
11
10
9
8
7
6
1
0
*
*
1 if Option key down, 0 if not
1 if Caps Lock key down, 0 if not
1 if Shift key down, 0 if not
1 if Command key down, 0 if not
1 if mouse button up, 0 if not
1 if window being activated, 0 if deactivated
*reserved for future use

Figure 4. Modifier Flags

The following predefined constants are useful as masks for reading the flags in the modifiers field:

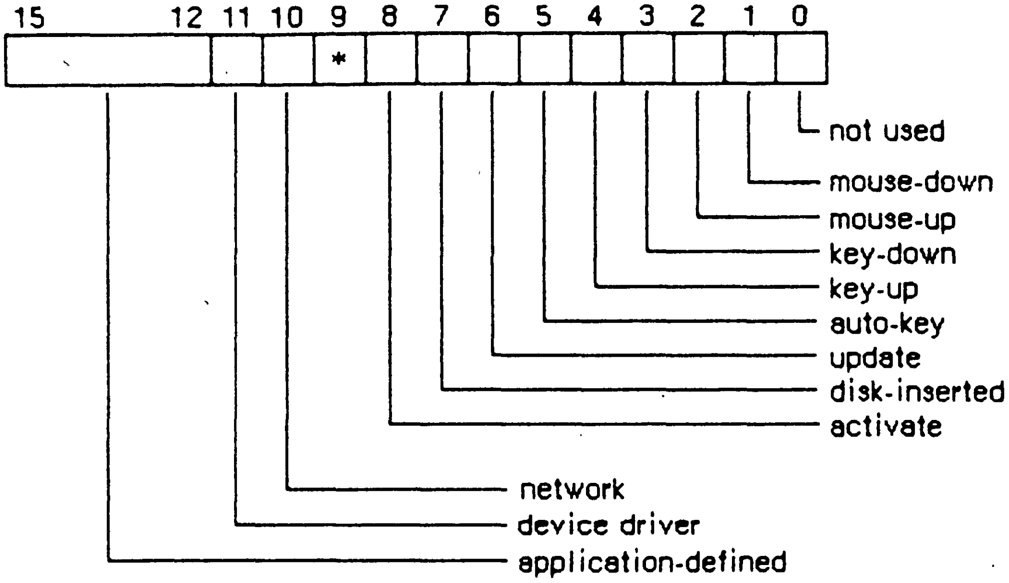
```
CONST activeFlag = 1;      {set if window being activated}
      btnState   = 128;    {set if mouse button up}
      cmdKey     = 256;    {set if Command key down}
      shiftKey   = 512;    {set if Shift key down}
      alphaLock  = 1Ø24;   {set if Caps Lock key down}
      optionKey  = 2Ø48;   {set if Option key down}
```

The activeFlag bit gives further information about activate events; it's set if the window pointed to by the event message is being activated, or Ø if the window is being deactivated. The remaining bits indicate the state of the mouse button and modifier keys. Notice that the btnState bit is set if the mouse button is **up**, whereas the bits for the four modifier keys are set if their corresponding keys are **down.**

EVENT MASKS

Some of the Event Manager routines can be restricted to operate on a specific event type or group of types; in other words, the specified event types are enabled while all others are disabled. For instance, instead of just requesting the next available event, you can specifically ask 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 there's one bit position for each event type, as shown in Figure 5. The bit position representing a given type corresponds to the event code for that type--for example, update events (event code 6) are specified by bit 6 of the mask. A 1 in bit 6 means that this Event Manager call applies to update events; a Ø means that it doesn't.



*reserved for future use

Figure 5. Event Mask

Masks for each individual event type are available as predefined constants:

```
CONST 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}
      networkMask = 1Ø24;    {network}
      driverMask  = 2Ø48;    {device driver}
      app1Mask    = 4Ø96;    {application-defined}
      app2Mask    = 8192;    {application-defined}
      app3Mask    = 16384;   {application-defined}
      app4Mask    = -32768;  {application-defined}
```

(note)

Null events can't be disabled; a null event will always be reported when none of the enabled types of events are available.

The following predefined mask designates all event types:

```
CONST everyEvent = -1;       {all event types}
```

You can form any mask you need by adding or subtracting these mask constants. For example, to specify every keyboard event, use

```
       keyDownMask + keyUpMask + autoKeyMask
```

For every event except an update, use

```
       everyEvent - updateMask
```

(note)

> It's recommended that you always use the event mask everyEvent unless there's a specific reason not to.

There's also a global system event mask that controls which event types get posted into the event queue. Only event types corresponding to bits set in the system event mask are posted; all others 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's initialized to

```
       everyEvent - keyUpMask
```

(note)

> Key-up events are meaningless for most applications. Your application will usually want to ignore them; if not, it can set the system event mask with the Operating System Event Manager procedure SetEventMask.

USING THE TOOLBOX EVENT MANAGER

Before using the Event Manager, you should initialize the Window Manager by calling its 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. Initializing the Window Manager requires you to have initialized QuickDraw and the Font Manager.

> Assembly-language note: If you want to use events but not windows, set the global variable WindowList to Ø instead of calling InitWindows.

It's also usually a good idea to issue the Operating System Event Manager call FlushEvents(everyEvent,Ø) to empty the event queue of any stray events left over from before your application was started up (such as keystrokes typed to the Finder). See the Operating System Event Manager manual for a description of FlushEvents.

Most Macintosh application programs are event-driven. Such programs have a main loop that repeatedly calls GetNextEvent to retrieve the next available event, and then uses a CASE statement to take whatever action is appropriate for each type of event; some typical responses to commonly occurring events are described below. Your program is expected to respond only to those events that are directly related to its own operations. After calling GetNextEvent, you should test its Boolean result to find out whether your application needs to respond to the event: TRUE means the event may be of interest to your application; FALSE usually means it will not be of interest.

In some cases, you may simply want to look at a pending event while leaving it available for subsequent retrieval by GetNextEvent. You can do this with the EventAvail function.

Responding to Mouse Events

On receiving a mouse-down event, your application should first call the Window Manager function FindWindow to find out where on the screen the mouse button was pressed, and then respond in whatever way is appropriate. Depending on the part of the screen in which the button was pressed, 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 manuals for details.

If your application attaches some special significance to pressing a modifier key along with the mouse button, you can discover the state of that modifier key while the mouse button is down by examining the appropriate flag in the modifiers field.

If you're using the TextEdit part of the Toolbox to handle text editing, mouse double-clicks will work automatically as a means of selecting a word; to respond to double-clicks in any other context, however, you'll have to detect them yourself. You can do so by comparing the time and location of a mouse-up event with those of the immediately following mouse-down event. You should assume a double-click has occurred if both of the following are true:

- The times of the mouse-up event and the mouse-down event differ by a number of ticks less than or equal to the value returned by the Event Manager function GetDblTime.

- The locations of the two mouse-down events separated by the mouse-up event are sufficiently close to each other. Exactly what this means depends on the particular application. For instance, in a word-processing application, you might consider the two locations essentially the same if they fall on the same character, whereas in a graphics application you might consider them essentially the same if the sum of the horizontal and vertical changes in position is no more than five pixels.

Mouse-up events may be significant in other ways; for example, they might signal the end of dragging in a graphics or spreadsheet application. Many simple applications, however, will ignore mouse-up events.

Responding to Keyboard Events

For a key-down event, 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. (See the Menu Manager manual for details.)

If the key-down event was not a menu command, you should then respond to the event in whatever way is appropriate for your application. For example, if one of your windows is active, you might want to insert the typed character into the active document; if none of your windows is active, you might want to ignore the event.

Usually your application can handle auto-key events the same as key-down events. You may, however, want to ignore auto-key events that invoke commands that shouldn't be continually repeated.

(note)

> Remember that most applications will want to ignore key-up events; with the standard system event mask you won't get any.

If you wish to periodically inspect the state of the keyboard or keypad --say, while the mouse button is being held down--use the procedure GetKeys; this procedure is also the only way to tell whether a modifier key is being pressed alone.

Responding to Activate and Update Events

When your application receives an activate event for one of its 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 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 its own windows, your application should usually call the Window Manager procedure BeginUpdate, draw the window's contents, and then call EndUpdate. See the Window Manager manual for important additional information on activate and update events.

Responding to Disk-Inserted Events

Most applications will use the Standard File Package, which responds to disk-inserted events for you during standard file saving and opening; you'll usually want to ignore any other disk-inserted events, such as the user's inserting a disk when not expected. If, however, you do want to respond to other disk-inserted events, or if you plan not to use the Standard File Package, then you'll have to handle such events yourself.

When you receive a disk-inserted event, the system will already have attempted to mount the volume on the disk by calling the File Manager function MountVol. You should examine the result code returned by the File Manager in the message field of the event record. If the result code indicates that the attempt to mount the volume was unsuccessful, you might want to take some special action, such as calling the Disk Initialization Package function DIBadMount. See the File Manager and Macintosh Packages manuals for further details.

Other Operations

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 moves it with the button down, use StillDown or WaitMouseUp.

The function TickCount returns the number of ticks since the last system startup; you might, for example, compare this value to the when field of an event record to discover the delay since that event was posted.

Finally, the function GetCaretTime returns the number of ticks between blinks of the "caret" (usually a vertical bar) marking the insertion point in editable text. You should call GetCaretTime if you aren't using TextEdit and therefore need to cause the caret to blink yourself. You would check this value each time through your program's main event loop, to ensure a constant frequency of blinking.

TOOLBOX EVENT MANAGER ROUTINES

Accessing Events

```
FUNCTION GetNextEvent (eventMask: INTEGER; VAR theEvent: EventRecord) :
            BOOLEAN;
```

GetNextEvent returns the next available event of a specified type or types and, if the event is in the event queue, removes it from the queue. The event is returned as the value of the parameter theEvent. The eventMask parameter specifies which event types are of interest. GetNextEvent returns the next available event of any type designated by the mask, subject to the priority rules discussed above under "Priority of Events". If no event of any of the designated types is available, GetNextEvent returns a null event.

(note)

> Events in the queue that aren't designated in the mask are kept in the queue; if you want to remove them, you can do so by calling the Operating System Event Manager procedure FlushEvents.

Before reporting an event to your application, 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; a function result of TRUE means that your application should handle the event itself. The Desk Manager intercepts the following events:

- activate and update events directed to a desk accessory
- keyboard events if the currently active window belongs to a desk accessory

(note)

> In each case, the event is intercepted by the Desk Manager only if the desk accessory can handle that type of event; however, as a rule all desk accessories should be set up to handle activate, update, and keyboard events.

The Desk Manager also intercepts disk-inserted events: It attempts to mount the volume on the disk by calling the File Manager function MountVol. GetNextEvent will always return TRUE in this case, though, so that your application can take any further appropriate action after examining the result code returned by MountVol in the event message. (See the Desk Manager and File Manager manuals for further details.) GetNextEvent returns TRUE for all other non-null events (including all mouse-down events, regardless of which window is active), leaving them for your application to handle.

Assembly-language note: If for some reason you don't want GetNextEvent to call SystemEvent, set the global variable SEvtEnb to Ø.

GetNextEvent also makes the following processing happen, invisible to your program:

- If the "alarm" is set and the current time is the alarm time, the alarm goes off (a beep followed by blinking the title of the Apple menu). The user can set the alarm with the Alarm Clock desk accessory.
- If the user holds down the Command and Shift keys while pressing a numeric key that has a special effect, that effect occurs. The standard such keys are 1 and 2 for ejecting the disk in the internal or external drive, and 3 and 4 for writing a snapshot of the screen to a MacPaint document or to the printer.

(note)

Advanced programmers can implement their own code to be executed in response to Command-Shift-number combinations (except for Command-Shift-1 and 2, which can't be changed). The code corresponding to a particular number must be a routine having no parameters, stored in a resource whose type is 'FKEY' and whose ID is the number. The system resource file contains code for the numbers 3 and 4.

Assembly-language note: You can disable GetNextEvent's processing of Command-Shift-number combinations by setting the global variable ScrDmpEnb to Ø.

```
FUNCTION EventAvail (eventMask: INTEGER; VAR theEvent: EventRecord) :
          BOOLEAN;
```

EventAvail works exactly the same as GetNextEvent except that if the event is in the event queue, it's left there.

(note)

An event returned by EventAvail will not be accessible later if in the meantime the queue becomes full and the event is discarded from it; since the events discarded are always the oldest ones in the queue, however, this will happen only in an unusually busy environment.

Reading the Mouse

PROCEDURE GetMouse (VAR mouseLoc: Point);

GetMouse returns the current mouse location in the mouseLoc parameter. The location is given 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; that location is always in global coordinates.

FUNCTION Button : BOOLEAN;

The Button function returns TRUE if the mouse button is currently down, and FALSE if it isn't.

FUNCTION StillDown : BOOLEAN;

Usually 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 pending in the event queue. This is a true test of whether the button is still down from the original press--unlike Button (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 (above), except that if the button is not still down from the original press, WaitMouseUp removes the preceding mouse-up event before returning FALSE. If, for instance, your application attaches some special significance both to mouse double-clicks and to mouse-up events, this function would allow your application to recognize a double-click without being confused by the intervening mouse-up.

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 [Ø..127] OF BOOLEAN;
```

Each key on the keyboard or keypad corresponds to an element in the keyMap. The index into the keyMap for a particular key is the same as the key code for that key. (The key codes are shown in Figure 3 above.) The keyMap element is TRUE if the corresponding key is down and FALSE if it isn't. The maximum number of keys that can be down simultaneously is two character keys plus any combination of the four modifier keys.

Miscellaneous Routines

FUNCTION TickCount : LONGINT;

TickCount returns the current number of ticks (sixtieths of a second) since the system was last started up.

Assembly-language note: The value returned by this function is contained in the global variable Ticks.

FUNCTION GetDblTime : LONGINT; [No trap macro]

GetDblTime returns the suggested maximum difference (in ticks) that should exist between the times of a mouse-up event and a mouse-down event for those two mouse clicks to be considered a double-click. The user can adjust this value by means of the Control Panel desk accessory.

Assembly-language note: This value is available to assembly-language programmers in the global variable DoubleTime.

FUNCTION GetCaretTime : LONGINT; [No trap macro]

GetCaretTime returns the time (in ticks) between blinks of the "caret" (usually a vertical bar) marking an insertion point in editable text. If you aren't using TextEdit, you'll need to cause the caret to blink yourself; on every pass through your program's main event loop, you should check this value against the elapsed time since the last blink of the caret. The user can adjust this value by means of the Control Panel desk accessory.

Assembly-language note: This value is available to assembly-language programmers in the global variable CaretTime.

THE JOURNALING MECHANISM

So far, this manual has talked about the Event Manager as responding to events generated by users--keypresses, mouse clicks, disk insertions, and so on. By using the Event Manager's journaling mechanism, though, you can "decouple" the Event Manager from the user and feed it events from some other source. Such a source might be a file into which have been recorded all the events that occurred during some portion of a user's session with the Macintosh. This section describes the journaling mechanism briefly and gives some examples of its use; then, if you wish, you can read on to learn the technical information necessary to use it yourself.

(note)

> The journaling mechanism can be accessed only through assembly language; Pascal programmers may want to skip this discussion.

In the usual sense, "journaling" means the recording of a sequence of user-generated events into a file; specifically, this file is a recording 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 journaling device driver, which records the call (and the results of the call) in a file. When the journal is played back, these recorded Event Manager calls are taken from the journal file and sent directly to the Event Manager. The result is that the recorded sequence of user-generated events is reproduced when the journal is played back.

The Macintosh Guided Tour is an example of such a journal. It was recorded using the Journal desk accessory, a special device driver that's available to users who want to record standard journal files for the purpose of, say, making their own Guided Tours. For more information about the Journal desk accessory, see A Guide to Making Guided Tours *** (forthcoming from Macintosh User Education) ***.

Using the journaling mechanism need not involve a file. Before Macintosh was introduced, Macintosh Software Engineering created a special desk accessory of its own for testing Macintosh software. This desk accessory, which was based on the journaling mechanism, didn't use a file--it generated events randomly, putting Macintosh "through its paces" for long periods of time without requiring a user's attention.

So, the Event Manager's journaling mechanism has a much broader utility than a mechanism simply for "journaling" as it's normally defined. With the journaling mechanism, you can decouple the Event Manager from the user and feed it events from a journaling device driver of your own design. Figure 6 illustrates what happens when the journaling mechanism is off, in recording mode, and in playback mode.

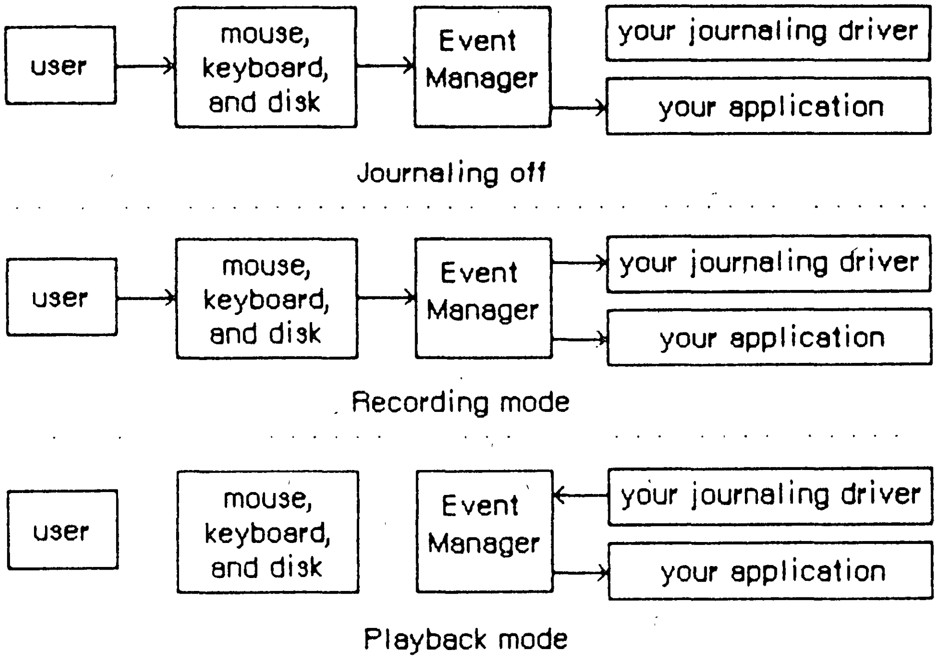


Figure 6. The Journaling Mechanism

Writing Your Own Journaling Device Driver

If you want to implement journaling in a new way, you'll need to write your own journaling device driver. Details about how to do this are given below; however, you must already have read about writing your own device driver in the Device Manager manual. Furthermore, if you want to implement your journaling device driver as a desk accessory, you'll have to be familiar with details given in the Desk Manager manual.

Whenever a call is made to any of the Event Manager routines GetNextEvent, EventAvail, GetMouse, Button, GetKeys, and TickCount, the information returned by the routine is passed to the journaling device driver by means of a Control call. The routine makes the Control call to the journaling device driver with the reference number stored in the global variable JournalRef, so be sure ahead of time that JournalRef contains the reference number of your own journaling device driver.

(note)

> The reference number of the standard journaling device driver is -2 and is available as the global constant jRefNum.

You control whether the journaling mechanism is playing or recording by setting the global variable JournalFlag to a negative or positive value. Before the Event Manager routine makes the Control call, it copies one of the following global constants into the csCode parameter of the Control call, depending on the value of JournalFlag:

JournalFlag	Value of csCode		Meaning
Negative	jPlayCtl	.EQU 16	Journal in playback mode
Positive	jRecordCtl	.EQU 17	Journal in recording mode

If you set the value of JournalFlag to Ø, the Control call won't be made at all.

Before the Event Manager routine makes the Control call, it copies into csParam a pointer to the actual data being polled by the routine (for example, a pointer to a keyMap for GetKeys, or a pointer to an event record for GetNextEvent). It also copies, into csParam+4, a journal code designating which routine is making the call:

Control call made during:	CsParam contains pointer to:	Journal code at csParam+4:		
TickCount	long integer	jcTickCount	.EQU	Ø
GetMouse	point	jcGetMouse	.EQU	1
Button	Boolean	jcButton	.EQU	2
GetKeys	keyMap	jcGetKeys	.EQU	3
GetNextEvent	event record	jcEvent	.EQU	4
EventAvail	event record	jcEvent	.EQU	4

SUMMARY OF THE TOOLBOX EVENT MANAGER

Constants

```
CONST { Event codes }

      nullEvent    = 0;   {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}
      networkEvt   = 10;  {network}
      driverEvt    = 11;  {device driver}
      app1Evt      = 12;  {application-defined}
      app2Evt      = 13;  {application-defined}
      app3Evt      = 14;  {application-defined}
      app4Evt      = 15;  {application-defined}

      { Masks for accessing keyboard event message }

      charCodeMask = $000000FF;  {character code}
      keyCodeMask  = $0000FF00;  {key code}

      { Masks for forming event mask }

      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}
      networkMask  = 1024;    {network}
      driverMask   = 2048;    {device driver}
      app1Mask     = 4096;    {application-defined}
      app2Mask     = 8192;    {application-defined}
      app3Mask     = 16384;   {application-defined}
      app4Mask     = -32768;  {application-defined}
      everyEvent   = -1;      {all event types}
```

```
{ Modifier flags in event record }

activeFlag = 1;      {set if window being activated}
btnState   = 128;    {set if mouse button up}
cmdKey     = 256;    {set if Command key down}
shiftKey   = 512;    {set if Shift key down}
alphaLock  = 1Ø24;   {set if Caps Lock key down}
optionKey  = 2Ø48;   {set if Option key down}
```

Data Types

```
TYPE EventRecord = RECORD
                     what:       INTEGER;   {event code}
                     message:    LONGINT;   {event message}
                     when:       LONGINT;   {ticks since startup}
                     where:      Point;     {mouse location}
                     modifiers: INTEGER     {modifier flags}
                   END;

     KeyMap = PACKED ARRAY [Ø..127] OF BOOLEAN;
```

Routines

Accessing Events

```
FUNCTION GetNextEvent (eventMask: INTEGER; VAR theEvent: EventRecord) :
                        BOOLEAN;
FUNCTION EventAvail   (eventMask: INTEGER; VAR theEvent: EventRecord) :
                        BOOLEAN;
```

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 Routines

```
FUNCTION TickCount    : LONGINT;
FUNCTION GetDblTime   : LONGINT;  [No trap macro]
FUNCTION GetCaretTime : LONGINT;  [No trap macro]
```

Event Message in Event Record

Event type	Event message
Keyboard	Character code and key code in low-order word
Activate, update	Pointer to window
Disk-inserted	Drive number in low-order word, File Manager result code in high-order word
Mouse-down, mouse-up, null	Meaningless
Network	See AppleBus Manager manual
Device driver	See driver documentation
Application-defined	Whatever you wish

Assembly-Language Information

Constants

```
; Event codes

nullEvt        .EQU  Ø   ;null
mButDwnEvt     .EQU  1   ;mouse-down
mButUpEvt      .EQU  2   ;mouse-up
keyDwnEvt      .EQU  3   ;key-down
keyUpEvt       .EQU  4   ;key-up
autoKeyEvt     .EQU  5   ;auto-key
updatEvt       .EQU  6   ;update
diskInsertEvt  .EQU  7   ;disk-inserted
activateEvt    .EQU  8   ;activate
networkEvt     .EQU  1Ø  ;network
ioDrvrEvt      .EQU  11  ;device driver
app1Evt        .EQU  12  ;application-defined
app2Evt        .EQU  13  ;application-defined
app3Evt        .EQU  14  ;application-defined
app4Evt        .EQU  15  ;application-defined

; Modifier flags in event record

activeFlag     .EQU  Ø   ;set if window being activated
btnState       .EQU  2   ;set if mouse button up
cmdKey         .EQU  3   ;set if Command key down
shiftKey       .EQU  4   ;set if Shift key down
alphaLock      .EQU  5   ;set if Caps Lock key down
optionKey      .EQU  6   ;set if Option key down
```

```
; Journaling mechanism Control call

jRefNum         .EQU  -2  ;reference number of standard journaling
                          ;device driver
jPlayCtl        .EQU  16  ;journal in playback mode
jRecordCtl      .EQU  17  ;journal in recording mode
jcTickCount     .EQU  Ø   ;journal code for TickCount
jcGetMouse      .EQU  1   ;journal code for GetMouse
jcButton        .EQU  2   ;journal code for Button
jcGetKeys       .EQU  3   ;journal code for GetKeys
jcEvent         .EQU  4   ;journal code for GetNextEvent and EventAvail
```

Event Record Data Structure

evtNum	Event code
evtMessage	Event message
evtTicks	Ticks since startup
evtMouse	Mouse location
evtMeta	State of modifier keys
evtMBut	State of mouse button
evtBlkSize	Length of above structure

Variables

Name	Size	Contents
KeyThresh	2 bytes	Auto-key threshold
KeyRepThresh	2 bytes	Auto-key rate
WindowList	4 bytes	Pointer to first window in window list; Ø if using events but not windows
SEvtEnb	1 byte	Ø if GetNextEvent shouldn't call SystemEvent
ScrDmpEnb	1 byte	Ø if GetNextEvent shouldn't process Command-Shift-number combinations
Ticks	4 bytes	Current number of ticks since system startup
DoubleTime	4 bytes	Double-click interval in ticks
CaretTime	4 bytes	Caret-blink interval in ticks
JournalRef	4 bytes	Reference number of journaling device driver
JournalFlag	2 bytes	Journaling mode

GLOSSARY

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 repeatedly when the user presses and holds down a character key on the keyboard or keypad.

auto-key rate: The rate at which a character key repeats after it's begun to do so.

auto-key threshold: The length of time a character key must be held down before it begins to repeat.

character code: An integer representing the character that a key or combination of keys on the keyboard or keypad stands for.

character key: Any key except Shift, Caps Lock, Command, or Option.

device driver event: An event generated by one of the Macintosh's device drivers.

disk-inserted event: An event generated when the user inserts a disk in a disk drive or takes any other action that requires a volume to be mounted.

event: A notification to an application of some occurrence that the application may want to respond to.

event code: An integer representing a particular type of event.

event mask: A parameter passed to a Toolbox or Operating System Event Manager routine to specify which types of events the routine should apply to.

event message: A field of an event record containing information specific to the particular type of event.

event queue: The Operating System Event Manager's list of pending events.

event record: The internal representation of an event, through which your program learns all pertinent information about that event.

journal code: A code passed by a Toolbox Event Manager routine in its Control call to the journaling device driver, to designate which routine is making the Control call.

journaling mechanism: A mechanism that allows you to feed the Toolbox Event Manager events from some source other than the user.

key code: An integer representing a key on the keyboard or keypad, without reference to the character that the key stands for.

key-down event: An event generated when the user presses a character key on the keyboard or keypad.

key-up event: An event generated when the user releases a character key on the keyboard or keypad.

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 character key on the keyboard or keypad; any key-down, key-up, or auto-key event.

modifier key: A key (Shift, Caps Lock, Option, or Command) that generates no keyboard events of its own, but changes the meaning of other keys or mouse actions.

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 AppleBus Manager.

null event: An event reported when there are 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.

tick: A sixtieth of a second.

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.

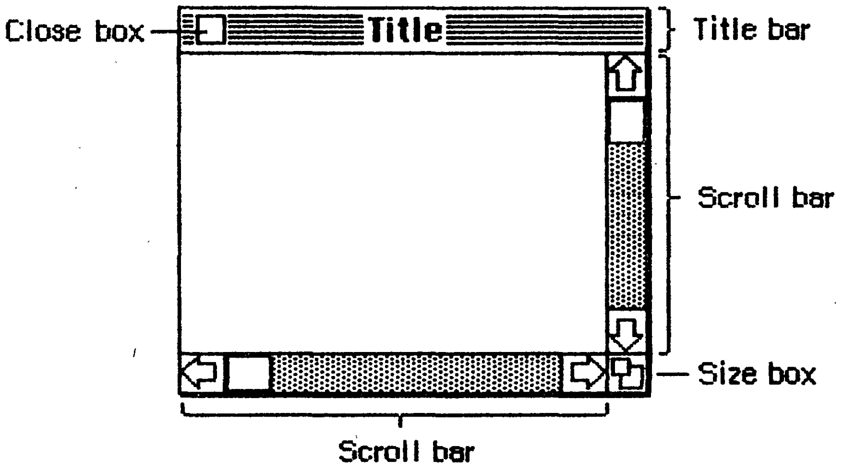


Figure 1. An Active Document Window

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).

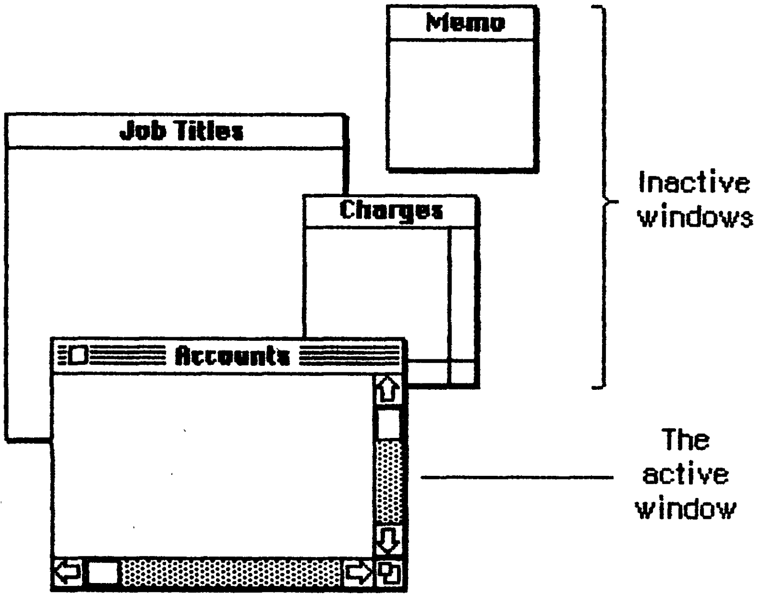


Figure 2. Overlapping Document Windows

(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.

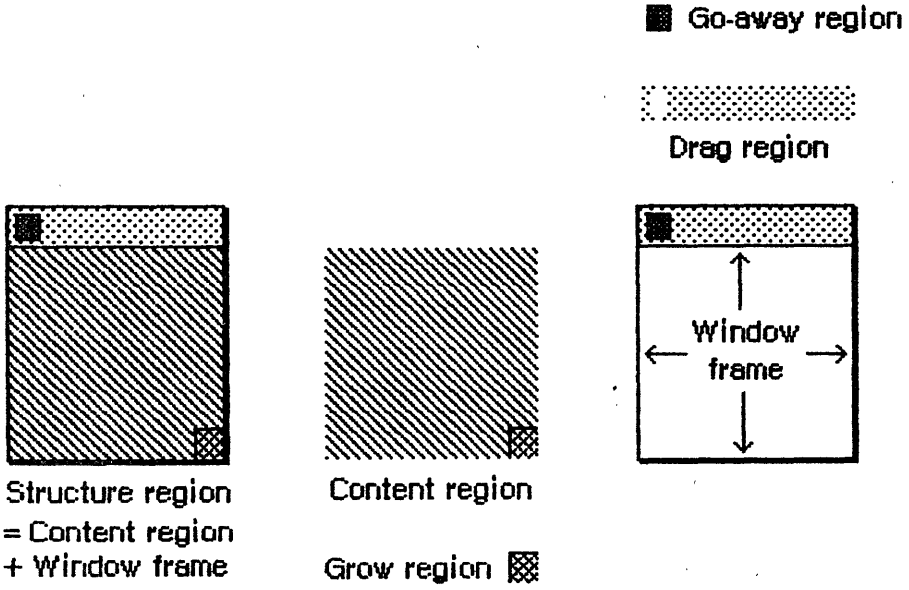


Figure 3. Document Window Regions and Frame

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  = 0;  {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}
```

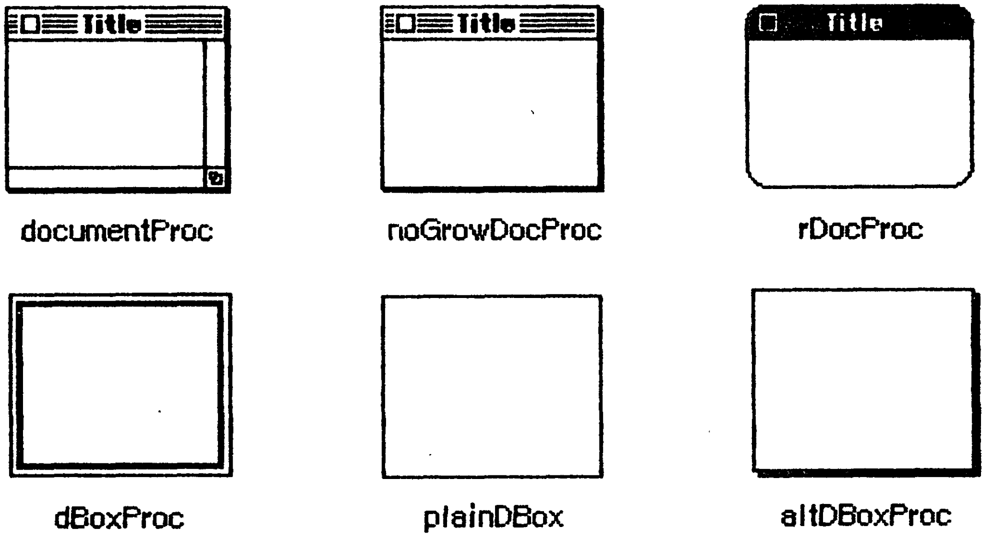


Figure 4. Predefined Types of Windows

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	10, 10
rDocProc + 5	12, 12
rDocProc + 6	20, 20
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.

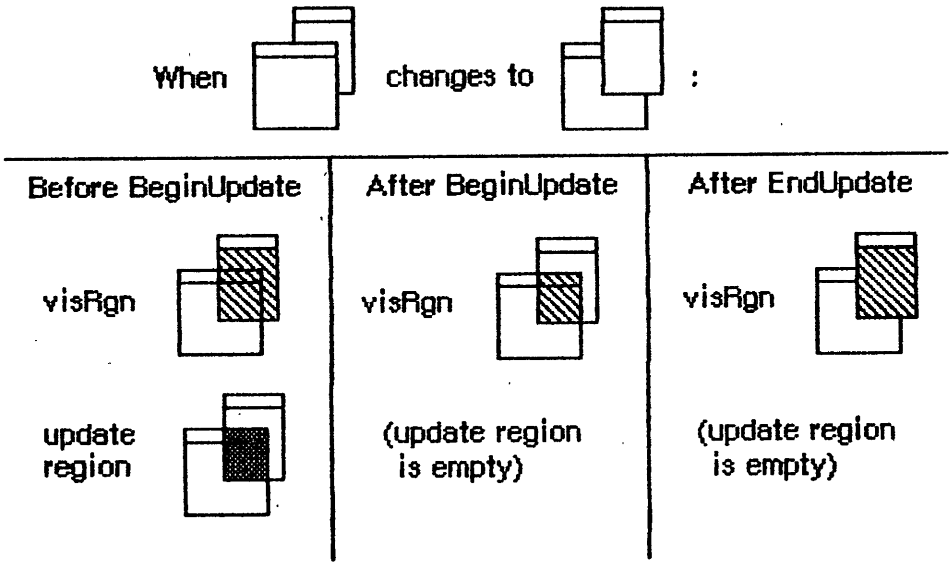


Figure 5. Updating Window Contents

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).

<u>Assembly-language note</u>: 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);

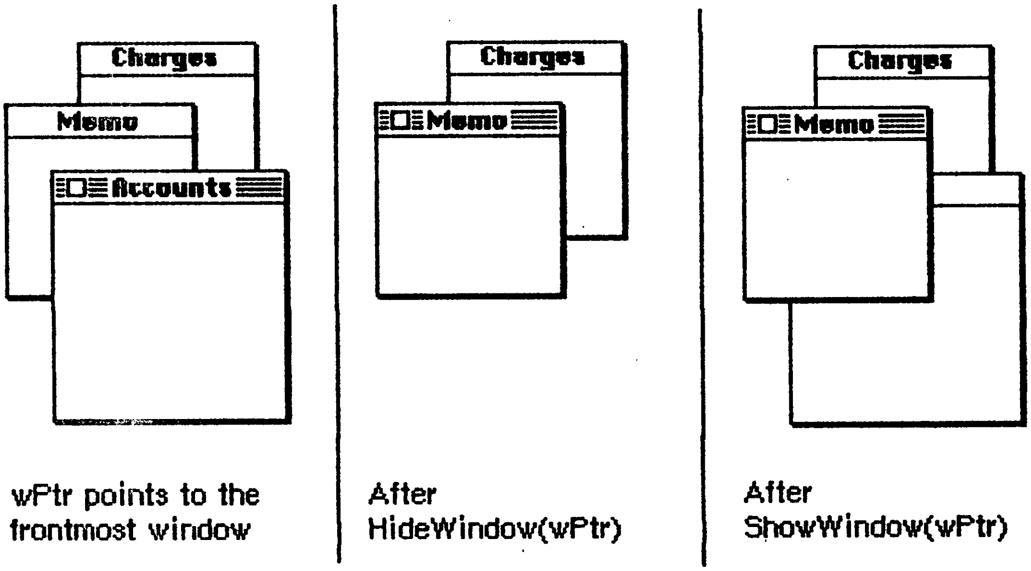
GetWTitle returns theWindow's title as the value of the title parameter.

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.



wPtr points to the frontmost window

After HideWindow(wPtr)

After ShowWindow(wPtr)

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).

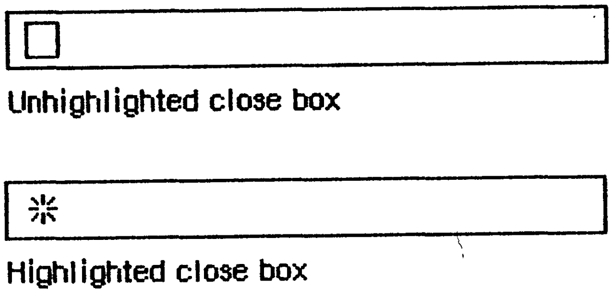


Figure 7. A Document Window's Close Box

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.

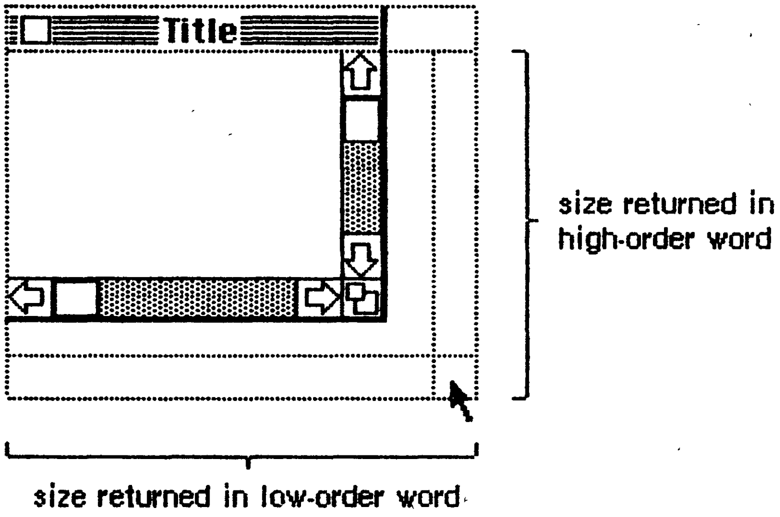

Title
size returned in high-order word
size returned in low-order word

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.

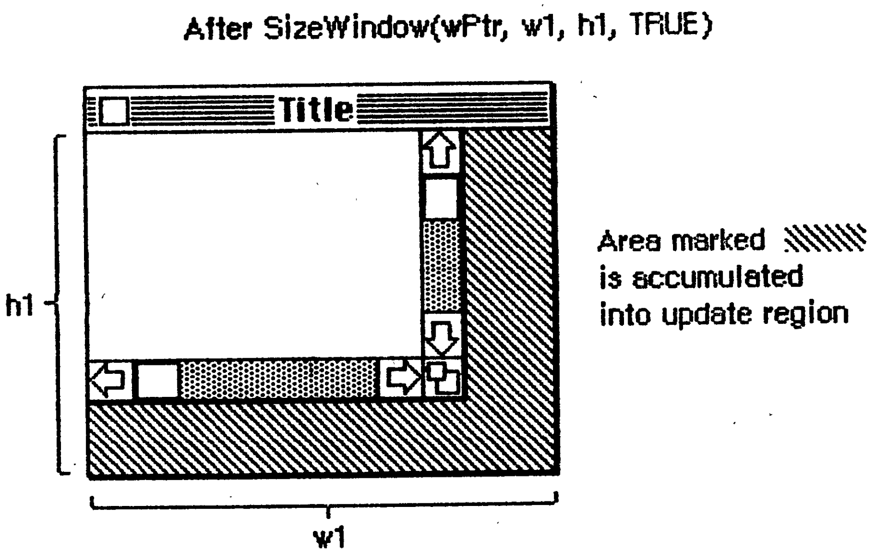


Figure 9. SizeWindow Operation on a Document Window

(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.

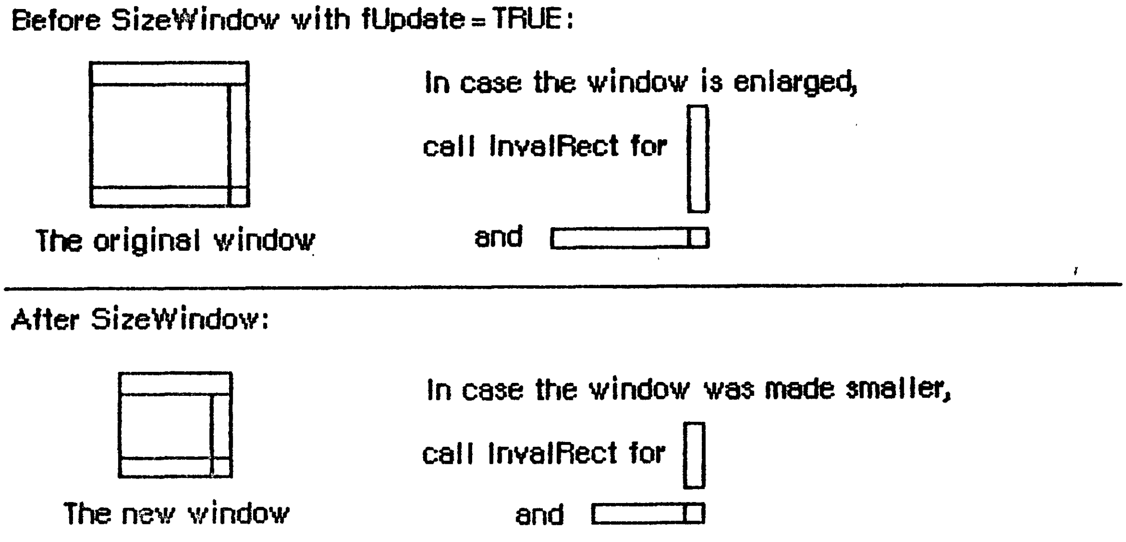

Before SizeWindow with fUpdate=TRUE:
In case the window is enlarged,
call InvalRect for
The original window
and
After SizeWindow:
In case the window was made smaller,
call InvalRect for
The new window
and

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.

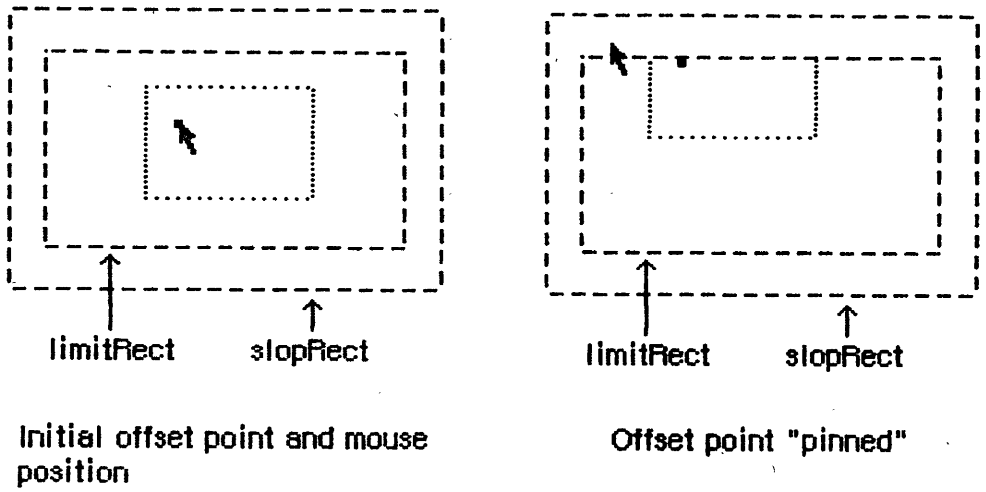
LimitRect and slopRect are also in the local coordinates of the current grafPort. To explain these parameters, the concept of "offset point" must be introduced: this is the point whose vertical and horizontal offsets from the top left corner of the region's enclosing rectangle are the same as those of 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.



Initial offset point and mouse position

Offset point "pinned"

Figure 11. DragGrayRgn Operation on a Rectangular Region

If the mouse button is released outside slopRect, DragGrayRgn returns -32768 ($8ØØØ); 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)

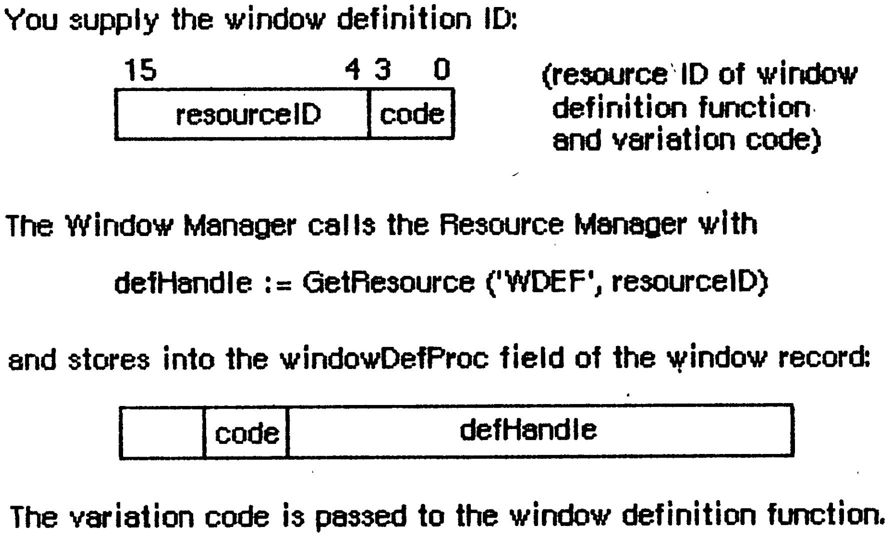
> For the convenience of your application's user, remember to conform to the Macintosh User Interface Guidelines for windows as much as possible.

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.



You supply the window definition ID:

15
4 3
0
resourceID
code
(resource ID of window definition function and variation code)

The Window Manager calls the Resource Manager with

defHandle := GetResource ('WDEF', resourceID)

and stores into the windowDefProc field of the window record:

The variation code is passed to the window definition function.

Figure 12. Wiñdow 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.

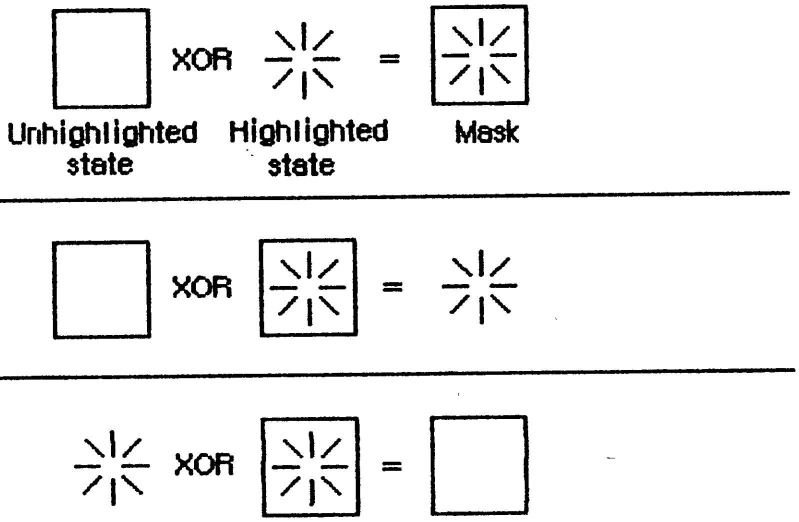


Figure 13. Toggling the Go-Away Region

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.

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.

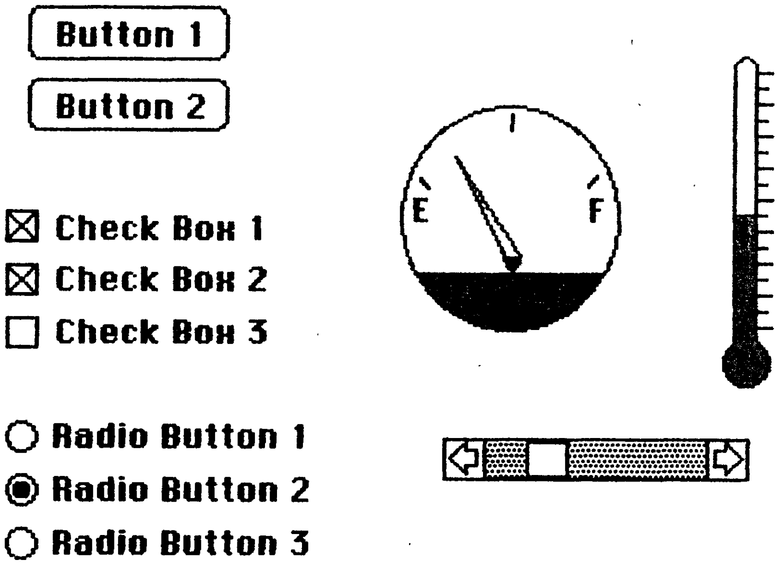


Figure 1. Controls

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").

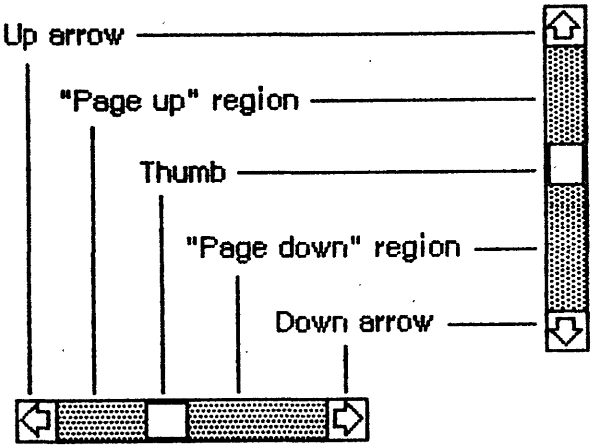


Figure 2. Parts of a Scroll Bar

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.

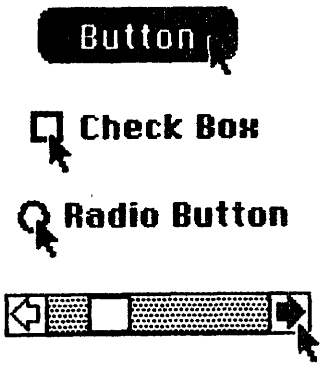


Figure 3. Highlighted Controls

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).

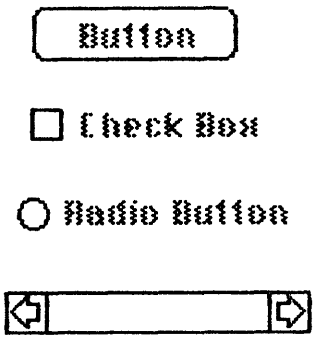


Figure 4. Inactive Controls

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 (0,0)--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   = 0;   {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 <u>control record</u> 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 <u>visible</u> or <u>invisible</u>. 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 <u>action procedure</u>. 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

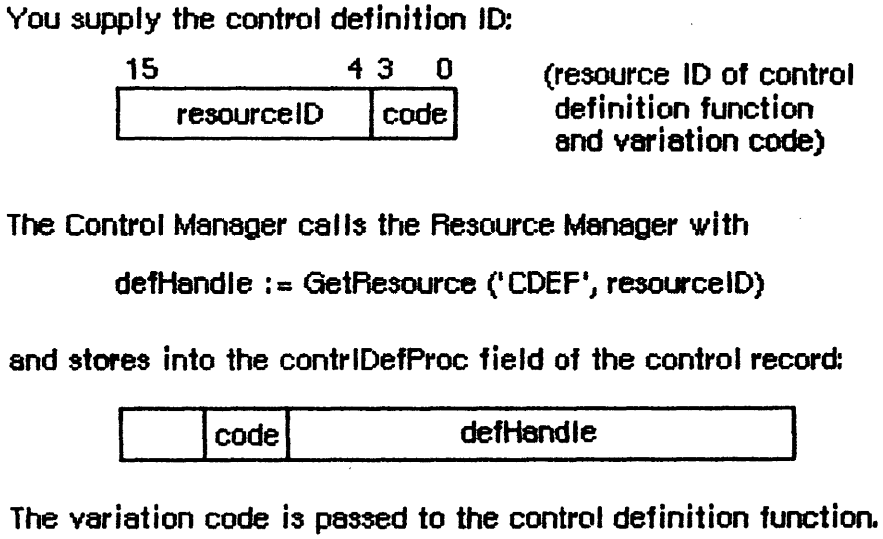
In addition to the standard, built-in control types (buttons, check boxes, radio buttons, and scroll bars), the Control Manager allows you to define "custom" control types of your own. Maybe you need a three-way selector switch, a memory-space indicator that looks like a thermometer, or a thruster control 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.



You supply the control definition ID:

15
4 3
0
resourceID
code
(resource ID of control definition function and variation code)

The Control Manager calls the Resource Manager with

defHandle := GetResource ('CDEF', resourceID)

and stores into the contrlDefProc field of the control record:

The variation code is passed to the control definition function.

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  = 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}
```

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 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 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 Ø 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      10       ;simple button
inCheckBox        .EQU      11       ;check box or radio button
inUpButton        .EQU      20       ;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       0       ;no constraint
hAxisOnly         .EQU       1       ;horizontal axis only
vAxisOnly         .EQU       2       ;vertical axis only

; Messages to control definition function

drawCtlMsg    .EQU   0  ;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).

MACINTOSH USER EDUCATION

The Menu Manager: A Programmer's Guide /MMGR/MENUS

See Also:

Macintosh User Interface Guidelines
Inside Macintosh: A Road Map
Macintosh Memory Management: An Introduction
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
Programming Macintosh Applications in Assembly Language

Modification History:	First Draft	Pam Stanton-Wyman	5/82
	Second Draft	Chris Espinosa	12/23/82
	Updated (ROM 2.Ø)	Chris Espinosa	1/24/83
	Third Draft (ROM 3.Ø)	Chris Espinosa & Caroline Rose	5/17/83
	Fourth Draft (ROM 7)	Caroline Rose	11/1/83
	Fifth Draft	Katie Withey	9/24/84

ABSTRACT

Menus free the user from having to remember long strings of command words. The menu bar and pull-down menus let the user see all available menu choices at any time. 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 draft:

- For menus stored in resource files, the menu ID isn't necessarily the resource ID (though the Resource Compiler sets the menu ID to the resource ID) (page 8).

- Two new constants have been defined for symbols to mark menu items (the Command key symbol and a diamond symbol), and the constant appleSymbol has been changed to appleMark (page 12).

- If no item is chosen from a menu, only the high-order word of the long integer returned by MenuSelect or MenuKey is Ø; the low-order word is undefined.

- Important changes have been made to "Defining Your Own Menus" (page 27).

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 the comprehensive Inside Macintosh manual. *** 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.

Like all Toolbox documentation, this manual assumes you're familiar with Lisa Pascal and the information in the following manuals:

- Inside Macintosh: A Road Map
- Macintosh User Interface Guidelines
- Macintosh Memory Management: An Introduction
- Programming Macintosh Applications in Assembly Language, if you're using assembly language

You should also be familiar with:

- Resources, as described in the Resource Manager manual.
- The basic concepts and structures behind QuickDraw, particularly points, rectangles, and character style.
- The Toolbox Event Manager. Some Menu Manager routines should be called only in response to certain events.

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 through the menu causes each of the menu 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.

When the user chooses an item, the Menu Manager tells the application which item was chosen, and the application performs the corresponding action. When the application completes the action, it removes the highlighting from the menu title, indicating to the user that the operation is complete.

If the user moves the cursor out of the menu with the mouse button held down, the menu remains visible, though no menu items are highlighted. If the mouse button is released outside the menu, no choice is made: the menu just disappears and the application takes no action. The user can always look at a menu without causing any changes in the document or on the screen.

The Menu Bar

The menu bar always appears at the top of the Macintosh screen; nothing but the cursor ever appears in front of it. The menu bar is white, 2Ø pixels high, and as wide as the screen, with a thin black lower border. The menu titles in it are always in the system font and the system font size (see Figure 1).

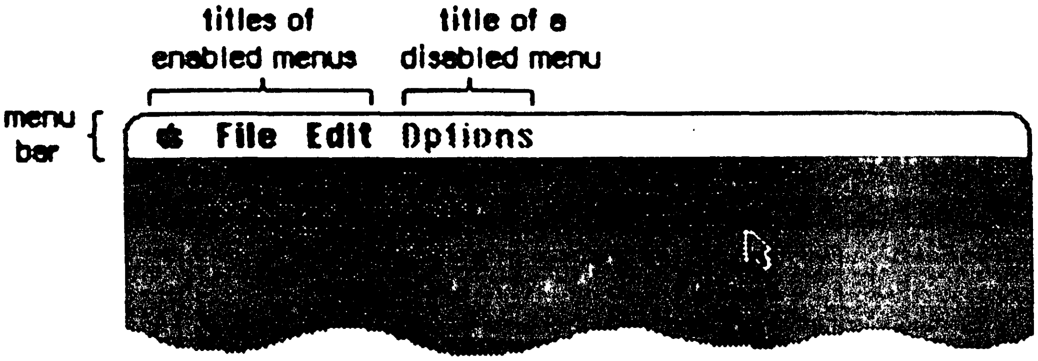


Figure 1. The Menu Bar

In applications that support desk accessories, the first menu should be the standard Apple menu (the menu whose title is an apple symbol). The Apple menu contains the names of all available desk accessories. When the user chooses a desk accessory from the menu, the title of a menu belonging to the desk accessory may appear in the menu bar, for as long as the accessory is active, or the entire menu bar may be replaced by menus belonging to the desk accessory. (Desk accessories are discussed in detail in the Desk Manager manual.)

A menu may be temporarily disabled, so that none of the items in it can be chosen. A disabled menu can still be pulled down, but its title and all the items in it are dimmed.

The maximum number of menu titles in the menu bar is 16; however, ten to twelve titles are 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 menu items listed vertically inside a shadowed rectangle. A menu item may be the text of a command, or just a line dividing groups of choices (see Figure 2). An ellipsis (...) following the text of an item indicates that selecting the item will bring up a dialog box to get further information before the command is executed. 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.

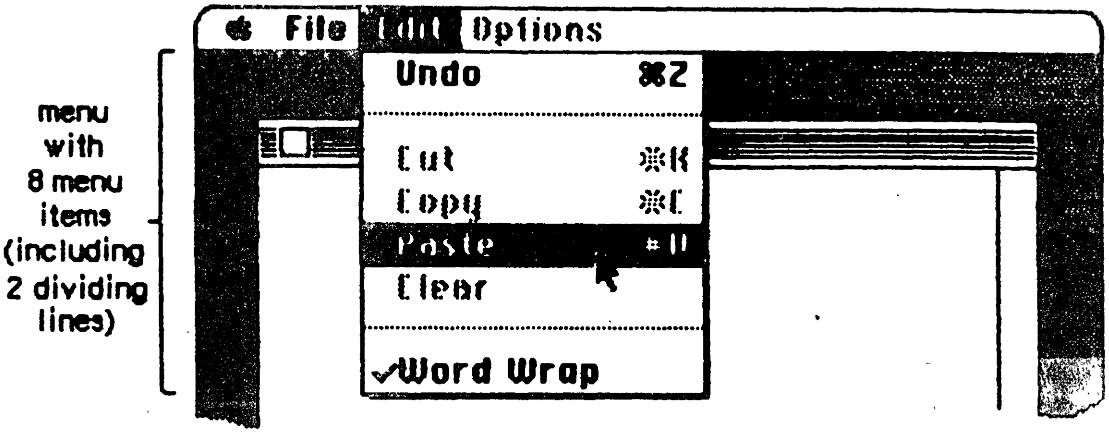


Figure 2. A Standard Menu

The text of a menu item 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). Pressing this key while holding down the Command key invokes the item just as if it had been chosen from the menu (see "Keyboard Equivalents for Commands" below).

- 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, and can't be chosen. The Cut, Copy, and Clear commands in Figure 2 are disabled; dividing lines are always disabled.

(note)

> 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 (less 1 for each item that contains an icon). The fewer menu items you have, the simpler and clearer the menu appears to the user.

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.

Keyboard Equivalents for Commands

Your program can set up a keyboard equivalent for any of its menu commands so the command can be invoked from the keyboard. The character you specify for a keyboard equivalent will usually be a letter. The user can type the letter in either uppercase or lowercase. For example, typing either "C" or "c" while holding down the Command key invokes the command whose equivalent is "C".

(note)

> For consistency between applications, you should specify the letter in uppercase in the menu.

You can specify characters other than letters for keyboard equivalents. However, the Shift key will be ignored when the equivalent is typed, so you shouldn't specify shifted characters. For example, when the user types Command-Shift-+, the system reads it as Command-Shift-=.

Command-Shift-number combinations are not keyboard equivalents. They're detected and handled by the Toolbox Event Manager function GetNextEvent, and are never returned to your program. (This is how disk ejection with Command-Shift-1 or 2 is implemented.) Although it's possible to use unshifted, Command-number combinations as keyboard equivalents, you shouldn't do so, to avoid confusion. *** (Command-Shift-number will be documented in the next draft of the Toolbox Event Manager manual.) ***

(warning)

> You must use the standard keyboard equivalents Z, X, C, and V for the editing commands Undo, Cut, Copy, and Paste, or editing won't work correctly in desk accessories.

MENUS AND RESOURCES

The general appearance and behavior of a menu is determined by a routine called its menu definition procedure, which is usually stored as a resource in a resource file. The menu definition procedure performs all actions that differ from one menu type to another, such as drawing the menu. The Menu Manager calls the menu definition procedure whenever it needs to perform one of these basic actions, passing it a message that tells which action to perform.

The standard menu definition procedure is part of the system resource file. It lists the menu items vertically, and each item may have an icon, a check mark or other symbol, a keyboard equivalent, a different character style, or a dimmed appearance. If you want to define your own, nonstandard menu types, you'll have to write your own menu definition procedure for them, as described later in the section "Defining Your Own Menus".

You can also use a resource file to store the contents of your application's menus. This allows the menus to be edited or translated to foreign languages without affecting the application's source code. The Menu Manager lets you read complete menu bars as well as individual menus from a resource file.

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; the Menu Manager calls the Resource Manager to read in the icons.

(note)

> You can create menu-related resources 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 formats of these resources in the file, but for interested programmers this information is given in the section "Formats of Resources for Menus".

There's a Menu Manager procedure that scans all open resource files for resources of a given type and installs 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 or fonts, 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 following:

- The menu ID, a number that identifies the menu. The menu ID can be the same number as the menu's resource ID, though it doesn't have to be.
- 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 record 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 the exact field names. However, if you want more information about the exact structure of a menu record--if you're defining your own menu types, for instance--it's given below.

The MenuInfo Data Type

The type MenuInfo is defined as follows:

```
TYPE MenuInfo = RECORD
                  menuID:       INTEGER;  {menu ID}
                  menuWidth:    INTEGER;  {menu width in pixels}
                  menuHeight:   INTEGER;  {menu height in pixels}
                  menuProc:     Handle;   {menu definition procedure}
                  enableFlags:  LONGINT;
                                  {tells if menu or items are enabled}
                  menuData:     Str255    {menu title (and other data)}
                END;
```

The menuID field contains the menu ID. MenuWidth and menuHeight are the menu's horizontal and vertical dimensions in pixels. MenuProc is a handle to the menu definition procedure for this type of menu.

Bit Ø of the enableFlags field is 1 if the menu is enabled, or Ø if it's disabled. Bits 1 to 31 similarly tell 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.

(warning)

> 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 global constant menuBlkSize equals the length in bytes of all the fields of a menu record except menuData.

MENU LISTS

A menu list contains handles to one or more menus, along with information about the position of each menu in the menu bar. The current menu list contains handles to all the menus currently in the menu bar; the menu bar shows the titles, in order, of all menus in the menu list. When you initialize the Menu Manager, it allocates space for the maximum size menu list.

The Menu Manager provides all the necessary routines for manipulating the current menu list, so there's no need to access it directly yourself. 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 global variable MenuList contains a handle to the current menu list. The menu list has the format shown below.

Number of bytes	Contents
2 bytes	Offset from beginning of menu list to last menu handle (the number of menus in the list times 6)
2 bytes	Horizontal coordinate of right edge of menu title of last menu in list
2 bytes	Not used
For each menu:	
4 bytes	Menu handle
2 bytes	Horizontal coordinate of left edge of menu

CREATING A MENU IN YOUR PROGRAM

The best way to create your application's menus is to set them up as resources and read them in from a resource file. If you want your application to create the menus itself, though, it must call the NewMenu and AppendMenu routines. 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 dividing line, use one hyphen (-); AppendMenu ignores any following characters, and draws a continuous line across the width of the menu. For a blank item, use one or more spaces. Other characters interspersed in the string 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 aren't displayed in the menu.

Meta-character	Meaning
; or Return	Separates items
^	Item has an icon
!	Item has a check 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're 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.

(note)

> 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.

Multiple 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 text in the menu. The digit, called the icon number, yields the resource ID of the icon in the resource file. Icon 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.

(note)

> 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.

You can also use the SetItemIcon procedure to install icons in a menu; it accepts any icon number from 1 to 255.

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.

The SetItemStyle procedure allows you to assign any combination of stylistic variations to an item. For a further discussion of character style, see the QuickDraw manual.

Items with Keyboard Equivalents

A slash (/) followed by a character associates that character with the item, allowing the item to be invoked from the keyboard with the Command key. The specified character (preceded by the Command key symbol) appears at the right of the item's text in the menu.

(note)

> Remember to specify the character in uppercase if it's a letter, and not to specify other shifted characters or numbers.

Given a keyboard equivalent typed by the user, you call the MenuKey function to find out what menu item was invoked.

Disabled Items

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.

Menu items that are used to separate groups of items (such as a line or a blank item) should always be disabled. For example, the call

```
AppendMenu(thisMenu,'Undo;(-;Cut')
```

adds two enabled menu items, Undo and Cut, with a disabled item consisting of a line between them.

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:

- Read an entire prepared menu list from a resource file with GetNewMBar, and place it in the menu bar with SetMenuBar.

- Read the menus individually from a resource file using GetMenu, and place them in the menu bar using InsertMenu.

- Allocate the menus with NewMenu, fill them with items using AppendMenu, and place them in the menu bar using InsertMenu.

- 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.

When you no longer need a menu, call the Resource Manager procedure ReleaseResource if you read the menu from a resource file, or DisposeMenu if you allocated it with NewMenu.

If you call NewMenu to allocate a menu, it will store a handle to the standard menu definition procedure in the menu record, so if you want the menu to be one you've designed, you must replace that handle with a handle to your own menu definition procedure. For more information, see "Defining Your Own Menus".

After setting up the menu bar, you need to draw it with the DrawMenuBar procedure.

At any time you can change or examine a menu item's text with the SetItem and GetItem procedures, or its icon, style, or mark with the procedures SetItemIcon, GetItemIcon, SetItemStyle, GetItemStyle, CheckItem, SetItemMark, and GetItemMark. Individual items or whole menus can be enabled or disabled with the EnableItem and DisableItem procedures. You can change the number of menus in the menu list with InsertMenu or DeleteMenu, remove all the menus with ClearMenuBar, or change the entire menu list with GetNewMBar or GetMenuBar followed by 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 <u>menu item number</u> of the item that was chosen. The menu item number is the index, starting from 1, of the item in the menu. If no item was chosen, the high-order word of the long integer is Ø, and the low-order word is undefined.

- If the high-order word of the long integer returned is Ø, the application should just continue to poll for further events.

- If the high-order word is nonzero, the application should invoke the menu item specified by the low-order word, in the menu specified by the high-order word. Only after the action is completely finished (after all dialogs, alerts, or screen actions have been taken care of) should the application remove the highlighting from the menu bar by calling HiliteMenu(Ø), 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. Applications should respond the same way to auto-key events as to key-down events when the Command key is held down.if the command being invoked is repeatable.

(note)

> 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.

There are several miscellaneous Menu Manager routines that you normally won't need to use. CalcMenuSize calculates the dimensions of a menu. CountMItems counts the number of items in a menu. GetMHandle returns the handle of a menu in the menu list. FlashMenuBar inverts the menu bar. SetMenuFlash controls the number of times a menu item blinks when it's chosen.

MENU MANAGER ROUTINES

Initialization and Allocation

PROCEDURE InitMenus;

InitMenus initializes the Menu Manager. It allocates space for the menu list (a relocatable block on the heap large enough for the maximum size menu list), and draws the (empty) menu bar. Call InitMenus 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.

(note)

> The Window Manager initialization procedure InitWindows has already drawn the empty menu bar; InitMenus redraws it.

```
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. It sets up the menu to use the standard menu definition procedure. 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.

If you want to set up the title of the Apple menu from your program instead of reading it in from a resource file, you can use the predefined constant appleMark (equal to $14, the value of the apple symbol). For example, you can declare the string variable

```
VAR myTitle: STRING[1];
```

and do the following:

```
myTitle := ' ';
myTitle[1] := CHR(appleMark)
```

To release the memory occupied by a menu that you created with NewMenu, call DisposeMenu.

```
FUNCTION GetMenu (resourceID: INTEGER) : MenuHandle;
```

GetMenu returns a menu handle for the menu having the given resource ID. It calls the Resource Manager to read the menu 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.

(warning)

> Only call GetMenu once for a particular menu. If you need the menu handle to a menu that's already in memory, use the Resource Manager function GetResource.

To release 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 (theMenu: MenuHandle);

Call DisposeMenu to release 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.

(warning)

> 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.

Forming the Menus

PROCEDURE AppendMenu (theMenu: 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. If it begins with a hyphen (-), the item will be a dividing line across the width of the menu. As described in the section "Creating a Menu in Your Program", 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 (theMenu: 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".

(note)

> So that you can have resources of the given type that won't appear in the menu, any resource names that begin with a period (.) or a percent sign (%) aren't appended by AddResMenu.

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 (theMenu: 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.

(note)

> InsertResMenu inserts the names in the reverse of the order that AddResMenu appends them. For consistency between applications in the appearance of menus, use AddResMenu instead of InsertResMenu if possible.

Forming the Menu Bar

PROCEDURE InsertMenu (theMenu: 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 or the menu list is already full, 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.

(note)

> DeleteMenu simply removes the menu from the list of currently available menus; it doesn't release the memory occupied by 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.

(note)

> ClearMenuBar, like DeleteMenu, doesn't release the memory occupied by 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 created by GetNewMBar the current menu list, call SetMenuBar. To release the memory occupied by the menu list, use the Memory Manager procedure DisposHandle.

(warning)

> 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 Routines").

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 release the memory occupied by the saved menu list, use the Memory Manager procedure DisposHandle.

(warning)

> GetMenuBar doesn't copy the menus themselves, only a list containing their handles. Do not dispose of any menus that might be in a saved menu list.

PROCEDURE SetMenuBar (menuList: Handle);

SetMenuBar copies the given menu list to 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 there's a mouse-down event in the menu bar, the application should call MenuSelect with startPt equal to the point (in global coordinates) where the mouse button was pressed. MenuSelect keeps control until the mouse button is released, tracking the mouse, pulling down menus as needed, and highlighting enabled 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.

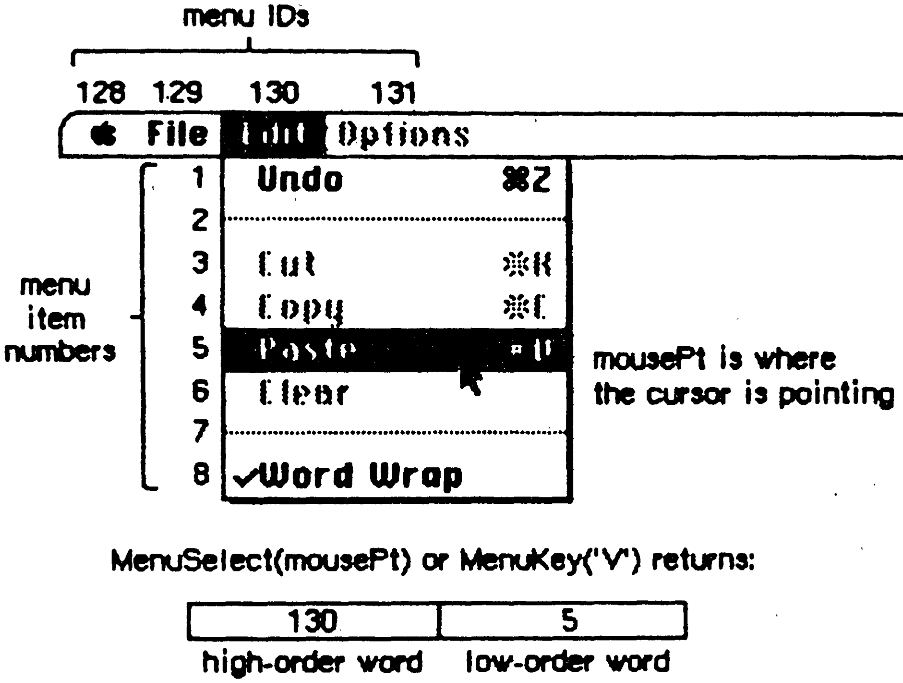


Figure 3. MenuSelect and MenuKey

If no choice is made, MenuSelect returns Ø in the high-order word of the long integer, and the low-order word is undefined. This includes the case where the mouse button is released over a disabled menu item (such as Cut, Copy, Clear, or one of the dividing lines in Figure 3), over any menu title, or outside the menu.

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 in the high-order word of the result.

Assembly-language note: If the global variable 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 global variable MenuHook normally contains Ø; if you store the address of a routine in MenuHook, MenuSelect will call that routine repeatedly (with no parameters) 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--or an auto-key event, if the command being invoked is repeatable--call MenuKey with the character that was typed. MenuKey highlights the appropriate menu title, and returns a long integer containing the menu ID in its high-order word and the menu item number in its low-order word, just as MenuSelect does (see Figure 3 above). After performing the chosen task, your application should call HiliteMenu(Ø) to remove the highlighting from the menu title.

If the given character isn't associated with any enabled menu item currently in the menu list, MenuKey returns Ø in the high-order word of the long integer, and the low-order word is undefined.

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 in the high-order word of the result.

(note)

> 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 it encounters, scanning the menus from right to left 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 (if any).

After MenuSelect or MenuKey, your application should perform the chosen task and then call HiliteMenu(Ø) to unhighlight the chosen menu title.

<u>Assembly-language note</u>: The global variable TheMenu contains the menu ID of the currently highlighted menu.

Controlling Items' Appearance

```
PROCEDURE SetItem (theMenu: 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.

(note)

> It's good practice to store the text of itemString in a resource file instead of passing it directly.

Use SetItem to flip between two alternative menu items-- for example, to change "Show Clipboard" to "Hide Clipboard" when the Clipboard is already showing.

(note)

> To avoid confusing the user, don't capriciously change the text of menu items.

```
PROCEDURE GetItem (theMenu: 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 (theMenu: 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 in the high-order word of their result 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 (theMenu: 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 (theMenu: 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 SetItemMark (theMenu: 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 noMark      = Ø;     {NUL character, to remove a mark}
      commandMark = $11;   {Command key symbol}
      checkMark   = $12;   {check mark}
      diamondMark = $13;   {diamond symbol}
      appleMark   = $14;   {apple symbol}
```

<u>Assembly-language</u> <u>note</u>: The macro you invoke to call SetItemMark from assembly language is named _SetItmMark.

PROCEDURE GetItemMark (theMenu: 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.

PROCEDURE SetItemIcon (theMenu: MenuHandle; item: INTEGER; icon: Byte);

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.

(warning)

> 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 (theMenu: MenuHandle; item: INTEGER; VAR icon:
          Byte);

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 (theMenu: 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 (theMenu: 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.

Miscellaneous Routines

```
PROCEDURE CalcMenuSize (theMenu: 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 internally by the Menu Manager after every AppendMenu, SetItem, SetItemIcon, and SetItemStyle call.

```
FUNCTION CountMItems (theMenu: 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.

PROCEDURE SetMenuFlash (count: INTEGER);

When the mouse button is released over an enabled menu item, the item blinks briefly to confirm the choice. Normally, your application shouldn't be concerned with this blinking; the user sets it with the Control Panel desk accessory. If you're writing a desk accessory like the Control Panel, though, SetMenuFlash allows you to control the duration of this blinking. Count is the number of times menu items will blink; it's initially 3 if the user hasn't changed it. A count of Ø disables blinking. Values greater than 3 can be annoyingly slow.

(warning)

> Don't call SetMenuFlash from your main program.

> Assembly-language note: The macro you invoke to call SetMenuFlash from assembly language is named _SetMFlash. The current count is stored in the global variable MenuFlash.

(note)

> 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 below.

DEFINING YOUR OWN MENUS

The standard type of Macintosh menu is predefined for you. However, you may want to define your own type of menu--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 write a menu definition procedure and (usually) store it in a resource file. The Menu Manager calls the menu definition procedure to perform basic operations such as drawing the menu.

A menu in a resource file contains the resource ID of its menu definition procedure. The routine you use to read in the menu is GetMenu (or GetNewMBar, which calls GetMenu). 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.

If you create your menus with NewMenu instead of storing them as resources, NewMenu 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, then call CalcMenuSize. If your menu 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.

(note)

> Advanced programmers can include the menu definition procedure in with the program code instead of storing it as a separate resource.

The Menu Definition Procedure

The menu definition procedure 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 the procedure. Here's how you would declare one named MyMenu:

```
PROCEDURE MyMenu (message: INTEGER; theMenu: MenuHandle; VAR
        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 item was chosen and }
                       { highlight it}
      mSizeMsg   = 2;  {calculate the menu's dimensions}
```

The parameter theMenu indicates the menu that the operation will affect. MenuRect is the rectangle (in global coordinates) in which the menu is located; it's used when the message is mDrawMsg or mChooseMsg.

(note)

> MenuRect is declared as a VAR parameter not because its value is changed, but because of a Pascal feature that will cause an error when that parameter isn't used.

The message mDrawMsg tells the menu definition procedure to draw the menu inside menuRect. The current grafPort will be the Window Manager port. (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). You should also check the enableFlags field of the menu record to see if the menu is disabled (or if any of the menu items are disabled, if you're using all the flags), and if so, draw it in gray.

(warning)

> Don't change the font from the system font for menu text. (The Window Manager port uses 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 released, and the whichItem parameter is the item number of the last item that was chosen from this menu. The procedure should determine if the mouse button was released in an enabled menu item, by checking whether hitPt is inside menuRect, whether the menu is enabled, and whether hitPt is in an enabled menu item:

- If the mouse button was released in an enabled menu item, unhighlight whichItem and highlight the newly chosen item (unless the new item is the same as the whichItem), and return the item number of the new item in whichItem.

- If the mouse button wasn't released in an enabled item, unhighlight whichItem and return Ø.

(note)

> 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 resource data is simply the compiled or 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 '; if you use the SetItem procedure to change a menu item's text, you should store the alternate text as a string resource.

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 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
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, bits 1 to 31 of the enableFlags field may be set and used in any way desired by the menu definition procedure; bit Ø applies to the entire menu and must reflect whether it's enabled or disabled.

If your menu definition procedure does use the enableFlags field, 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.

(note)

See "Using QuickDraw from Assembly Language" in the QuickDraw manual for the exact format of the character style byte.

(warning)

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 { Special characters }

      noMark      = Ø;    {NUL character, to remove a mark}
      commandMark = $11;  {Command key symbol}
      checkMark   = $12;  {check mark}
      diamondMark = $13;  {diamond symbol}
      appleMark   = $14;  {apple symbol}

      { Messages to menu definition procedure }

      mDrawMsg   = Ø;   {draw the menu}
      mChooseMsg = 1;   {tell which item was chosen and highlight it}
      mSizeMsg   = 2;   {calculate the menu's dimensions}

      { Resource ID of standard menu definition procedure }

      textMenuProc = Ø;
```

Data Types

```
TYPE MenuHandle = ^MenuPtr;
     MenuPtr    = ^MenuInfo;
     MenuInfo   = RECORD
                    menuID:       INTEGER;  {menu ID}
                    menuWidth:    INTEGER;  {menu width in pixels}
                    menuHeight:   INTEGER;  {menu height in pixels}
                    menuProc:     Handle;   {menu definition procedure}
                    enableFlags:  LONGINT;
                                      {tells if menu or items are enabled}
                    menuData:     Str255    {menu title (and other data)}
                  END;
```

Routines

Initialization and Allocation

```
PROCEDURE InitMenus;
FUNCTION  NewMenu       (menuID: INTEGER; menuTitle: Str255) :
                         MenuHandle;
FUNCTION  GetMenu       (resourceID: INTEGER) : MenuHandle;
PROCEDURE DisposeMenu   (theMenu: MenuHandle);
```

Forming the Menus

```
PROCEDURE AppendMenu    (theMenu: MenuHandle; data: Str255);
PROCEDURE AddResMenu    (theMenu: MenuHandle; theType: ResType);
PROCEDURE InsertResMenu (theMenu: MenuHandle; theType: ResType;
                         afterItem: INTEGER);
```

Forming the Menu Bar

```
PROCEDURE InsertMenu    (theMenu: MenuHandle; beforeID: INTEGER);
PROCEDURE DrawMenuBar;
PROCEDURE DeleteMenu    (menuID: INTEGER);
PROCEDURE ClearMenuBar;
FUNCTION  GetNewMBar    (menuBarID: INTEGER) : Handle;
FUNCTION  GetMenuBar :   Handle;
PROCEDURE SetMenuBar    (menuList: Handle);
```

Choosing from a Menu

```
FUNCTION  MenuSelect (startPt: Point) : LONGINT;
FUNCTION  MenuKey    (ch: CHAR) : LONGINT;
PROCEDURE HiliteMenu (menuID: INTEGER);
```

Controlling Items' Appearance

```
PROCEDURE SetItem      (theMenu: MenuHandle; item: INTEGER; itemString:
                        Str255);
PROCEDURE GetItem      (theMenu: MenuHandle; item: INTEGER; VAR
                        itemString: Str255);
PROCEDURE DisableItem  (theMenu: MenuHandle; item: INTEGER);
PROCEDURE EnableItem   (theMenu: MenuHandle; item: INTEGER);
PROCEDURE CheckItem    (theMenu: MenuHandle; item: INTEGER; checked:
                        BOOLEAN);
PROCEDURE SetItemMark  (theMenu: MenuHandle; item: INTEGER; markChar:
                        CHAR);
PROCEDURE GetItemMark  (theMenu: MenuHandle; item: INTEGER; VAR markChar:
                        CHAR);
PROCEDURE SetItemIcon  (theMenu: MenuHandle; item: INTEGER; icon: Byte);
PROCEDURE GetItemIcon  (theMenu: MenuHandle; item: INTEGER; VAR icon:
                        Byte);
PROCEDURE SetItemStyle (theMenu: MenuHandle; item: INTEGER; chStyle:
                        Style);
PROCEDURE GetItemStyle (theMenu: MenuHandle; item: INTEGER; VAR chStyle:
                        Style);
```

Miscellaneous Routines

```
PROCEDURE CalcMenuSize (theMenu: MenuHandle);
FUNCTION  CountMItems  (theMenu: MenuHandle) : INTEGER;
FUNCTION  GetMHandle   (menuID: INTEGER) : MenuHandle;
PROCEDURE FlashMenuBar (menuID: INTEGER);
PROCEDURE SetMenuFlash (count: 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; VAR menuRect:
                  Rect; hitPt: Point; VAR whichItem: INTEGER);
```

Assembly-Language Information

Constants

```
; Special characters

noMark          .EQU    Ø       ;NUL character, to remove a mark
commandMark     .EQU    $11     ;Command key symbol
checkMark       .EQU    $12     ;check mark
diamondMark     .EQU    $13     ;diamond symbol
appleMark       .EQU    $14     ;apple symbol

; Messages to menu definition procedure

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
```

```
; Resource ID of standard menu definition procedure

textMenuProc    .EQU    Ø
```

Menu Record Data Structure

menuID	Menu ID
menuWidth	Menu width in pixels
menuHeight	Menu height in pixels
menuDefHandle	Handle to menu definition procedure
menuEnable	Enable flags
menuData	Menu title followed by data defining the items
menuBlkSize	Length of above structure 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

Variables

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 plain 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 255 to which the Menu Manager adds 256 to get the resource ID of an icon associated with a menu item.

keyboard equivalent: The combination of the Command key and another key, used to invoke a menu item from the keyboard.

menu: A list of menu items that appears when the user points to a menu title in the menu bar and presses the mouse button. 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: A number in the menu record that identifies the menu.

menu item: A choice in a menu, usually a command to the current application.

menu item number: The index, starting from 1, of a menu item in a menu.

menu list: A list containing menu handles for all menus in the menu bar, along with information on the position of each menu.

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.

TextEdit: A Programmer's Guide /TEXTEDIT/EDIT

See Also: The Macintosh User Interface Guidelines
Inside Macintosh: A Road Map
Macintosh Memory Management: An Introduction
QuickDraw: A Programmer's Guide
The Font Manager: A Programmer's Guide
The Window Manager: A Programmer's Guide
The Control Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Scrap Manager: A Programmer's Guide
The Toolbox Utilities: A Programmer's Guide
Programming Macintosh Applications in Assembly Language

Modification History:			
	First Draft (ROM 7)	Bradley Hacker	9/28/83
	Second Draft	Katie Withey	1/14/85

ABSTRACT

TextEdit is the part of the Macintosh User Interface Toolbox that handles basic text formatting and editing capabilities in a Macintosh application. This manual describes the TextEdit routines and data types in detail.

Summary of significant changes and additions since last draft:

- Several field names in the edit record and the descriptions of some of the fields have changed; all fields are now shown (page 9).

- Writing word break and automatic scrolling routines is now documented (pages 12-13).

- Routines for cutting and pasting text between applications have been added: TEFromScrap, TEToScrap, TEScrapHandle, TEGetScrapLen, and TESetScrapLen (pages 23-24).

- Assembly-language information has been added. (See especially pages 25-26.)

TABLE OF CONTENTS

ABOUT THIS MANUAL

TextEdit is the part of the Macintosh User Interface Toolbox that handles basic text formatting and editing capabilities in a Macintosh application. This manual describes the TextEdit routines and data types in detail. *** Eventually it will become a chapter in the comprehensive _Inside Macintosh_ manual. ***

Like all Toolbox documentation, this manual assumes you're familiar with Lisa Pascal and the information in the following manuals:

- _Inside Macintosh: A Road Map_
- _Macintosh User Interface Guidelines_
- _Macintosh Memory Management: An Introduction_
- _Programming Macintosh Applications in Assembly Language_, if you're using assembly language

You should also be familiar with:

- the basic concepts and structures behind QuickDraw, particularly points, rectangles, grafPorts, fonts, and character style
- the Toolbox Event Manager
- the Window Manager, particularly update and activate events

ABOUT TEXTEDIT

TextEdit is a set of routines and data types that provide the basic text editing and formatting capabilities needed in an application. These capabilities include:

- inserting new text
- deleting characters that are backspaced over
- translating mouse activity into text selection
- scrolling text within a window
- deleting selected text and possibly inserting it elsewhere, or copying text without deleting it

The TextEdit routines follow the Macintosh User Interface Guidelines; using them ensures that your application will present a consistent user interface. For example, the Dialog Manager uses TextEdit for text editing in dialog boxes.

TextEdit supports these standard features:

- Selecting text by clicking and dragging with the mouse, double-clicking to select words. To TextEdit, a word is any series of printing characters, excluding spaces (ASCII code $2Ø) but including nonbreaking spaces (ASCII code $CA).

- Extending or shortening the selection by Shift-clicking.

- Inverse highlighting of the current text selection, or display of a blinking vertical bar at the insertion point.

- Word wraparound, which prevents a word from being split between lines when text is drawn.

- Cutting (or copying) and pasting within an application via the Clipboard. TextEdit puts text you cut or copy into the TextEdit scrap.

(note)

> The TextEdit scrap is used only by TextEdit; it's not the same as the "desk scrap" used by the Scrap Manager. To support cutting and pasting between applications, or between applications and desk accessories, you must transfer information between the two scraps.

Although TextEdit is useful for many standard text editing operations, there are some additional features that it doesn't support. TextEdit does **not** support:

- use of more than one font or stylistic variation in a single edit record

- fully justified text (text aligned with both the left and right margins)

- "intelligent cut and paste" (adjusting spaces between words during cutting and pasting)

- tabs

TextEdit also provides "hooks" for implementing some features such as automatic scrolling or a more precise definition of a word than that given above.

EDIT RECORDS

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 specifying a destination rectangle in which to draw the text and a view rectangle in which the text will be visible. TextEdit incorporates the rectangles and the drawing environment of the current grafPort into an edit record, and returns a handle of type TEHandle 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.

For most operations, you don't need to know the exact structure of an edit record; TextEdit routines access the record for you. However, to support some operations, such as scrolling, you need to access the fields of the edit record directly. 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).

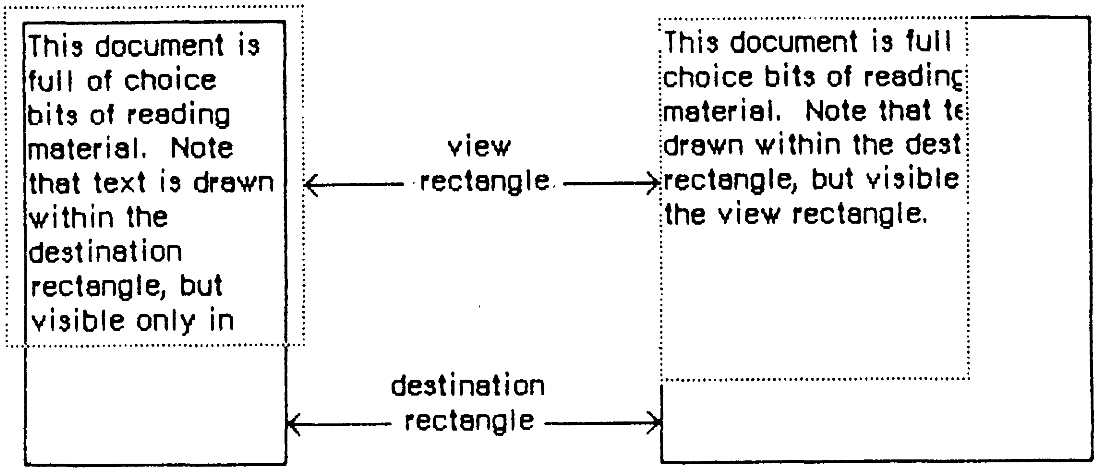

This document is full of choice bits of reading material. Note that text is drawn within the destination rectangle, but visible only in
view rectangle
destination rectangle
This document is full choice bits of readinç material. Note that te drawn within the dest rectangle, but visible the view rectangle.

Figure 1. Destination and View Rectangles

You specify both rectangles in the local coordinates of the grafPort. To ensure that the first and last characters in each line are legible in a document window, you may want to inset the destination rectangle at least four pixels from the left and right edges of the grafPort's portRect (2Ø pixels from the right edge if there's a scroll bar or size box).

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.

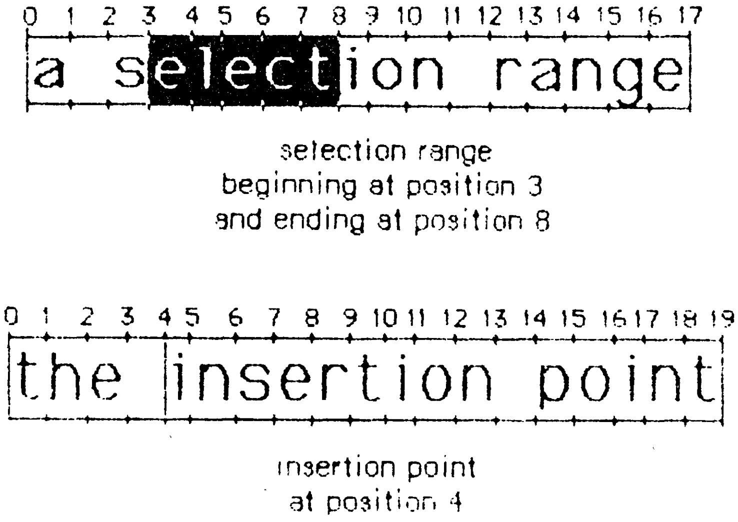
Normally, at the right edge of the destination rectangle, the text automatically wraps around to the left edge to begin a new line. A new line also begins where explicitly specified by a Return character in the text. Word wraparound ensures that no word is ever split between lines unless it's too long to fit entirely on one line, in which case it's split at the right edge of the destination rectangle.

The Selection Range

In the text editing environment, a character position is an index into the text, with position Ø 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 where the next editing operation will occur. For example, the procedures that cut or copy from the text of an edit record do so to the current selection range.

The selection range, which is inversely highlighted when the window is active, extends from the beginning character position to the end character position. Figure 2 shows a selection range between positions

3 and 8, consisting of five characters (the character at position 8 isn't included). The end position of a selection range may be 1 greater than the position of the last character of the text, so that the selection range can include the last character.



selection range
beginning at position 3
and ending at position 8

insertion point
at position 4

Figure 2. Selection Range and Insertion Point

If the selection range is empty--that is, its beginning and end positions are the same--that position is the text's insertion point, the position where characters will be inserted. By default, it's marked with a blinking caret. 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".

(note)

> 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 (|).

If you call a procedure to insert characters when there's a selection range of one or more characters rather than an insertion point, the editing procedure automatically deletes the selection range and replaces it with an insertion point before inserting the characters.

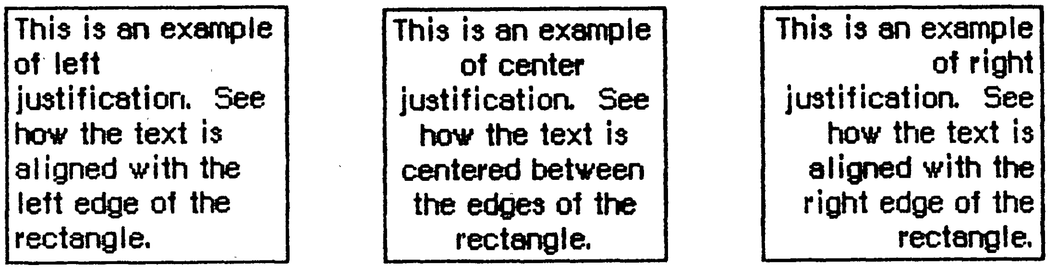
Justification

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 supported by TextEdit 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 each line of text between the left and right edges of the destination rectangle.

- Right justification aligns the text with the right edge of the destination rectangle.



This is an example
of left
justification. See
how the text is
aligned with the
left edge of the
rectangle.

This is an example
of center
justification. See
how the text is
centered between
the edges of the
rectangle.

This is an example
of right
justification. See
how the text is
aligned with the
right edge of the
rectangle.

Figure 3. Justification

(note)
Trailing spaces on a line are ignored for justification. For example, "Fred" and "Fred " will be aligned identically. (Leading spaces are not ignored.)

TextEdit provides three predefined constants for setting the justification:

```
CONST teJustLeft   = Ø;
      teJustCenter = 1;
      teJustRight  = -1;
```

The TERec Data Type

The structure of an edit record is given here. Some TextEdit features are available only if you access fields of the edit record directly.

```
  TYPE TERec = RECORD
                 destRect:   Rect;     {destination rectangle}
                 viewRect:   Rect;     {view rectangle}
                 selRect:    Rect;     {used from assembly language}
                 lineHeight: INTEGER;  {for line spacing}
                 fontAscent: INTEGER;  {caret/highlighting position}
                 selPoint:   Point;    {used from assembly language}
                 selStart:   INTEGER;  {start of selection range}
                 selEnd:     INTEGER;  {end of selection range}
                 active:     INTEGER;  {used internally}
                 wordBreak:  ProcPtr;  {for word break routine}
                 clikLoop:   ProcPtr;  {for click loop routine}
                 clickTime:  LONGINT;  {used internally}
                 clickLoc:   INTEGER;  {used internally}
                 caretTime:  LONGINT;  {used internally}
                 caretState: INTEGER;  {used internally}
                 just:       INTEGER;  {justification of text}
                 teLength:   INTEGER;  {length of text}
                 hText:      Handle;   {text to be edited}
                 recalBack:  INTEGER;  {used internally}
                 recalLines: INTEGER;  {used internally}
                 clikStuff:  INTEGER;  {used internally}
                 crOnly:     INTEGER;  {if <Ø, new line at Return only}
                 txFont:     INTEGER;  {text font}
                 txFace:     Style;    {character style}
                 txMode:     INTEGER;  {pen mode}
                 txSize:     INTEGER;  {font size}
                 inPort:     GrafPtr;  {grafPort}
                 highHook:   ProcPtr;  {used from assembly language}
                 caretHook:  ProcPtr;  {used from assembly language}
                 nLines:     INTEGER;  {number of lines}
                 lineStarts: ARRAY[Ø..16ØØØ] OF INTEGER
                                       {positions of line starts}
               END;
```

(warning)

> Don't change any of the fields marked "used internally"-- these exist solely for internal use among the TextEdit routines.

The destRect and viewRect fields specify the destination and view rectangles, respectively.

The lineHeight and fontAscent fields have to do with the vertical spacing of the lines of text, and where the caret or highlighting of the selection range is drawn relative to the text. The fontAscent field specifies how far above the base line the pen is positioned to begin drawing the caret or highlighting. For single-spaced text, this is the ascent of the text in pixels (the height of the tallest characters in the font from the base line). The lineHeight field specifies the vertical distance from the ascent line of one line of text down to the ascent line of the next. For single-spaced text, this is the same as the font size, but in pixels. The values of the lineHeight and fontAscent fields for single-spaced text are shown in Figure 4. For more information on fonts, see the Font Manager manual.

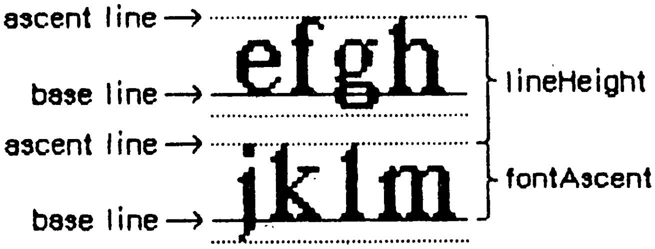


Figure 4. LineHeight and FontAscent

If you want to change the vertical spacing of the text, you should change both the lineHeight and fontAscent fields by the same amount, otherwise the placement of the caret or highlighting of the selection range may not look right. For example, to double the line spacing, add the value of lineHeight to both fields. (This doesn't change the size of the characters; it affects only the spacing between lines.) If you change the size of the text, you should also change these fields; you can get font measurements you'll need with the QuickDraw procedure GetFontInfo.

Assembly-language note: The selPoint field (whose assembly-language offset is named teSelPoint) contains the point selected with the mouse, in the local coordinates of the current grafPort. You'll need this for hit-testing if you use the routine pointed to by the global variable TEDoText (see "Advanced Routines" in the "TextEdit Routines" section).

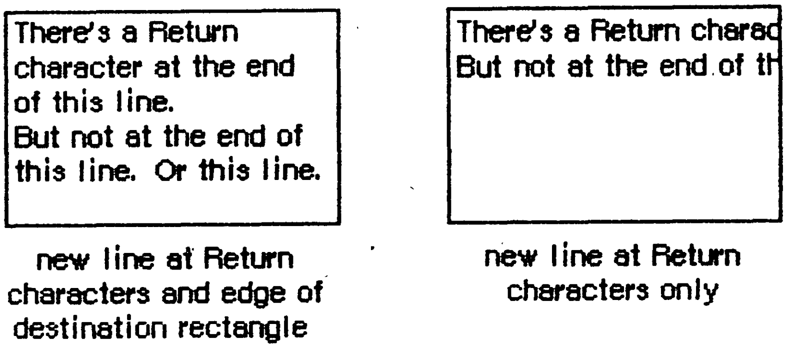
The selStart and selEnd fields specify the character positions of the beginning and end of the selection range. Remember that character position Ø refers to the first character, and that the end of a selection range can be 1 greater than the position of the last character of the text.

The wordBreak field lets you change TextEdit's definition of a word, and the clikLoop field lets you implement automatic scrolling. These two fields are described in separate sections below.

The just field specifies the justification of the text. (See "Justification", above.)

The teLength field contains the number of characters in the text to be edited, and the hText field is a handle to 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 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 faster than word wraparound, and is useful in an application similar to a programming-language editor, where you may not want a single line of code to be split onto two lines.



new line at Return characters and edge of destination rectangle

new line at Return characters only

Figure 5. New Lines

The txFont, txFace, txMode, and txSize fields specify the font, character style, pen mode, and font size, respectively, of all the text in the edit record. (See the QuickDraw manual for more details about these characteristics.) If you change one of these values, the entire text of this edit record will have the new characteristics when it's redrawn. If you change the txSize field, remember to change the lineHeight and fontAscent fields, too.

The inPort field contains a pointer to the grafPort associated with this edit record.

(warning)

> The current port is **not** preserved when TextEdit is called; you must preserve it before all calls to TextEdit routines.

> Assembly-language note: The highHook and caretHook fields--at the offsets teHiHook and teCarHook in assembly language--contain the addresses of routines that deal with text highlighting and the caret. These routines pass arguments in registers; the application must save and restore the registers.
>
> If you store the address of a routine in teHiHook, that routine will be used instead of the QuickDraw procedure InvertRect whenever a selection range is to be highlighted. The routine

can destroy the contents of registers AØ, A1, DØ, D1, and D2. On entry, A3 should be a dereferenced handle to a locked edit record; teSelRect(A3) is the rectangle enclosing the text being highlighted. For example, if you store the address of the following routine in teHiHook, selection ranges will be underlined instead of inverted:

```
UnderHigh
    PEA     teSelRect(A3)       ;get address of rectangle to be
    MOVE.L  (SP),AØ             ; highlighted
    MOVE    bottom(AØ),top(AØ)  ;make the top coordinate equal to
    SUBQ    #1,top(AØ)          ; the bottom coordinate minus 1
    _InverRect                  ;invert the resulting rectangle
    RTS
```

The routine whose address is stored in teCarHook acts exactly the same way as the teHiHook routine, but on the caret instead of the selection highlighting, allowing you to change the appearance of the caret. The routine is called with teSelRect(A3) containing the rectangle that encloses the caret.

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 16ØØ1 elements to comply with Pascal range checking; it's actually a dynamic data structure having only as many elements as needed. You shouldn't change the elements of lineStarts.

(note)

The values of selStart, selEnd, and the elements of the lineStarts array are stored internally as unsigned integers.

The WordBreak Field

The wordBreak field of an edit record lets you specify the record's word break routine--the routine that determines the "word" that's highlighted when the user double-clicks in the text, and the position at which text is wrapped around at the end of a line. The default routine breaks words at any character with an ASCII value of $2Ø or less (the space character or nonprinting control characters).

Normally the word break routine is written in assembly language. To write it in Pascal, you must declare it as follows:

```
FUNCTION PasWordBreak (text: Ptr; charPos: INTEGER) : BOOLEAN;
```

The function must be named "PasWordBreak". It should return TRUE to break a word at the character at position charPos in the specified text, or FALSE not to break there. To access PasWordBreak, set:

```
myEditRec^^.wordBreak := @AsmWordBreak
```

AsmWordBreak is an assembly-language procedure provided for the convenience of Pascal programmers. It sets the necessary registers and calls PasWordBreak.

Assembly-language note: You can set this field to point to your own assembly-language word break routine instead of using AsmWordBreak. The registers must contain the following:

```
On entry     AØ:  pointer to text
             DØ:  character position (word)

On exit      Z (zero) condition code:
                  Ø to break at specified character
                  1 not to break there
```

The ClikLoop Field

The clikLoop field contains the address of a routine that's called repeatedly (by the TEClick procedure, described below) as long as the mouse button is held down within the text. You can use this to implement the automatic scrolling of text when the user is making a selection and drags the cursor out of the view rectangle.

The click loop routine, like the word break routine, is normally written in assembly language. To write it in Pascal, you must declare it as follows:

```
FUNCTION PasClikLoop : BOOLEAN;
```

The function must be named "PasClikLoop". It should return TRUE. To access PasClikLoop, set:

```
myEditRec^^.clikLoop := @AsmClikLoop
```

AsmClikLoop is an assembly-language procedure provided for the convenience of Pascal programmers. It sets the necessary registers and calls PasClikLoop.

An automatic scrolling routine might check the mouse position, and call a scrolling routine if the mouse position is outside the view rectangle. (The scrolling routine can be the same routine that the Control Manager function TrackControl calls.) The handle to the current edit record should be kept as a global variable so the scrolling routine can access it.

(warning)
Returning FALSE from PasClikLoop tells the TEClick procedure that the mouse button has been released, which aborts TEClick.

Assembly-language note: Your routine should set register DØ to 1, and preserve register D2. (Returning Ø in register DØ aborts TEClick.)

USING TEXTEDIT

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.

When you've finished working with the text of an edit record, you can get a handle to the text as a packed array of characters with the TEGetText function.

(note)
To convert text from an edit record to a Pascal string, you can use the Dialog Manager procedure GetIText, passing it the text handle from the edit record.

When you're completely done with an edit record and want to dispose of it, call TEDispose.

(note)
To change the cursor to an I-beam, you can call the Toolbox Utility function GetCursor and the QuickDraw procedure SetCursor. The resource ID for the I-beam cursor is defined in the ToolBox Utilities as the constant iBeamCursor.

To make a blinking caret appear at the insertion point, call the TEIdle procedure as often as possible (at least once each time through the main event loop); if it's not called often enough, the caret will blink irregularly.

When a mouse-down event occurs in the view rectangle (and the window is active) call the TEClick procedure. TEClick controls the placement and highlighting of the selection range, including supporting use of the Shift key to make extended selections.

Key-down, auto-key, and mouse events that pertain to text editing can be handled by several TextEdit procedures:

- TEKey inserts characters and deletes characters backspaced over.
- TECut transfers the selection range to the TextEdit scrap, removing the selection range from the text.
- TEPaste inserts the contents of the TextEdit scrap. 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 TextEdit 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). You can use TEDelete to implement the Clear command.
- TEInsert inserts specified text. You can use this to combine two or more documents. TEDelete and TEInsert do not modify the scrap, so they're useful for implementing the Undo command.

After each editing procedure, TextEdit redraws the text if necessary from the insertion point to the end of the destination rectangle. You never have to set the selection range or insertion point yourself; TEClick and the editing procedures leave it where it should be. If you want to modify the selection range directly, however--to highlight an initial default name or value, for example--you can use the TESetSelect procedure.

When GetNextEvent reports an update event for a text editing window, call TEUpdate--along with the Window Manager procedures BeginUpdate and EndUpdate--to redraw the text.

(note)

> 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 GetNextEvent reports an activate event for a text editing window. TEActivate simply highlights the selection range or displays a caret at the insertion point; TEDeactivate unhighlights the selection range or removes the caret.

To specify the justification of the text, you can use TESetJust. If you change the justification, be sure to call TEUpdate to redraw the text.

To scroll text within the view rectangle, you can use the TEScroll procedure.

The TESetText procedure lets you change the text being edited. For example, if your application has several separate pieces of text that must be edited one at a time, you don't have to allocate an edit record for each of them. Allocate a single edit record, then use TESetText to change the text. (This is the method used in dialog boxes.)

(note)

> TESetText actually makes a copy of the text to be edited. Advanced programmers can save space by storing a handle to the text in the hText field of the edit record itself, then calling TECalText to recalculate the beginning of each line.

If you ever want to draw noneditable text in any given rectangle, you can use the TextBox procedure.

To implement cutting and pasting of text between different applications, or between applications and desk accessories, you need to transfer the text between the TextEdit scrap (which is a private scrap used only by TextEdit) and the Scrap Manager's desk scrap. You can do this using the functions TEFromScrap and TEToScrap. For programmers who wish to access scrap information directly, the low-level routines TEScrapHandle, TEGetScrapLen, and TESetScrapLen are also provided. (See the Scrap Manager manual for more information about scrap handling.)

TEXTEDIT ROUTINES

Initialization and Allocation

PROCEDURE TEInit;

TEInit initializes TextEdit by allocating a handle for the TextEdit scrap. The scrap is initially empty. Call this procedure once and only once at the beginning of your program.

(note)

> You should call TEInit even if your application doesn't use TextEdit, so that desk accessories and dialog and alert boxes will work correctly.

FUNCTION TENew (destRect,viewRect: Rect) : TEHandle;

TENew allocates a handle for the text, creates 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. The destination rectangle must always be at least as wide as the first character drawn (about 2∅ pixels is usually a good width). The view rectangle must not be empty (for example, don't make its right edge less than its left edge if you don't want any text visible--specify a rectangle off the screen instead).

Call TENew 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 ∅.

(note)
The caret won't appear until you call TEActivate.

```
PROCEDURE TEDispose (hTE: TEHandle);
```

TEDispose releases the memory allocated for the edit record and text specified by hTE. Call this procedure when you're completely through with an edit record.

Accessing the Text of an Edit Record

```
PROCEDURE TESetText (text: Ptr; length: LONGINT; hTE: TEHandle);
```

TESetText incorporates a copy of the specified text 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 doesn't affect the text drawn in the destination rectangle, so call TEUpdate afterward if necessary. TESetText doesn't dispose of any text currently in the edit record.

```
FUNCTION TEGetText (hTE: TEHandle) : CharsHandle;
```

TEGetText returns a handle to the text of the specified edit record. The result is the same as the handle in the hText field of the edit record, but has the CharsHandle data type, which is defined as:

```
TYPE CharsHandle = ^CharsPtr;
     CharsPtr    = ^Chars;
     Chars       = PACKED ARRAY[∅..32∅∅∅] OF CHAR;
```

You can get the length of the text from the teLength field of the edit record.

The Insertion Point and 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. (The caret appears only when the window containing that text is active, of course.) 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.

(note)

> The initial minimum blink interval setting is 3Ø ticks. The user can adjust this setting with the Control Panel desk accessory.

To provide a constant frequency of blinking, you should call TEIdle as often as possible--at least once each 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.

(note)

> You actually need to call TEIdle only when the window containing the text is active.

PROCEDURE TEClick (pt: Point; extend: BOOLEAN; hTE: TEHandle);

TEClick controls the placement and highlighting of the selection range as determined by mouse events. Call TEClick whenever a mouse-down event occurs in the view rectangle of the edit record specified by hTE, and the window associated with that edit record is active. TEClick keeps control until the mouse button is released. Pt is the mouse location (in local coordinates) at the time the button was pressed, obtainable from the event record.

(note)

> 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.

Pass TRUE for the extend parameter if the Event Manager indicates that the Shift key was held down at the time of the click (to extend the selection).

TEClick unhighlights the old selection range unless the selection range is being extended. If the mouse moves, meaning that a drag is occurring, TEClick expands or shortens the selection range accordingly. In the case of a double-click, the word under the cursor becomes the selection range; dragging expands or shortens the selection a word at a time.

PROCEDURE TESetSelect (selStart,selEnd: LONGINT; hTE: TEHandle);

TESetSelect sets the selection range to the text between selStart and selEnd in the text specified by hTE. The old selection range is unhighlighted, and the new one is highlighted. If selStart equals selEnd, the selection range is an insertion point, and a caret is displayed.

SelEnd and selStart can range from Ø to 32767. If selEnd is anywhere beyond the last character of the text, the position just past the last character is used.

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 window containing the edit record 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 window containing the edit record has become inactive.

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 selection range or the character immediately to the left of the insertion point is deleted. TEKey redraws the text as necessary. Call TEKey every time the Toolbox Event Manager function GetNextEvent reports a keyboard event that your application decides should be handled by TextEdit.

(note)

> TEKey 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 the Command key).

PROCEDURE TECut (hTE: TEHandle);

TECut removes the selection range from the text specified by hTE and places it in the TextEdit scrap. The text is redrawn as necessary. Anything previously in the scrap is lost. (See Figure 6.) If the selection range is an insertion point, the scrap is emptied.

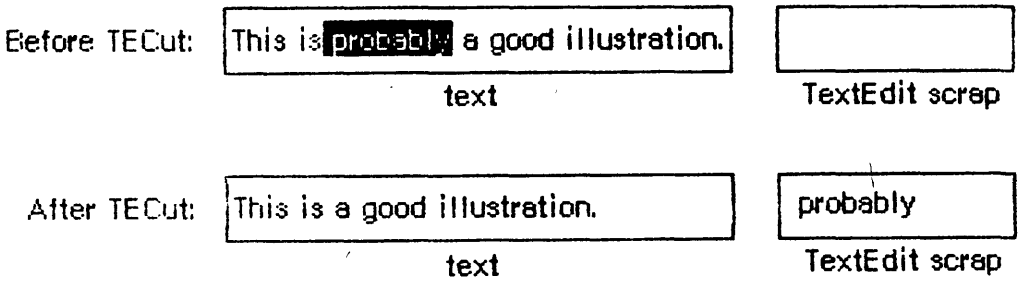


Figure 6. Cutting

PROCEDURE TECopy (hTE: TEHandle);

TECopy copies the selection range from the text specified by hTE into the TextEdit 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 contents of the TextEdit scrap, and leaves an insertion point just past the inserted text. (See Figure 7.) The text is redrawn as necessary. If the scrap is empty, the selection range is deleted. If the selection range is an insertion point, TEPaste just inserts the scrap there.

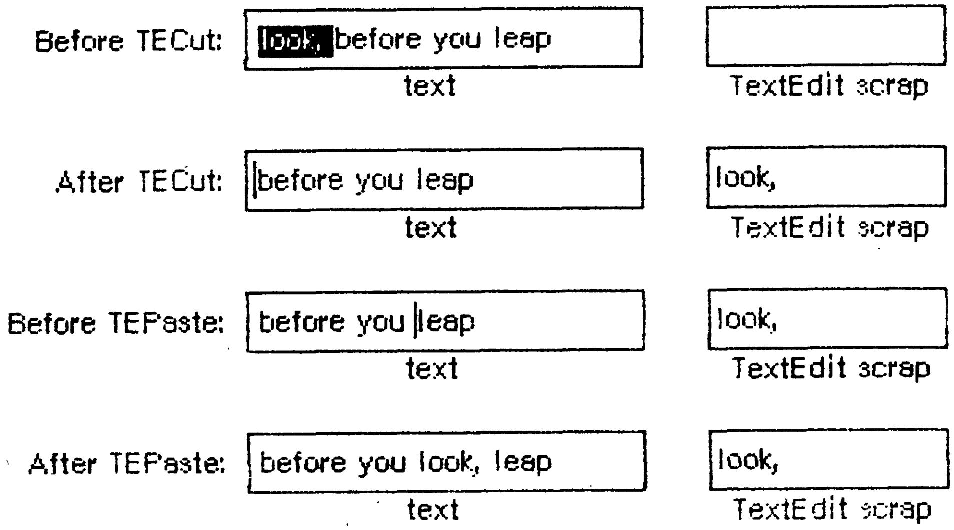


Figure 7. Cutting and Pasting

PROCEDURE TEDelete (hTE: TEHandle);

TEDelete removes the selection range from the text specified by hTE, and redraws the text as necessary. TEDelete is 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, redrawing the text as necessary. The text parameter points to the text to be inserted, and the length parameter indicates the number of characters to be inserted. TEInsert doesn't affect either the current selection range or the scrap.

Text Display and Scrolling

PROCEDURE TESetJust (just: INTEGER, hTE: TEHandle);

TESetJust sets the justification of the text specified by hTE to just. (See "Justification" under "Edit Records".) TextEdit provides three predefined constants for setting justification:

```
CONST teJustLeft   = Ø;
      teJustCenter = 1;
      teJustRight  = -1;
```

By default, text is left-justified. If you change the justification, call TEUpdate after TESetJust, to redraw the text with the new justification.

PROCEDURE TEUpdate (rUpdate: Rect; hTE: TEHandle);

TEUpdate draws the text specified by hTE within the rectangle specified by rUpdate. The rUpdate rectangle must be given in the coordinates of the current grafPort. Call TEUpdate every time the Toolbox Event Manager function GetNextEvent reports an update event for a text editing window--after you call the Window Manager procedure BeginUpdate, and before you call EndUpdate.

Normally you'll do the following when an update event occurs:

```
BeginUpdate(myWindow);
EraseRect(myWindow^.portRect);
TEUpdate(myWindow^.portRect,hTE);
EndUpdate(myWindow)
```

If you don't include the EraseRect call, the caret may sometimes remain visible when the window is deactivated.

PROCEDURE TextBox (text: Ptr; length: LONGINT; box: Rect; just:
INTEGER);

TextBox draws the specified text in the rectangle indicated by the box parameter, with justification just. (See "Justification" under "Edit Records".) The text parameter points to the text, and the length parameter indicates the number of characters to draw. The rectangle is specified in local coordinates, and must be at least as wide as the first character drawn (about 2Ø pixels is usually a good width). TextBox does not create an edit record, nor can the text that it draws be edited; it's used solely for drawing text. For example:

```
str := 'String in a box';
SetRect(r,1ØØ,1ØØ,2ØØ,2ØØ);
TextBox(POINTER(ORD(@str)+1),LENGTH(str),r,teJustCenter);
FrameRect(r)
```

Because Pascal strings start with a length byte, you must advance the pointer one position past the beginning of the string to point to the start of the text.

PROCEDURE TEScroll (dh,dv: INTEGER; hTE: TEHandle);

TEScroll 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,-hTE^^.lineHeight,hTE)
```

scrolls the text up one line. Remember that you scroll text **up** when the user clicks in the scroll arrow pointing **down**. The destination rectangle is offset by the amount you scroll.

(note)

> To implement automatic scrolling, you store the address of a routine in the clikLoop field of the edit record, as described above under "The TERec Data Type".

Scrap Handling

The TEFromScrap and TEToScrap functions return a result code of the type OSErr (defined as INTEGER in the Operating System Utilities) indicating whether an error occurred. If no error occurred, they return the result code

```
        CONST noErr = Ø  {no error}
```

Otherwise, they return an Operating System result code indicating an error. (See the Operating System Utilities manual for a list of all result codes.)

FUNCTION TEFromScrap : OSErr; [Not in ROM]

TEFromScrap copies the desk scrap to the TextEdit scrap.

> Assembly-language note: From assembly language, you can store a handle to the desk scrap in the global variable TEScrpHandle, and the length of the desk scrap in global variable TEScrpLength; you can get these values with the Scrap Manager function GetScrap.

FUNCTION TEToScrap : OSErr; [Not in ROM]

TEToScrap copies the TextEdit scrap to the desk scrap.

(warning)

> You must call the Scrap Manager function ZeroScrap to initialize the desk scrap or clear its previous contents before calling TEToScrap.

Assembly-language note: From assembly language, you can call the Scrap Manager function PutScrap; you can get the values you need from the global variables TEScrpHandle and TEScrpLength.

FUNCTION TEScrapHandle : Handle; [Not in ROM]

TEScrapHandle returns a handle to the TextEdit scrap.

Assembly-language note: The global variable TEScrpHandle contains a handle to the TextEdit scrap.

FUNCTION TEGetScrapLen : LONGINT; [Not in ROM]

TEGetScrapLen returns the size of the TextEdit scrap in bytes.

Assembly-language note: The global variable TEScrpLength contains the size of the TextEdit scrap in bytes.

PROCEDURE TESetScrapLen (length: LONGINT); [Not in ROM]

TESetScrapLen sets the size of the TextEdit scrap to the given number of bytes.

Assembly-language note: From assembly language, you can set the global variable TEScrpLength.

Advanced Routines

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 any other field that affects the number of characters per line.

(note)

> There are two ways to specify text to be edited. The easiest method is to use TESetText, which takes an existing edit record, creates a copy of the specified text, and stores a handle to the copy in the edit record. You can instead directly change the hText field of the edit record, and then call TECalText to recalculate the lineStarts array to match the new text. If you have a lot of text, you can use the latter method to save space.

Assembly-language note: The global variable TEReCal contains the address of the routine called by TECalText to recalculate the line starts and set the first and last characters that need to be redrawn. The registers should contain the following:

On entry	A3:	dereferenced handle to the locked edit record
	D7:	change in the length of the record (word)
On exit	D2:	line start of the line containing the first character to be redrawn (word)
	D3:	position of first character to be redrawn (word)
	D4:	position of last character to be redrawn (word)

Assembly-language note: The global variable TEDoText contains the address of a multi-purpose text editing routine that advanced programmers may find useful. It lets you display, highlight, and hit-test characters, and position the pen to draw the caret. "Hit-test" means decide where to place the insertion point when the user clicks the mouse button; the point selected with the mouse is in the teSelPoint field. The registers should contain the following:

On entry	A3:	dereferenced handle to the locked edit record
	D3:	position of first character to be redrawn (word)
	D4:	position of last character to be redrawn (word)
	D7:	(word) Ø to hit-test a character 1 to highlight the selection range -1 to display the text -2 to position the pen to draw the caret
On exit	AØ:	pointer to current grafPort
	DØ:	if hit-testing, character position or -1 for none (word)

SUMMARY OF TEXTEDIT

Constants

```
CONST { Text justification }

      teJustLeft   = Ø;
      teJustCenter = 1;
      teJustRight  = -1;
```

Data Types

```
TYPE CharsHandle = ^CharsPtr;
     CharsPtr    = ^Chars;
     Chars       = PACKED ARRAY[Ø..32ØØØ] OF CHAR;

     TEHandle = ^TEPtr;
     TEPtr    = ^TERec;
```

```
    TERec    = RECORD
                  destRect:    Rect;     {destination rectangle}
                  viewRect:    Rect;     {view rectangle}
                  selRect:     Rect;     {used from assembly language}
                  lineHeight:  INTEGER;  {for line spacing}
                  fontAscent:  INTEGER;  {caret/highlighting position}
                  selPoint:    Point;    {used from assembly language}
                  selStart:    INTEGER;  {start of selection range}
                  selEnd:      INTEGER;  {end of selection range}
                  active:      INTEGER;  {used internally}
                  wordBreak:   ProcPtr;  {for word break routine}
                  clikLoop:    ProcPtr;  {for click loop routine}
                  clickTime:   LONGINT;  {used internally}
                  clickLoc:    INTEGER;  {used internally}
                  caretTime:   LONGINT;  {used internally}
                  caretState:  INTEGER;  {used internally}
                  just:        INTEGER;  {justification of text}
                  teLength:    INTEGER;  {length of text}
                  hText:       Handle;   {text to be edited}
                  recalBack:   INTEGER;  {used internally}
                  recalLines:  INTEGER;  {used internally}
                  clikStuff:   INTEGER;  {used internally}
                  crOnly:      INTEGER;  {if <Ø, new line at Return only}
                  txFont:      INTEGER;  {text font}
                  txFace:      Style;    {character style}
                  txMode:      INTEGER;  {pen mode}
                  txSize:      INTEGER;  {font size}
                  inPort:      GrafPtr;  {grafPort}
                  highHook:    ProcPtr;  {used from assembly language}
                  caretHook:   ProcPtr;  {used from assembly language}
                  nLines:      INTEGER;  {number of lines}
                  lineStarts:  ARRAY[Ø..16ØØØ] OF INTEGER
                                         {positions of line starts}
               END;
```

Routines

Initialization and Allocation

```
PROCEDURE TEInit;
FUNCTION  TENew     (destRect,viewRect: Rect) : TEHandle;
PROCEDURE TEDispose (hTE: TEHandle);
```

Accessing the Text of an Edit Record

```
PROCEDURE TESetText (text: Ptr; length: LONGINT; hTE: TEHandle);
FUNCTION  TEGetText (hTE: TEHandle) : CharsHandle;
```

The Insertion Point and Selection Range

```
PROCEDURE TEIdle       (hTE: TEHandle);
PROCEDURE TEClick      (pt: Point; extend: BOOLEAN; hTE: TEHandle);
PROCEDURE TESetSelect  (selStart,selEnd: LONGINT; hTE: TEHandle);
PROCEDURE TEActivate   (hTE: TEHandle);
PROCEDURE TEDeactivate (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);
```

Text Display and Scrolling

```
PROCEDURE TESetJust (just: INTEGER; hTE: TEHandle);
PROCEDURE TEUpdate  (rUpdate: Rect; hTE: TEHandle);
PROCEDURE TextBox   (text: Ptr; length: LONGINT; box: Rect; just: INTEGER);
PROCEDURE TEScroll  (dh,dv: INTEGER; hTE: TEHandle);
```

Scrap Handling [Not in ROM]

```
FUNCTION  TEFromScrap :    OSErr;
FUNCTION  TEToScrap :      OSErr;
FUNCTION  TEScrapHandle :  Handle;
FUNCTION  TEGetScrapLen :  LONGINT;
PROCEDURE TESetScrapLen : (length: LONGINT);
```

Advanced Routines

```
PROCEDURE TECalText (hTE: TEHandle);
```

Word Break Routine

```
FUNCTION PasWordBreak (text: Ptr; charPos: INTEGER) : BOOLEAN;

myEditRec^^.wordBreak := @AsmWordBreak
```

Click Loop Routine

```
FUNCTION PasClikLoop : BOOLEAN;

myEditRec^^.clikLoop := @AsmClikLoop
```

Assembly-Language Information

Constants

```
; Text justification

teJustLeft      .EQU    Ø
teJustCenter    .EQU    1
teJustRight     .EQU    -1
```

Edit Record Data Structure

teDestRect	Destination rectangle (8 bytes)
teViewRect	View rectangle (8 bytes)
teSelRect	Selection rectangle (8 bytes)
teLineHite	For line spacing (word)
teAscent	Caret/highlighting position (word)
teSelPoint	Point selected with mouse (long)
teSelStart	Start of selection range (word)
teSelEnd	End of selection range (word)
teWordBreak	Address of word break routine (see below)
teClikProc	Address of click loop routine (see below)
teJust	Justification of text (word)
teLength	Length of text (word)
teTextH	Handle to text
teCROnly	If <Ø, new line at Return only (byte)
teFont	Text font (word)
teFace	Character style (word)
teMode	Pen mode (word)
teSize	Font size (word)
teGrafPort	Pointer to grafPort
teHiHook	Address of text highlighting routine (see below)
teCarHook	Address of routine to draw caret (see below)
teNLines	Number of lines (word)
teLines	Positions of line starts (teNLines*2 bytes)
teRecSize	Size in bytes of above structure except teLines

Word break routine

On entry	AØ:	pointer to text
	DØ:	character position (word)
On exit	Z condition code:	Ø to break at specified character 1 not to break there

Click loop routine

On exit	DØ:	1
	D2:	must be preserved

Text highlighting routine

On entry	A3:	dereferenced handle to locked edit record

Caret drawing routine

On entry	A3:	dereferenced handle to locked edit record

Variables

TEScrpHandle	Handle to TextEdit scrap
TEScrpLength	Size in bytes of TextEdit scrap (long)
TEReCal	Address of routine to recalculate line starts (see below)
TEDoText	Address of multi-purpose routine (see below)

TEReCal routine

On entry	A3:	dereferenced handle to locked edit record
	D7:	change in length of edit record (word)
On exit	D2:	line start of line containing first character to be redrawn (word)
	D3:	position of first character to be redrawn (word)
	D4:	position of last character to be redrawn (word)

TEDoText routine

On entry	A3:	dereferenced handle to locked edit record
	D3:	position of first character to be redrawn (word)
	D4:	position of last character to be redrawn (word)
	D7:	(word) Ø to hit-test a character 1 to highlight selection range -1 to display text -2 to position pen to draw caret
On exit	AØ:	pointer to current grafPort
	DØ:	if hit-testing, character position or -1 for none (word)

GLOSSARY

ascent: The vertical distance from a font's base line to its ascent line.

ascent line: A horizontal line that coincides with the tops of the tallest characters in a font.

base line: A horizontal line that coincides with the bottom of each character in a font, excluding descenders (such as the tail of a "p").

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 text, starting at Ø for the first character.

destination rectangle: In TextEdit, the rectangle in which text is drawn.

edit record: A complete editing environment in TextEdit, which includes the text to be edited, the grafPort and rectangle in which to display the text, the arrangement of the text within the rectangle, and other editing and display information.

insertion point: An empty selection range; the character position where text will be inserted (usually marked with a blinking caret).

justification: The horizontal placement of lines of text relative to the edges of the rectangle in which the text is drawn.

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.

TextEdit scrap: The place where certain TextEdit routines store the characters most recently cut or copied from text.

view rectangle: In TextEdit, the rectangle in which text is visible.

word: In TextEdit, any series of printing characters, excluding spaces (ASCII code $2Ø) but including nonbreaking spaces (ASCII code $CA).

word wraparound: Keeping words from being split between lines when text is drawn.

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.

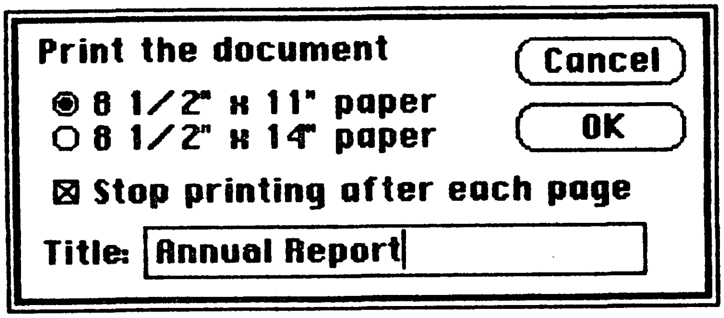
Following these descriptions are sections that will not interest all readers. There's a discussion of how to modify definitions of dialogs and alerts after they've been read from a resource file, and a section that gives the exact formats of resources related to dialogs and alerts.

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.



Print the document

● 8 1/2" x 11" paper

○ 8 1/2" x 14" paper

☒ Stop printing after each page

Title: Annual Report

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:

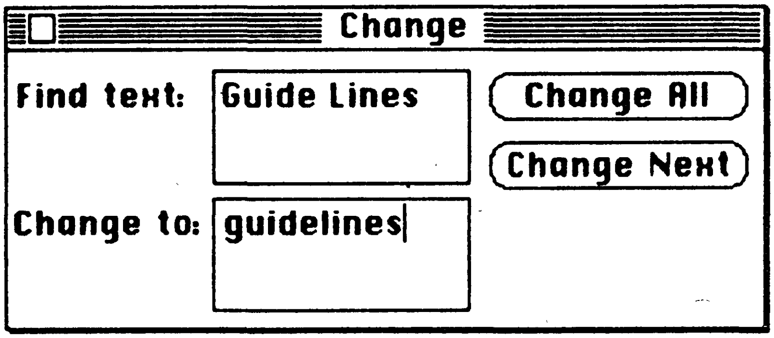
- informative or instructional text
- rectangles in which text may be entered (initially blank or containing default text that can be edited)
- controls of any kind
- graphics (icons or QuickDraw pictures)
- anything else, as defined by the application

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 <u>default button</u> 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 <u>modeless</u> 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.



Find text: Guide Lines

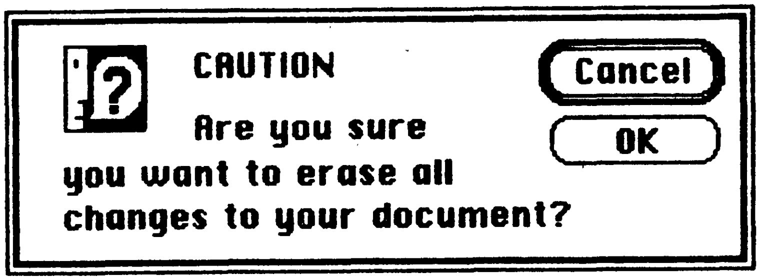
Change to: guidelines

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 <u>alert</u> mechanism provides applications with a means of reporting errors or giving warnings. An <u>alert box</u> 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.



CAUTION

Are you sure you want to erase all changes to your document?

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.

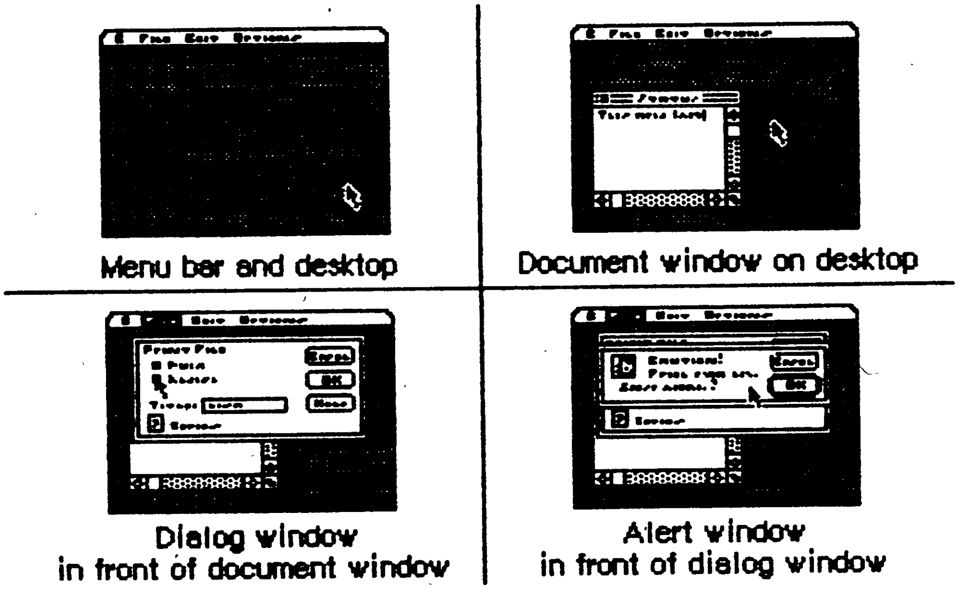
DIALOG AND ALERT WINDOWS

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.



Menu bar and desktop

Document window on desktop

Dialog window
in front of document window

Alert window
in front of dialog window

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.

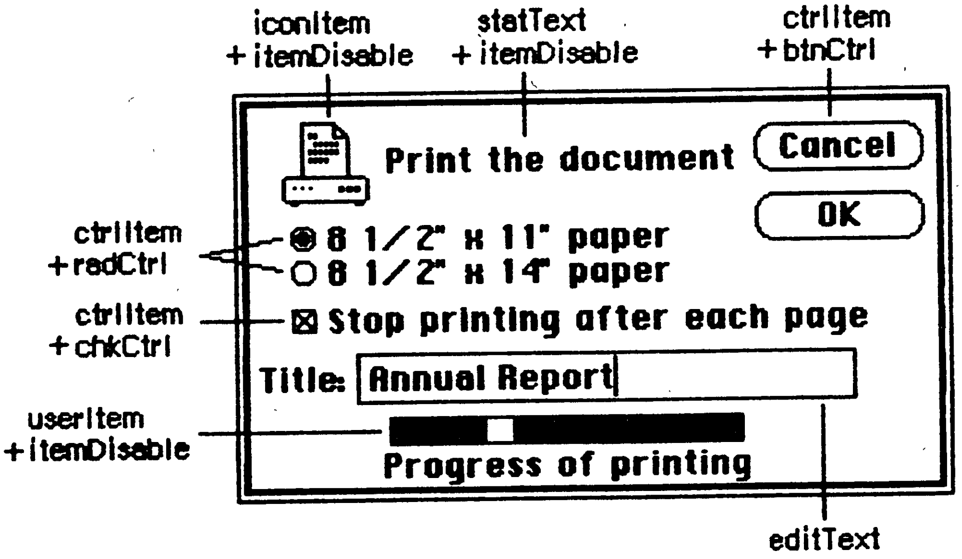
- A display rectangle, which determines the location of the item within the dialog or alert box.

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.



● 8 1/2" x 11" paper
○ 8 1/2" x 14" paper
☒ Stop printing after each page
Title: Annual Report

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 4Ø 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).

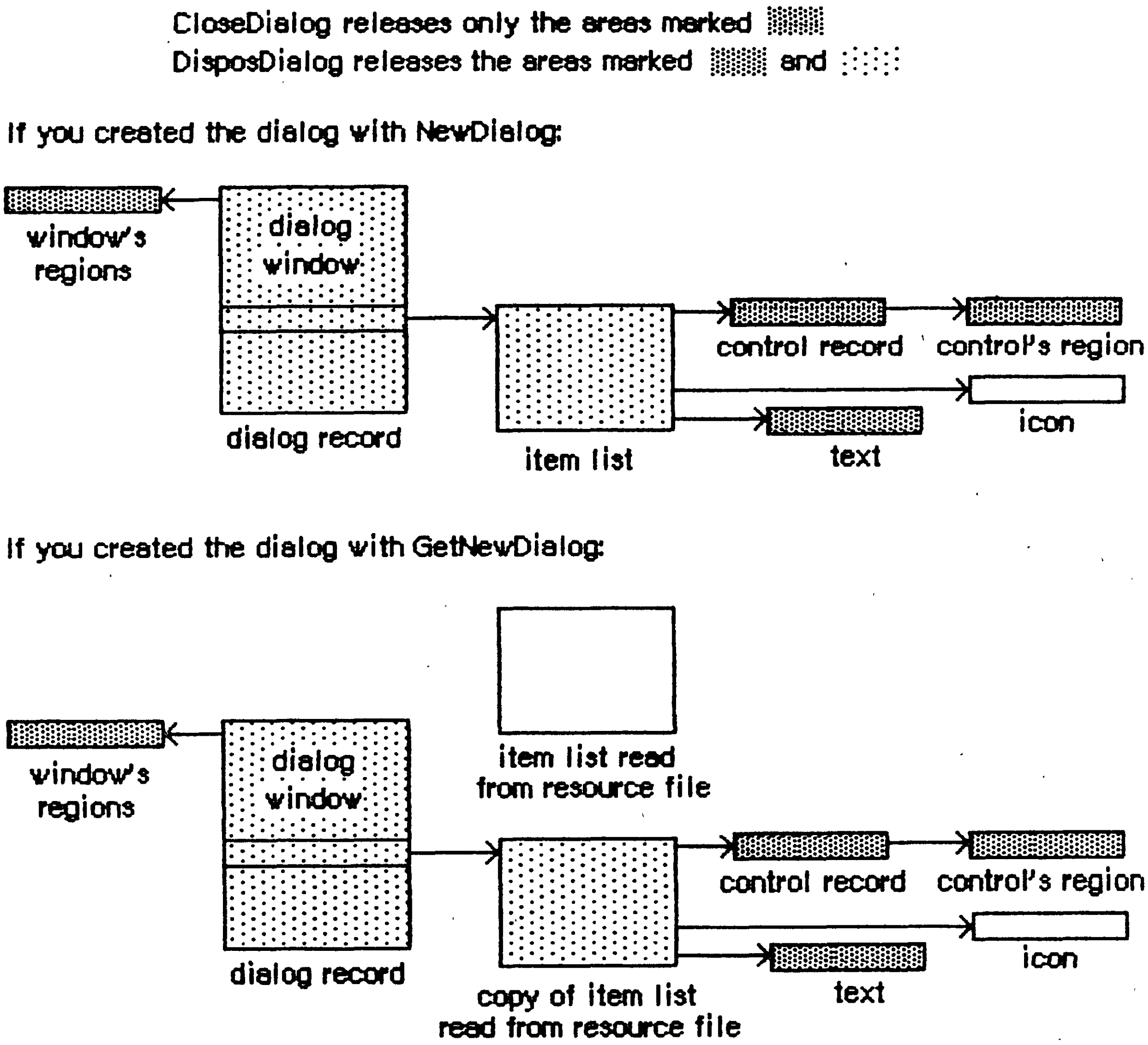

CloseDialog releases only the areas marked ▒
DisposDialog releases the areas marked ▒ and ░
If you created the dialog with NewDialog:
window's regions
dialog window
dialog record
item list
control record
control's region
icon
text
If you created the dialog with GetNewDialog:
window's regions
dialog window
dialog record
item list read from resource file
copy of item list read from resource file
control record
control's region
icon
text

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.

<u>Assembly-language note</u>: 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).

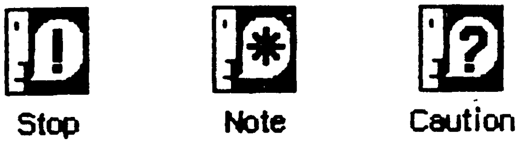


Figure 7. Standard Alert Icons

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)

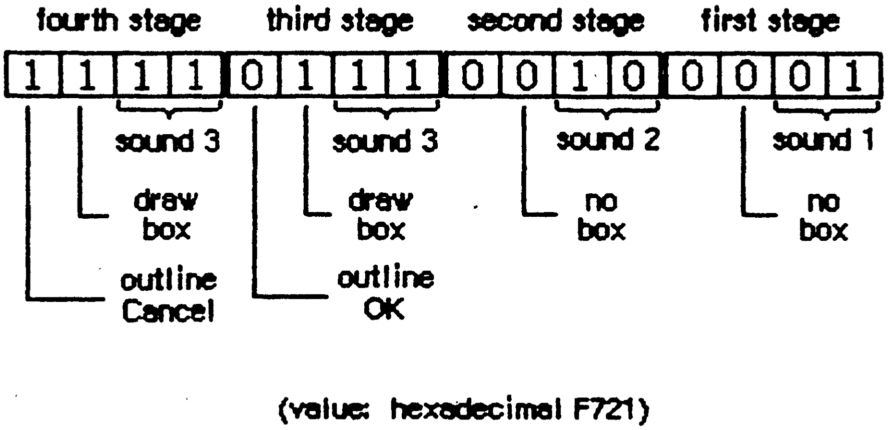


Figure 8. Sample Stages Word

(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	If item type is:	Content is:
(n is even)	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      (param∅,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.

MACINTOSH USER EDUCATION

The Desk Manager: A Programmer's Guide /DSKMGR/DESK

See Also: Inside Macintosh: A Road Map
Macintosh User Interface Guidelines
Macintosh Memory Management: An Introduction
Programming Macintosh Applications in Assembly Language
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
The Dialog Manager: A Programmer's Guide
The Menu Manager: A Programmer's Guide
The Scrap Manager: A Programmer's Guide
The Device Manager: A Programmer's Guide

Modification History:			
	First Draft (ROM 2.Ø)	Caroline Rose	2/3/83
	Erratum Added	Caroline Rose	2/28/83
	Second Draft (ROM 4)	Caroline Rose	6/14/83
	Third Draft (ROM 7)	Caroline Rose	9/26/83
	Fourth Draft	Bradley Hacker & Caroline Rose	8/22/84

ABSTRACT

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 how your application can support existing desk accessories, and tells you how to write your own desk accessories.

Summary of significant changes and additions since last draft:

- Added new information on the OpenDeskAcc function (page 7).
- Added a value for specifying the Clear command in a SystemEdit call, corrected other values, and removed the predefined constants (page 9).
- Updated "Writing Your Own Desk Accessories" to match the terminology and content of the Device Manager manual (page 11).
- Added an equate for specifying the Clear command to a desk accessory's control routine, and corrected other equates (page 14).

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 the comprehensive Inside Macintosh manual. *** You'll learn how to use the Desk Manager routines and how to write your own accessories.

Like all Toolbox documentation, this manual assumes you're familiar with Lisa Pascal and the information in the following manuals:

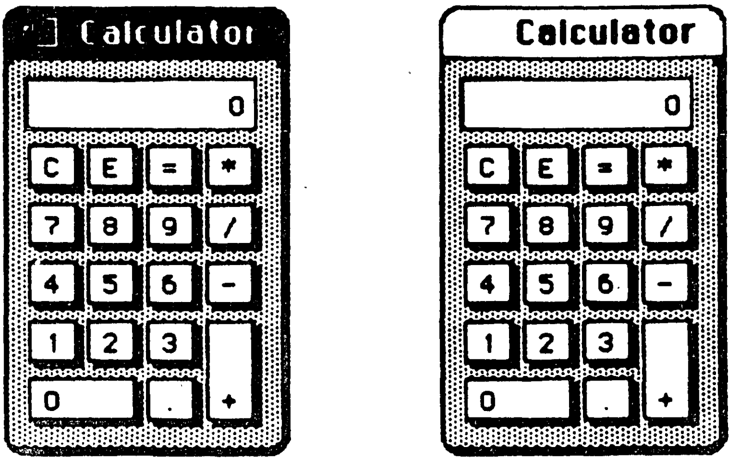
- Inside Macintosh: A Road Map
- Macintosh User Interface Guidelines
- Macintosh Memory Management: An Introduction
- Programming Macintosh Applications in Assembly Language, if you're using assembly language

You should also be familiar with:

- the basic concepts behind the Resource Manager and QuickDraw
- the Toolbox Event Manager, the Window Manager, the Menu Manager, and the Dialog Manager
- device drivers, as discussed in the Device Manager manual, if you want to write your own desk accessories

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. There are a number of standard desk accessories, such as the Calculator shown in Figure 1. You can also write your own desk accessories if you wish.

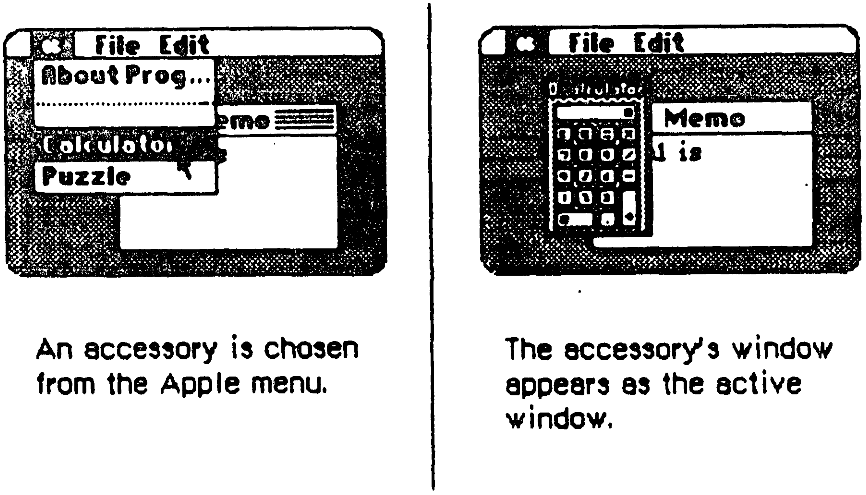


Active

Inactive

Figure 1. The Calculator Desk Accessory

The Macintosh user opens desk accessories by choosing them from the standard Apple 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.)



An accessory is chosen from the Apple menu.

The accessory's window appears as the active window.

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 frontmost becomes active.

The window associated with a desk accessory is usually a rounded-corner window (as shown in Figure 1) or a standard document window, although it can be any type of window. It may even 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 any case, the window will be a system window, as indicated by the fact that its windowKind field contains a negative value.

The Desk Manager provides a mechanism that lets standard commands chosen from the Edit menu 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 can be copied into a document prepared in MacWrite.

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. In this case, 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.

A desk accessory is actually a special type of device driver--special in that it may have its own windows and menus for interacting with the user. The value in the windowKind field of a desk accessory's window is a reference number that uniquely identifies the driver, returned by the Device Manager when the driver was opened. Desk accessories and other RAM 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 procedure AddResMenu (see the Menu Manager manual for details).

- Set up an Edit menu that includes the standard commands Undo, Cut, Copy, Paste, and Clear (in that order, with a gray line separating Undo and Cut), even if your application itself doesn't support any of these commands.

(note)

> Applications should leave enough space in the menu bar for a desk accessory's menu to be added.

When the user chooses a desk accessory from the Apple menu, call the Menu Manager procedure GetItem to get the name of the desk accessory, and then the Desk Manager function OpenDeskAcc to open and display the accessory. When a system window is active and the user chooses Close from the File menu, close the desk accessory with the CloseDeskAcc procedure.

(warning)

> Most open desk accessories allocate nonrelocatable objects (such as windows) on the heap, resulting in fragmentation of heap space. Before beginning an operation that requires a large amount of memory, your application may want to close all open desk accessories.

When the Toolbox Event Manager function GetNextEvent reports that a mouse-down event has occurred, your application should call 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, call the Desk Manager procedure SystemClick. SystemClick handles mouse-down events in system windows, routing them to desk accessories where appropriate.

(note)

> The application needn't be concerned with exactly which desk accessories are currently open.

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, and it should enable the standard editing commands Undo, Cut, Copy, Paste, and Clear, in the Edit menu. An application should disable any editing commands it doesn't support when one of its own windows becomes active.

When a mouse-down event occurs in the menu bar, and the application determines that one of the five standard editing commands 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.

Keyboard equivalents of the standard editing commands are passed on to desk accessories by the Desk Manager, not by your application.

(warning)

> The standard keyboard equivalents for the commands in the Edit menu must not be changed or assigned to other commands; the Desk Manager automatically interprets Command-Z, X, C, and V as Undo, Cut, Copy, and Paste, respectively.

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

Opening and Closing Desk Accessories

FUNCTION OpenDeskAcc (theAcc: Str255) : INTEGER;

OpenDeskAcc opens the desk accessory having the given name and displays its window (if any) as the active window. 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.

You should ignore the value returned by OpenDeskAcc. If the desk accessory is successfully opened, the function result is its driver reference number; as described under CloseDeskAcc below, you don't need this number to close the accessory. If the desk accessory can't be opened, the function result is undefined; the accessory will have taken care of informing the user of the problem (such as memory full) and not displaying itself.

(warning)

> It may occasionally happen that the current grafPort will be the desk accessory's port upon return from OpenDeskAcc. To be safe, you should bracket your call to OpenDeskAcc with calls to the QuickDraw procedures GetPort and SetPort, to save and restore the current port.

Before you open a desk accessory it's a gcod idea to determine whether there's enough memory available. Here's an example of how to do that:

```
SetResLoad(FALSE);
myResHandle := GetNamedResource('DRVR', theAcc);
size := SizeResource(myResHandle);
myHandle := NewHandle(size + 3072);
IF myHandle = NIL
  THEN {put up an alert indicating there's not enough memory}
  ELSE OpenDeskAcc(theAcc)
```

The extra 3K bytes in the argument to the Memory Manager's NewHandle function is an average amount of heap space used by desk accessories while they're running.

PROCEDURE CloseDeskAcc (refNum: INTEGER);

When a system window is active and the user chooses Close from the File menu, call CloseDeskAcc to close the desk accessory. RefNum is the driver reference number for the desk accessory, which you get from the windowKind field of its window.

The Desk Manager automatically closes a desk accessory if the user clicks its close box. Also, since the application heap is released 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 function 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 procedure 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 function 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 there's a mouse-down event in the menu bar and the user chooses one of the five standard editing commands from the Edit menu. Pass one of the following as the value of the editCmd parameter:

EditCmd	Editing command
Ø	Undo
2	Cut
3	Copy
4	Paste
5	Clear

If your Edit menu contains these five commands in the standard arrangement (the order listed above, with a gray line separating Undo and Cut), you can simply call

```
SystemEdit(menuItem - 1)
```

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.

(note)

> It's up to the application to make sure desk accessories get their editing commands that are chosen from the Edit menu. In particular, make sure your application hasn't disabled the Edit menu or any of the five standard commands when a desk accessory is activated.

Assembly-language note: The macro you invoke to call SystemEdit from assembly language is named _SysEdit.

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.

(note)

> SystemTask should be called at least every sixtieth of a second.

Advanced Routines

```
FUNCTION SystemEvent (theEvent: EventRecord) : BOOLEAN;
```

SystemEvent is called only by the Toolbox Event Manager function 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 function 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 or keyboard 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 the event 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.

(note)

> 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 File Manager function MountVol) 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 functions 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.

WRITING YOUR OWN DESK ACCESSORIES

To write your own desk accessory, you must create it as a device driver and include it in a resource file, as described in the Device Manager manual. 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 a device driver 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 a device driver is described in the Device Manager manual. The rest of this section reviews some of that information and presents additional details pertaining specifically to device drivers that are desk accessories.

A device driver begins with a few words of flags and other data, followed by offsets to the routines that do the work of the driver, an optional title, and finally the routines themselves.

byte 0	drvrFlags (word)	flags
2	drvrDelay (word)	number of ticks between periodic actions
4	drvrEMask (word)	desk accessory event mask
6	drvrMenu (word)	menu ID of menu associated with driver
8	drvrOpen (word)	offset to open routine
10	drvrPrime (word)	offset to prime routine
12	drvrCtl (word)	offset to control routine
14	drvrStatus (word)	offset to status routine
16	drvrClose (word)	offset to close routine
18	drvrName (byte)	length of driver name
19	drvrName+1 (bytes)	characters of driver name
	driver routines	

Figure 3. Desk Accessory Device Driver

Two bits in the **high-order** byte of the drvrFlags word are used especially for desk accessories:

```
        dNeedTime    .EQU   5     ;set if driver needs time for
                                  ; performing a periodic action
        dNeedLock    .EQU   6     ;set if driver will be locked in
                                  ; memory as soon as it's opened
```

Desk accessories may need to perform predefined actions periodically. For example, a clock desk accessory may want to change the time it displays every second. If the dNeedTime flag is set, the desk accessory **does** need to perform a periodic action, and the drvrDelay word contains 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 at most every sixtieth of a second, 2 means at most every thirtieth of a second, and so on. Whether the action actually occurs this frequently depends on how often the application calls the Desk Manager procedure SystemTask. SystemTask calls the desk accessory's control routine (if the time indicated by drvrDelay has elapsed), and the control routine must perform whatever predefined action is desired.

(note)

> A desk accessory cannot rely on SystemTask being called regularly or frequently by an application. If it needs precise timing it should refer to the global variable Ticks.

The drvrEMask word contains an event mask specifying which events the desk accessory can handle. If the desk accessory has a window, the event mask should include update, activate, mouse-down, and keyboard events, and must **not** include mouse-up events. When an event occurs, the Toolbox Event Manager calls SystemEvent. SystemEvent checks the drvrEMask word to determine whether the desk accessory can handle the type of event, and if so, calls the desk accessory's control routine. The control routine must perform whatever action is desired.

If the desk accessory has its own menu (or menus), the drvrMenu word contains the menu ID of the menu (or of any one of the menus); otherwise, it contains Ø. The menu ID for a desk accessory menu must be negative, and it must be different from the menu ID stored in 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.

(note)

> A practical size limit for desk accessories is about 8K bytes.

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 function GetNewDialog (or NewDialog) 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 of the window record to the driver reference number for the desk accessory, 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.

If the open routine is unable to complete all of the above tasks (if it runs out of memory, for example), it must do the following:

- Open only the minimum of data structures needed to run the desk accessory.

- Modify the code of every routine (except the close routine) so that the routine just returns (or beeps) when called.

- Modify the code of the close routine so that it disposes of only the minimum data structures that were opened.

- Display an alert indicating failure, such as "The Note Pad is not available".

The close routine closes the desk accessory, disposing of its window (if any) and all the data structures associated with it 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.

(warning)

> A driver's private storage shouldn't be in the system heap, because when an application terminates the application heap is reinitialized and the driver is lost before it can dispose of its storage.

The action taken by the control routine depends on information passed in the parameter block pointed to by AØ. A message is passed in the csCode field; this message is simply a number that tells the routine what action to take. There are nine such messages:

```
accEvent    .EQU  64    ;handle a given event
accRun      .EQU  65    ;take the periodic action, if any, for
                        ; this desk accessory
accCursor   .EQU  66    ;change cursor shape if appropriate;
                        ; generate null event if window was
                        ; created by Dialog Manager
accMenu     .EQU  67    ;handle a given menu item
accUndo     .EQU  68    ;handle the Undo command
accCut      .EQU  7Ø    ;handle the Cut command
accCopy     .EQU  71    ;handle the Copy command
accPaste    .EQU  72    ;handle the Paste command
accClear    .EQU  73    ;handle the Clear command
```

Along with the accEvent message, the control routine receives in the csParam field a pointer to an event record. The control routine must respond 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 global variable MBarEnable. (This is the negative menu ID in the device 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 by SystemTask 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 a 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. 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: csParam contains the menu ID of the desk accessory's menu and csParam+2 contains the menu item number. The control routine should take the appropriate action for when the given menu item is chosen from the menu, and then make 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 five messages--accUndo through accClear--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 Scrap Manager manual.

(note)
The control routine doesn't have to worry about savinq and restoring the current grafPort; the Desk Manager will take care of it.

(note)
You can't segment the code of a desk accessory, since the jump table is used for the application code.

A Sample Desk Accessory

*** to be supplied; meanwhile, contact Macintosh Technical Support ***

SUMMARY OF THE DESK MANAGER

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);
```

Assembly-Language Information

Constants

```
; Desk accessory flags

dNeedTime       .EQU      5      ;set if driver needs time for
                                 ; performing a periodic action
dNeedLock       .EQU      6      ;set if driver will be locked in
                                 ; memory as soon as it's opened

; Control routine messages

accEvent        .EQU     64      ;handle a given event
accRun          .EQU     65      ;take the periodic action, if any, for
                                 ; this desk accessory
accCursor       .EQU     66      ;change cursor shape if appropriate;
                                 ; generate null event if window was
                                 ; created by Dialog Manager
accMenu         .EQU     67      ;handle a given menu item
accUndo         .EQU     68      ;handle the Undo command
accCut          .EQU     7Ø      ;handle the Cut command
accCopy         .EQU     71      ;handle the Copy command
```

```
accPaste        .EQU      72       ;handle the Paste command
accClear        .EQU      73       ;handle the Clear command
```

Variable

Name	Size	Contents
MBarEnable	2 bytes	Menu ID of active desk accessory's menu

Special Macro Name

Routine name	Macro name
SystemEdit	_SysEdit

GLOSSARY

desk accessory: A "mini-application", implemented as a device driver, that can be run at the same time as a Macintosh application.

tick: A sixtieth of a second.

MACINTOSH USER EDUCATION

The Scrap Manager: A Programmer's Guide /SMGR/SCRAP

See Also: Macintosh User Interface Guidelines
Inside Macintosh: A Road Map
Macintosh Memory Management: An Introduction
QuickDraw: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Toolbox Event Manager: A Programmer's Guide
The Segment Loader: A Programmer's Guide
The Desk Manager: A Programmer's Guide
TextEdit: A Programmer's Guide
The File Manager: A Programmer's Guide
The Memory Manager: A Programmer's Guide
Programming Macintosh Applications in Assembly Language

Modification History:			
	First Draft (ROM 7)	Caroline Rose	1Ø/21/83
	Erratum Added	Caroline Rose	11/16/83
	Second Draft	Katie Withey	1/31/85

ABSTRACT

This manual describes the Scrap Manager, the part of the Macintosh User Interface Toolbox that supports cutting and pasting among applications and desk accessories.

Summary of significant changes and additions since last draft:

- Assembly-language information for the scrap is in global variables, not in offsets (page 17).
- Correct examples for using GetScrap are given (pages 12-13).
- Many other minor changes have been made throughout the manual.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Scrap Manager, the part of the Macintosh User Interface Toolbox that supports cutting and pasting among applications and desk accessories. *** Eventually it will become part of the comprehensive Inside Macintosh manual. ***

Like all Toolbox documentation, this manual assumes you're familiar with Lisa Pascal and the information in the following manuals:

- Inside Macintosh: A Road Map

- Macintosh User Interface Guidelines

- Macintosh Memory Management: An Introduction

- Programming Macintosh Applications in Assembly Language, if you're using assembly language

You should also be familiar with:

- resources, as discussed in the Resource Manager manual

- QuickDraw pictures

- the Toolbox Event Manager

ABOUT THE SCRAP MANAGER

The Scrap Manager is a set of routines and data types that enable Macintosh applications to 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, or it can set up its own private scrap for this purpose. The format of the private scrap can be whatever the application likes, since no other application will use it. For example, an application can simply maintain a pointer to data that's been cut or copied.

(note)

> The TextEdit scrap is a private scrap for applications that use TextEdit. TextEdit provides its own routines for dealing with its scrap.

From the user's point of view, there's a single place where all cut or copied data resides; 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 provide 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 between two desk accessories. Desk accessories that display text will usually 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.

(note)

> The Scrap Manager was designed to transfer **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, in a word processor or in the Calculator, the data is text; in a graphics application it's a picture. The amount of information retained about the data being 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 word processor document, 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 some control over how much information is retained when data is transferred.

The desk scrap is usually stored in memory, but can be stored on the disk (in the scrap file) if there's not enough room for it in memory. The scrap 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. The routines that access the scrap keep track of whether it's in memory or on the disk.

OVERVIEW OF THE DESK SCRAP

The desk scrap is initially located in the application heap; a handle to it is stored 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.

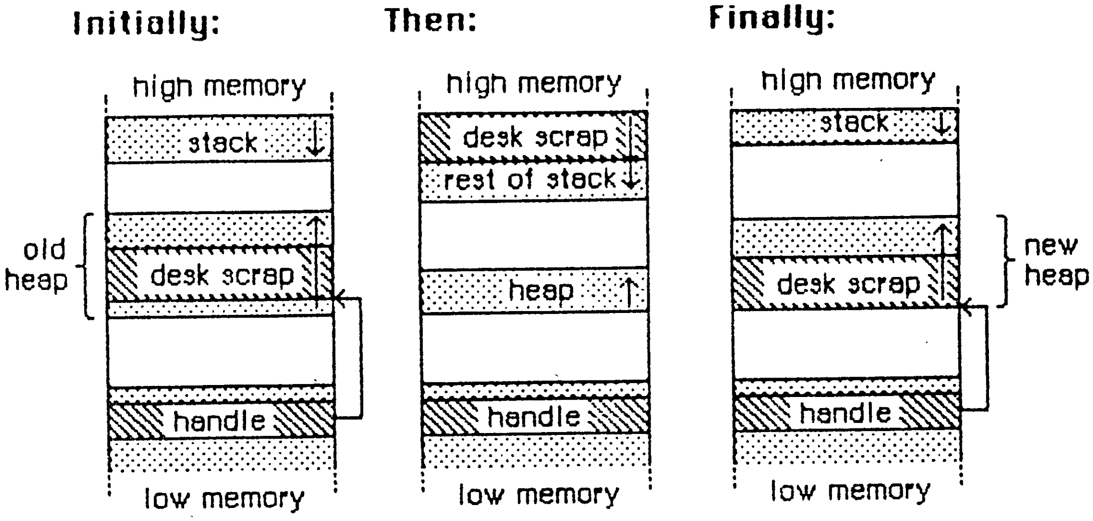


Figure 1. The Desk Scrap at Application Startup

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 could be set up so that a small initial segment of the application is loaded in just to check the scrap size. After a decision is made about whether to keep the scrap in memory or on the disk, the remaining segments of the application can be loaded in as needed.

There are certain disadvantages to keeping the desk scrap on the disk. The disk may be locked, it may not have enough room for the scrap, or it 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 operation 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); if it has a private scrap, this requires copying the desk scrap to the private scrap. Clearly the application must keep the contents of the desk scrap intact until the first Cut or Copy command is given. Thereafter, if it has a private scrap, 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 given in a desk accessory or in the next application.

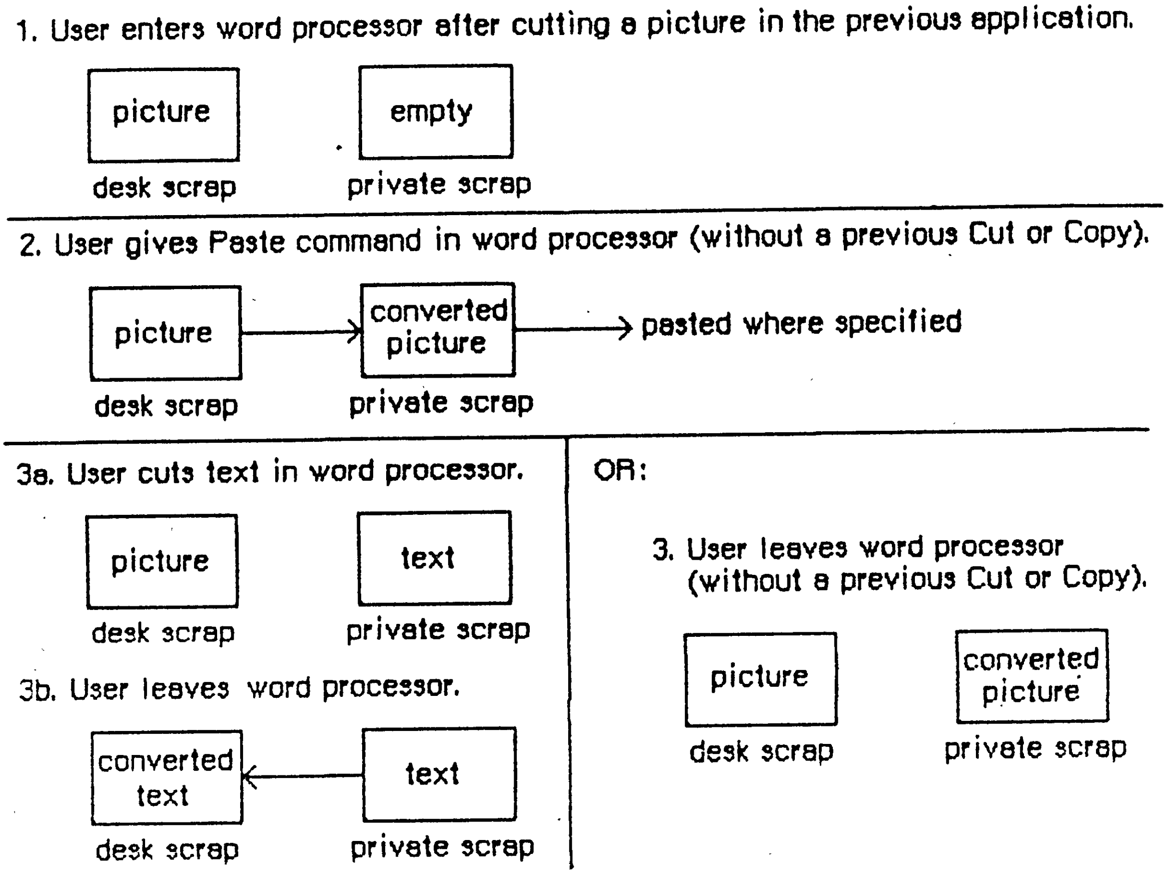

1. User enters word processor after cutting a picture in the previous application.
picture
desk scrap
empty
private scrap
2. User gives Paste command in word processor (without a previous Cut or Copy).
picture
desk scrap
converted picture
private scrap
pasted where specified
3a. User cuts text in word processor.
picture
desk scrap
text
private scrap
3b. User leaves word processor.
converted text
desk scrap
text
private scrap
OR:
3. User leaves word processor (without a previous Cut or Copy).
picture
desk scrap
converted picture
private scrap

Figure 2. Interaction between Scraps

Figure 2 illustrates how the interaction between the desk scrap and an application's private scrap might occur when the user gives a Paste command in a word processor after cutting a picture from a graphics application. As the picture that was cut gets copied from the desk scrap to the private scrap, it's converted to the format of the private scrap. If the user cuts or copies text in the word processor, it goes into the private scrap; then when the user then leaves the word processor, the text is copied from the private scrap into 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 startup (step 1), so that showing the Clipboard contents will always mean displaying the private scrap.

A process similar to the one shown in Figure 2 must be followed when the user reenters an application after using a desk accessory, since the user may have cut or copied something from the desk accessory. The application can check whether any such cutting or copying was done by looking at a count returned by InfoScrap. If this count changes during the use of the desk accessory, it means the contents of the desk scrap have changed. In this case, the application must 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 the application may store more than one version of the information in the 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, like resource types, are a sequence of four characters. 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
- '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 write at least its preferred standard type of data to the desk scrap, and may 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, the receiving application may or may not be able to convert the data to the type it needs. If not, some information may be lost in the transfer process. For example, a graphics application can easily convert text to a picture, but the reverse isn't true. Figure 3 illustrates the latter case: A picture consisting of text is cut from a graphics application, then pasted into a word processor document.

- 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 as a sequence of characters.

- On the other hand, if the graphics application takes the trouble to recognize which characters have been drawn in the picture, and writes them out to the desk scrap both as a picture and as text--like application B in Figure 3--the word processor will be able to treat them as editable text. In this case, however, any part of the picture that isn't text will be lost.

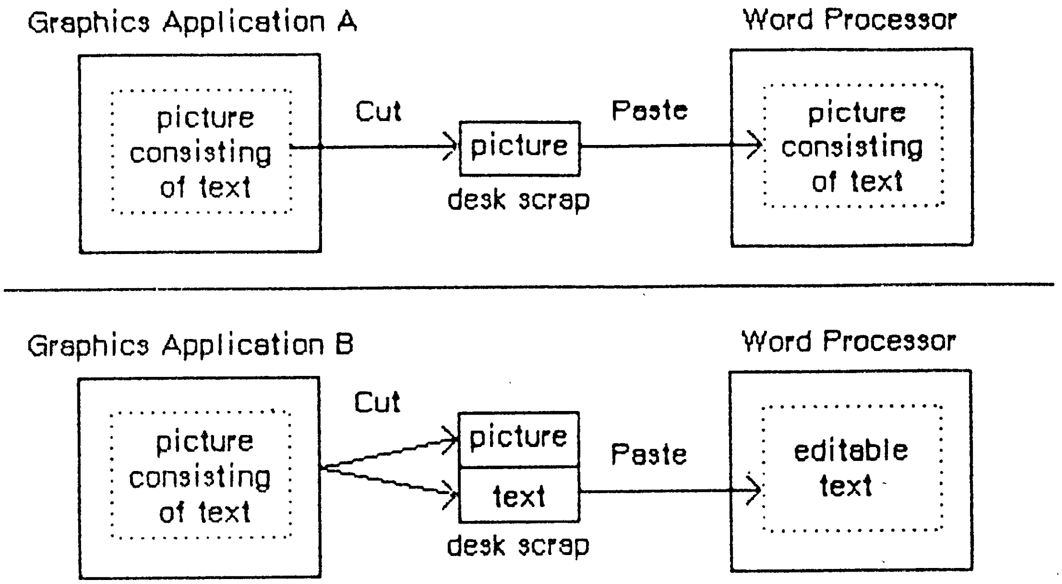


Figure 3. Inter-Application Cutting and Pasting

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, but still must write at least one of the standard types and be able to read both standard types.

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 has a private type of data as its preferred type, but also can write text and pictures, it should write the data in that order.

Since the size of the desk scrap is limited, it may be too costly to write out both an application-specific data type and one (or both) of the standard types. Instead of creating your own type, if your data is graphic, you may be able to use the standard picture type and encode additional information in picture comments. (As described in the QuickDraw manual, picture comments may be stored in the definition of a picture with the QuickDraw procedure PicComment; they're passed by the DrawPicture procedure to a special routine set up by the application for that purpose.) Applications that are to process that information can do so, while others can ignore it.

USING THE SCRAP MANAGER

Your application should call the InfoScrap function each time through its main event loop. You can use this function to find out whether there will be enough room in the heap for both the application itself and the desk scrap. If there won't be enough room for the scrap, 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 immediately when it starts up, or when the Paste command is given. Copying the desk scrap at startup is better if your application supports display of the Clipboard. The Scrap Manager routine that gets data from the desk scrap is called GetScrap.

When the user gives a command that terminates the application, call LoadScrap to read the desk scrap back into memory if it's on the disk (in case the user ejects the disk). If the application has a private scrap and any Cut or Copy commands were given within the application, the private scrap must be copied to the desk scrap.

To write data to the desk scrap, first call ZeroScrap to initialize it or clear its previous contents, and then PutScrap to put the data into it.

(note)

> GetScrap, PutScrap, and ZeroScrap all keep track of whether the scrap is in memory or on the disk, so you don't have to worry about it.

The same scrap interaction that happens at application startup should happen when the user returns to the application from a desk accessory. Similarly, the same interaction that happens when the application terminates should happen when the user accesses a desk accessory from the application.

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

Most of these routines return a result code indicating whether an error occurred. If no error occurred, they return the result code

```
       CONST noErr = Ø;    {no error}
```

If an error occurred at the Operating System level, an Operating System result code is returned; otherwise, a Scrap Manager result code is returned, as indicated in the routine descriptions. (See the Operating System Utilities manual for a list of all result codes.)

Getting Desk 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;  {size of desk scrap}
                     scrapHandle: Handle;   {handle to desk scrap}
                     scrapCount:  INTEGER;  {count changed by ZeroScrap}
                     scrapState:  INTEGER;  {tells where desk scrap is}
                     scrapName:   StringPtr {scrap file name}
                   END;
```

ScrapSize is the size of the 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. ScrapState is positive if the desk scrap is in memory, Ø if it's on the disk, or negative if it hasn't been initialized by ZeroScrap.

(note)

> ScrapState is actually Ø if the scrap **should** be on the disk; for instance, if the user deletes the Clipboard file and then cuts something, the scrap is really in memory, but ScrapState will be Ø.

ScrapName is a pointer to the name of the scrap file, usually "Clipboard File".

(note)

> InfoScrap assumes that the scrap file has a version number of Ø and is on the default volume. (Version numbers and volumes are described in the File Manager manual.)

Assembly-language note: The scrap information is available in global variables that have the same names as the Pascal fields.

Keeping the Desk Scrap on the Disk

```
FUNCTION UnloadScrap : LONGINT;
```

Assembly-language note: The macro you invoke to call UnloadScrap from assembly language is named _UnlodeScrap.

UnloadScrap writes the desk scrap from memory to the scrap file, and releases the memory it occupied. If the desk scrap is already on the disk, UnloadScrap does nothing. If no error occurs, UnloadScrap returns the result code noErr; otherwise, it returns an Operating System result code indicating an error.

```
FUNCTION LoadScrap : LONGINT;
```

Assembly-language note: The macro you invoke to call LoadScrap from assembly language is named _LodeScrap.

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 the result code noErr; otherwise, it returns an Operating System result code indicating an error.

Reading from the Desk Scrap

```
FUNCTION GetScrap (hDest: Handle; theType: ResType; VAR offset:
            LONGINT) : LONGINT;
```

Given an existing handle in hDest, 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 hDest to be a handle to the copy. Usually you'll pass in hDest a handle to a minimum-size block; GetScrap will resize the block and copy the scrap into 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.

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 result code (which will be negative) or the following Scrap Manager result code:

```
       CONST noTypeErr = -1Ø2;  {no data of the requested type}
```

For example, given the declarations

```
       VAR pHndl: Handle;   {handle for 'PICT' type}
           tHndl: Handle;   {handle for 'TEXT' type}
           length: LONGINT;
           offset: LONGINT;
```

you can make the following calls:

```
       pHndl := NewHandle(Ø);
       length := GetScrap(pHndl,'PICT',offset);
       IF length < Ø
         THEN
           {error-handling}
         ELSE DrawPicture(PicHandle(pHndl))
```

If your application wants data in the form of a picture, and the scrap contains only text, you can convert the text into a picture by doing the following:

```
tHndl := NewHandle(Ø);
length := GetScrap(tHndl,'TEXT',offset);
IF length < Ø
  THEN
    {error-handling}
  ELSE
    BEGIN
    HLock(tHndl);
    pHndl := OpenPicture(thePort^.portRect);
    TextBox(tHndl^,length,thePort^.portRect,teJustLeft);
    ClosePicture;
    HUnlock(tHndl);
    END
```

The Memory Manager procedures HLock and HUnlock are used to lock and unlock blocks when handles are dereferenced (see the Memory Manager Manual).

(note)
To copy the desk scrap to the TextEdit scrap, use the TextEdit function TEFromScrap.

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 each of the types it can read, and use the type that returns the lowest offset. (A lower offset means that this data type was written before the others, and therefore was preferred by the application that wrote it.)

(note)
If you're trying to read in a complicated picture, and there isn't enough room in memory for a copy of it, you can customize QuickDraw's picture retrieval so that DrawPicture will read 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.)

(note)
When reading in a picture from the scrap, allow a buffer of about 3.5K bytes. (There's a convention that the application defining the picture won't call the QuickDraw procedure CopyBits for more than 3K, so a 3.5K buffer should be large enough for any picture.)

Writing to the Desk Scrap

```
FUNCTION ZeroScrap : LONGINT;
```

If the scrap already exists (in memory or on the disk), ZeroScrap clears its contents; if not, the scrap is initialized in memory. You must call ZeroScrap before the first time you call PutScrap. If no error occurs, ZeroScrap returns the result code noErr; otherwise, it returns an Operating System result code indicating an error.

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.

(warning)

> Just check to see whether the scrapCount field has changed; don't rely on exactly how it has changed.

```
FUNCTION PutScrap (length: LONGINT; theType: ResType; source: Ptr) :
            LONGINT;
```

PutScrap writes the data pointed to by the source parameter to the desk scrap (in memory or on the disk). The length parameter indicates the number of bytes to write, and theType is the data type.

(warning)

> The specified type must be different from the type of any data already in the desk scrap. If you write data of a type already in the scrap, the new data will be appended to the scrap, and subsequent GetScrap calls will still return the old data.

If no error occurs, PutScrap returns the result code noErr; otherwise, it returns an Operating System result code indicating an error, or the following Scrap Manager result code:

```
CONST noScrapErr = -1ØØ;  {desk scrap isn't initialized}
```

(note)

> To copy the TextEdit scrap to the desk scrap, use the TextEdit function TEToScrap.

(warning)
Don't forget to call ZeroScrap to initialize the scrap or clear its previous contents.

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, add 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

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 { Scrap Manager result codes }

      noScrapErr = -1ØØ;  {desk scrap isn't initialized}
      noTypeErr  = -1Ø2;  {no data of the requested type}
```

Data Types

```
TYPE PScrapStuff = ^ScrapStuff;
     ScrapStuff  = RECORD
                     scrapSize:   LONGINT;  {size of desk scrap}
                     scrapHandle: Handle;   {handle to desk scrap}
                     scrapCount:  INTEGER;  {count changed by ZeroScrap}
                     scrapState:  INTEGER;  {tells where desk scrap is}
                     scrapName:   StringPtr {scrap file name}
                   END;
```

Routines

Getting Desk Scrap Information

```
FUNCTION InfoScrap : PScrapStuff;
```

Keeping the Desk Scrap on the Disk

```
FUNCTION UnloadScrap : LONGINT;
FUNCTION LoadScrap :   LONGINT;
```

Reading from the Desk Scrap

```
FUNCTION GetScrap (hDest: Handle; theType: ResType; VAR offset: LONGINT)
                   : LONGINT;
```

Writing to the Desk Scrap

```
FUNCTION ZeroScrap : LONGINT;
FUNCTION PutScrap   (length: LONGINT; theType: ResType; source: Ptr) :
                     LONGINT;
```

Assembly-Language Information

Constants

```
; Scrap Manager result codes

noScrapErr     .EQU    -100      ;desk scrap isn't initialized
noTypeErr      .EQU    -102      ;no data of the requested type
```

Special Macro Names

Pascal name	Macro name
LoadScrap	_LodeScrap
UnloadScrap	_UnlodeScrap

Variables

ScrapSize	Size in bytes of desk scrap (long)
ScrapHandle	Handle to desk scrap in memory
ScrapCount	Count changed by ZeroScrap (word)
ScrapState	Tells where desk scrap is (word)
ScrapName	Pointer to scrap file name (preceded by length byte)

GLOSSARY

desk scrap: The place where data is stored when it's cut (or copied) and pasted among applications and desk accessories.

scrap: A place where cut or copied data is stored.

scrap file: The file containing the desk scrap (usually named "Clipboard File").

MACINTOSH USER EDUCATION

The Toolbox Utilities: A Programmer's Guide

/TOOLUTIL/UTIL

See Also: Macintosh User Interface Guidelines
Inside Macintosh: A Road Map
Macintosh Memory Management: An Introduction
QuickDraw: A Programmer's Guide
The Resource Manager: A Programmer's Guide
Programming Macintosh Applications in Assembly Language

Modification History:			
	First Draft	Caroline Rose	5/16/83
	Second Draft (ROM 7)	Caroline Rose	1/4/84
	Erratum Added	Caroline Rose	2/8/84
	Third Draft	Caroline Rose & Katie Withey	11/13/84

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.

Summary of significant changes and additions since last draft:

- Routines added: GetIndString (page 4); PackBits and UnpackBits (pages 6-7); GetIndPattern (pages 1Ø-11); DeltaPoint, SlopeFromAngle, and AngleFromSlope (pages 12-13).

- The section on fixed-point numbers and the StringPtr and StringHandle data types have been moved to <u>Macintosh Memory Management</u>: <u>An Introduction</u>.

- The Munger function returns the offset of the first byte past where a replacement or insertion occurred (pages 5-6).

- The resource IDs for the standard Macintosh pattern list (page 1Ø) and cursors (page 11) are included. The formats of these and other miscellaneous resources are given (page 14).

- The description of the ShieldCursor procedure has been changed (pages 11-12).

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 the comprehensive Inside Macintosh manual. ***

Like all Toolbox documentation, this manual assumes you're familiar with Lisa Pascal and the information in the following manuals:

- Inside Macintosh: A Road Map
- Macintosh User Interface Guidelines
- Macintosh Memory Management: An Introduction
- Programming Macintosh Applications in Assembly Language, if you're using assembly language

Depending on which Toolbox Utilities you're interested in using, you may also need to be familiar with:

- resources, as described in the Resource Manager manual
- the basic concepts and structures behind QuickDraw

TOOLBOX UTILITY ROUTINES

Fixed-Point Arithmetic

Fixed-point numbers are described in Macintosh Memory Management: An Introduction. *** The next draft of that manual will be corrected to show that bit 15 of the high-order word is the sign bit. *** Note that fixed-point values can be added and subtracted as long integers.

In addition to the following routines, the HiWord and LoWord functions (described under "Other Operations on Long Integers" below) are useful when you're working with fixed-point numbers.

FUNCTION FixRatio (numer,denom: INTEGER) : Fixed;

FixRatio returns the fixed-point quotient of numer and denom. Numer or denom may be any signed integer. The result is truncated. If denom is Ø, FixRatio returns $7FFFFFFF with the sign of numer.

FUNCTION FixMul (a,b: Fixed) : Fixed;

FixMul returns the fixed-point product of a and b. The result is computed MOD 65536, and truncated.

FUNCTION FixRound (x: Fixed) : INTEGER;

Given a positive fixed-point number, FixRound rounds it to the nearest integer and returns the result. If the value is halfway between two integers (.5), it's rounded up. To round a negative fixed-point number, multiply by -1, round, then multiply by -1 again.

String Manipulation

FUNCTION NewString (theString: Str255) : StringHandle;

NewString allocates the specified string as a relocatable object on the heap and returns a handle to it.

PROCEDURE SetString (h: StringHandle; theString: Str255);

SetString sets the string whose handle is passed in h to the string specified by theString.

FUNCTION GetString (stringID: INTEGER) : StringHandle;

GetString returns a handle 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). If the resource can't be read, GetString returns NIL.

(note)

> If your application uses a large number of strings, storing them in a string list in the resource file will be more efficient. You can access strings in a string list with GetIndString, as described below.

PROCEDURE GetIndString (VAR theString: Str255; strListID: INTEGER;
index: INTEGER); [No trap macro]

GetIndString returns in theString a string in the string list that has the resource ID strListID. It reads the string list from the resource file if necessary, by calling the Resource Manager function GetResource('STR#',strListID). It returns the string specified by the index parameter, which can range from 1 to the number of strings in the list. If the resource can't be read or the index is out of range, the empty string is returned.

Byte Manipulation

```
FUNCTION Munger (h: Handle; offset: LONGINT; ptrl: Ptr; lenl: LONGINT;
            ptr2: Ptr; len2: LONGINT) : LONGINT;
```

Munger (which rhymes with "plunger") lets you manipulate bytes in the string of bytes (the "destination string") to which h is a handle. The operation starts at the specified byte offset in the destination string.

(note)

> Although the term "string" is used here, Munger does **not** assume it's manipulating a Pascal string; if you pass it a handle to a Pascal string, you must take into account the length byte.

The exact nature of the operation done by Munger depends on the values you pass it in two pointer/length parameter pairs. In general, (ptrl,lenl) defines a target string to be replaced by the second string (ptr2,len2). If these four parameters are all positive and nonzero, Munger looks for the target string in the destination string, starting from the given offset and ending at the end of the string; it replaces the first occurrence it finds with the replacement string and returns the offset of the first byte past where the replacement occurred. Figure 1 illustrates this; the bytes represent ASCII characters as shown.

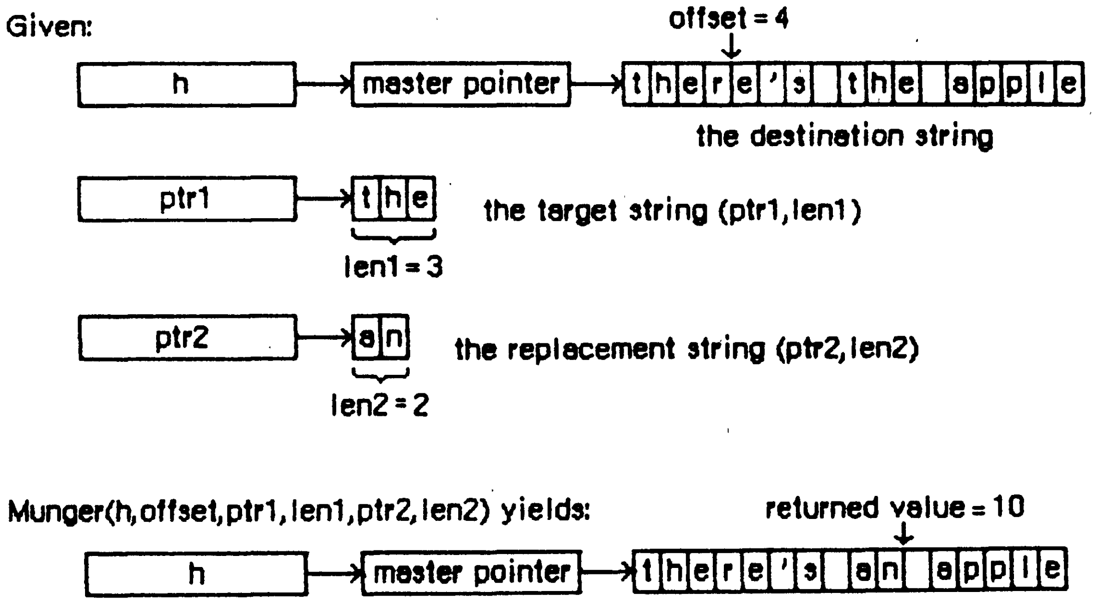


Figure 1. Munger Function

Different operations occur if either pointer is NIL or either length is Ø:

- If ptrl is NIL, the substring of length lenl starting at the given offset is replaced by the replacement string. If lenl is negative, the substring from the given offset to the end of the destination string is replaced by the replacement string. In either case, Munger returns the offset of the first byte past where the replacement occurred.

- If lenl is Ø, (ptr2,len2) is simply inserted at the given offset; no text is replaced. Munger returns the offset of the first byte past where the insertion occurred.

- If ptr2 is NIL, Munger returns the offset at which the target string was found. The destination string isn't changed.

- If len2 is Ø (and ptr2 is **not** NIL), the target string is deleted rather than replaced (since the replacement string is empty). Munger returns the offset at which the deletion occurred.

If it can't find the target string in the destination string, Munger returns a negative value.

There's one case in which Munger performs a replacement even if it doesn't find all of the target string. If the substring from the offset to the end of the destination string matches the beginning of the target string, the portion found is replaced with the replacement string.

(warning)

> Be careful not to specify an offset that's greater than the length of the destination string, or unpredictable results may occur.

(note)

> The destination string must be in a relocatable block that was allocated by the Memory Manager. Munger accesses the string's length by calling the Memory Manager routines GetHandleSize and SetHandleSize.

PROCEDURE PackBits (VAR srcPtr,dstPtr: Ptr; srcBytes: INTEGER);

PackBits compresses srcBytes bytes of data starting at srcPtr and stores the compressed data at dstPtr. The value of srcBytes should not be greater than 127. Bytes are compressed when there are three or more consecutive equal bytes. After the data is compressed, srcPtr is incremented by srcBytes and dstPtr is incremented by the number of bytes that the data was compressed to. In the worst case, the compressed data can be one byte longer than the original data.

PackBits is usually used to compress QuickDraw bit images; in this case, you should call it for one row at a time. (Because of the repeating patterns in QuickDraw images, there are more likely to be consecutive equal bytes there than in other data.) Use UnpackBits (below) to expand data compressed by PackBits.

PROCEDURE UnpackBits (VAR srcPtr,dstPtr: Ptr; dstBytes: INTEGER);

Given in srcPtr a pointer to data that was compressed by PackBits, UnpackBits expands the data and stores the result at dstPtr. DstBytes is the length that the expanded data will be; it should be the value that was passed to PackBits in the srcBytes parameter. After the data is expanded, srcPtr is incremented by the number of bytes that were expanded and dstPtr is incremented by dstBytes.

Bit Manipulation

Given a pointer and an offset, these routines can manipulate any specific bit. The pointer can point to an even or odd byte; the offset can be any positive long integer, starting at Ø for the high-order bit of the specified byte (see Figure 2).

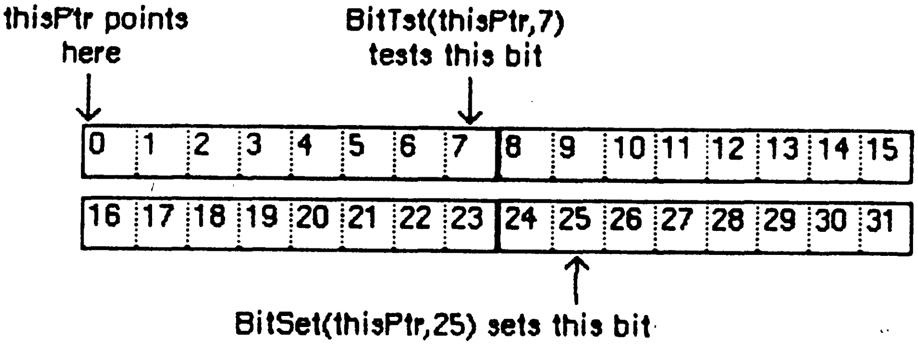


Figure 2. Bit Numbering for Utility Routines

(note)
This bit numbering is the opposite of the MC68ØØØ bit numbering to allow for greater generality. For example, you can directly access bit 1ØØØ of a bit image given a pointer to the beginning of the bit image.

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 (value1,value2: LONGINT) : LONGINT;

BitAnd returns the result of the AND logical operation on the bits comprising the given long integers (value1 AND value2).

FUNCTION BitOr (value1,value2: LONGINT) : LONGINT;

BitOr returns the result of the OR logical operation on the bits comprising given long integers (value1 OR value2).

FUNCTION BitXor (value1,value2: LONGINT) : LONGINT;

BitXor returns the result of the XOR logical operation on the bits comprising the given long integers (value1 XOR value2).

FUNCTION BitNot (value: LONGINT) : LONGINT;

BitNot returns the result of the NOT logical operation on the bits comprising the given long integer (NOT value).

FUNCTION BitShift (value: 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 MOD 32. 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.

(note)

If you're dealing with fixed-point numbers, you can define a variant record instead of using HiWord and LoWord. For example:

```
TYPE FixedAndInt =
       RECORD CASE INTEGER OF
         1: (fixedView: Fixed);
         2: (intView: RECORD
                         whole: INTEGER;
                         part:  INTEGER
                       END;
       END;
```

If you declare x to be of type FixedAndInt, you can access it as a fixed-point value with x.fixedView, or access the integer part with x.intView.whole and the fractional part with x.intView.part.

```
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). If the resource can't be read, GetIcon returns NIL.

PROCEDURE PlotIcon (theRect: Rect; theIcon: Handle);

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;

GetPattern 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). If the resource can't be read, GetPattern returns NIL. The PatHandle data type is defined in the Toolbox Utilities as follows:

```
TYPE PatPtr    = ^Pattern;
     PatHandle = ^PatPtr;
```

PROCEDURE GetIndPattern (VAR thePattern: Pattern; patListID: INTEGER;
          index: INTEGER); [No trap macro]

GetIndPattern returns in thePattern a pattern in the pattern list that has the resource ID patListID. It reads the pattern list from the resource file if necessary, by calling the Resource Manager function GetResource('PAT#',patListID). It returns the pattern specified by the index parameter, which can range from 1 to the number of patterns in the pattern list.

There's a pattern list in the system resource file that contains the standard Macintosh patterns used by MacPaint (see Figure 3). Its resource ID is:

```
CONST sysPatListID = Ø;
```

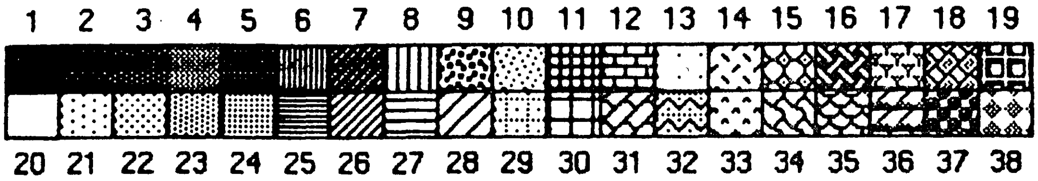


Figure 3. Standard Patterns

FUNCTION GetCursor (cursorID: INTEGER) : CursHandle;

GetCursor 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). If the resource can't be read, GetCursor returns NIL. The CursHandle data type is defined in the Toolbox Utilities as follows:

```
TYPE CursPtr    = ^Cursor;
     CursHandle = ^CursPtr;
```

The standard cursors shown in Figure 4 are defined in the system resource file. Their resource IDs are as follows:

```
CONST iBeamCursor = 1;  {to select text}
      crossCursor = 2;  {to draw graphics}
      plusCursor  = 3;  {to select cells in structured documents}
      watchCursor = 4;  {to indicate a long wait}
```

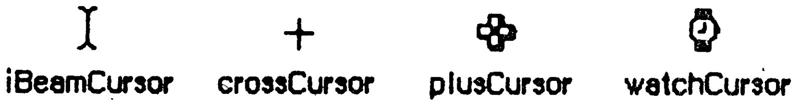


Figure 4. Standard Cursors

(note)
You can set the cursor with the QuickDraw procedure SetCursor. The arrow cursor is defined in QuickDraw as a global variable named arrow.

PROCEDURE ShieldCursor (shieldRect: Rect; offsetPt: Point);

ShieldCursor removes the cursor from the screen if the cursor and a given rectangle intersect. Like the QuickDraw procedure HideCursor, it decrements the cursor level and must be balanced by a call to ShowCursor. The rectangle may be given in global or local coordinates:

- If they're global coordinates, pass (Ø,Ø) in offsetPt.

- If they're a grafPort's local coordinates, pass the top left corner of the grafPort's boundary rectangle in offsetPt. (Like LocalToGlobal in QuickDraw, ShieldCursor will offset the coordinates of the rectangle by the coordinates of this point.)

FUNCTION GetPicture (picID: 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',picID). If the resource can't be read, GetPicture returns NIL. The PicHandle data type is defined in QuickDraw.

Miscellaneous Utilities

FUNCTION DeltaPoint (ptA,ptB: Point) : LONGINT;

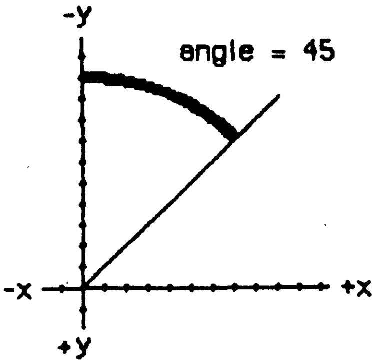
DeltaPoint subtracts the coordinates of ptA from the coordinates of ptB and returns the result as a long integer: the high-order word of the result is the vertical coordinate and the low-order word is the horizontal coordinate.

(note)

> The QuickDraw procedure SubPt does the same calculation but returns the result in a VAR parameter of type Point.

FUNCTION SlopeFromAngle (angle: INTEGER) : Fixed;

Given an angle, SlopeFromAngle returns the slope dh/dv of the line forming that angle with the y-axis (dh/dv is the horizontal change divided by the vertical change between any two points on the line). Figure 5 illustrates SlopeFromAngle (and AngleFromSlope, described below, which does the reverse). The angle is treated MOD 18Ø, and is in degrees measured from 12 o'clock; positive degrees are measured clockwise, negative degrees are measured counterclockwise (for example, 9Ø degrees is at 3 o'clock and -9Ø degrees is at 9 o'clock). Positive y is down; positive x is to the right.



SlopeFromAngle(45) = $FFFF0000
AngleFromSlope($FFFF0000) = 45

($FFFF0000 is -1.0)

Figure 5. SlopeFromAngle and AngleFromSlope

FUNCTION AngleFromSlope (slope: Fixed) : INTEGER;

Given the slope dh/dv of a line (see SlopeFromAngle above), AngleFromSlope returns the angle formed by that line and the y-axis. The angle is treated MOD 18Ø, and is measured in degrees the same way as for SlopeFromAngle.

AngleFromSlope is meant for use when speed is much more important than accuracy--its integer result is guaranteed to be within one degree of the correct answer, but not necessarily within half a degree. However, the equation

AngleFromSlope (SlopeFromAngle (x)) = x

is always true (although the reverse is not).

FORMATS OF MISCELLANEOUS RESOURCES

The following table shows the exact format of various resources. The lengths in the last column are given in bytes. For more information about the contents of the graphics-related resources, see the QuickDraw manual.

Resource	Resource type	Number of bytes	Contents
Icon	'ICON'	128 bytes	The icon
Icon list	'ICN#'	n * 128 bytes	n icons
Pattern	'PAT '	8 bytes	The pattern
Pattern list	'PAT#'	2 bytes	Number of patterns
		n * 8 bytes	n patterns
Cursor	'CURS'	32 bytes	The data
		32 bytes	The mask
		4 bytes	The hotSpot
Picture	'PICT'	2 bytes	Picture length (m+1Ø)
		8 bytes	Picture frame
		m bytes	Picture definition data
String	'STR '	m bytes	The string (1-byte length followed by the characters)
String list	'STR#'	2 bytes	Number of strings
		m bytes	The strings

(note)

Unlike a pattern list or a string list, an icon list doesn't start with the number of items in the list.

SUMMARY OF THE TOOLBOX UTILITIES

Constants

```
CONST { Resource ID of standard pattern list }

      sysPatListID = Ø;

      { Resource IDs of standard cursors }

      iBeamCursor = 1;  {to select text}
      crossCursor = 2;  {to draw graphics}
      plusCursor  = 3;  {to select cells in structured documents}
      watchCursor = 4;  {to indicate a long wait}
```

Data Types

```
TYPE Int64Bit = RECORD
                  hiLong: LONGINT;
                  loLong: LONGINT
                END;

     CursPtr    = ^Cursor;
     CursHandle = ^CursPtr;

     PatPtr    = ^Pattern;
     PatHandle = ^PatPtr;
```

Routines

Fixed-Point Arithmetic

```
FUNCTION FixRatio (numer,denom: INTEGER) : Fixed;
FUNCTION FixMul   (a,b: Fixed) : Fixed;
FUNCTION FixRound (x: Fixed) : INTEGER;
```

String Manipulation

```
FUNCTION  NewString    (theString: Str255) : StringHandle;
PROCEDURE SetString    (h: StringHandle; theString: Str255);
FUNCTION  GetString    (stringID: INTEGER) : StringHandle;
PROCEDURE GetIndString (VAR theString: Str255; strListID: INTEGER;
                         index: INTEGER);  [No trap macro]
```

Byte Manipulation

```
FUNCTION  Munger     (h: Handle; offset: LONGINT; ptrl: Ptr; lenl:
                      LONGINT; ptr2: Ptr; len2: LONGINT) : LONGINT;
PROCEDURE PackBits   (VAR srcPtr,dstPtr: Ptr; srcBytes: INTEGER);
PROCEDURE UnpackBits (VAR srcPtr,dstPtr: Ptr; dstBytes: INTEGER);
```

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   (valuel,value2: LONGINT) : LONGINT;
FUNCTION BitOr    (valuel,value2: LONGINT) : LONGINT;
FUNCTION BitXor   (valuel,value2: LONGINT) : LONGINT;
FUNCTION BitNot   (value: LONGINT) : LONGINT;
FUNCTION BitShift (value: 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;
PROCEDURE GetIndPattern (VAR thePattern: Pattern; patListID: INTEGER;
                         index: INTEGER);  [No trap macro]
FUNCTION  GetCursor     (cursorID: INTEGER) : CursHandle;
PROCEDURE ShieldCursor  (shieldRect: Rect; offsetPt: Point);
FUNCTION  GetPicture    (picID: INTEGER) : PicHandle;
```

Miscellaneous Utilities

```
FUNCTION  DeltaPoint     (ptA,ptB: Point) : LONGINT;
FUNCTION  SlopeFromAngle (angle: INTEGER) : Fixed;
FUNCTION  AngleFromSlope (slope: Fixed) : INTEGER;
```

Assembly-Language Information

Constants

```
; Resource ID of standard pattern list

sysPatListID    .EQU    Ø

; Resource IDs of standard cursors

iBeamCursor     .EQU    1       ;to select text
crossCursor     .EQU    2       ;to draw graphics
plusCursor      .EQU    3       ;to select calls in structured documents
watchCursor     .EQU    4       ;to indicate a long wait
```

MACINTOSH USER EDUCATION

Macintosh Packages: A Programmer's Guide

/PACKAGES/PACK

See Also: The Macintosh User Interface Guidelines
The Memory Manager: A Programmer's Guide
QuickDraw: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Window Manager: A Programmer's Guide
Macintosh Control Manager Programmer's Guide
The Event Manager: A Programmer's Guide
The Dialog Manager: A Programmer's Guide
TextEdit: A Programmer's Guide
Programming Macintosh Applications in Assembly Language
The Structure of a Macintosh Application

Modification History:	First Draft (ROM 7)	B. Hacker & C. Rose	2/29/84
	Second Draft	Caroline Rose	5/7/84

ABSTRACT

Packages are sets of data structures and routines that are stored as resources and brought into memory only when needed. There's a package for presenting the standard user interface when a file is to be saved or opened, and others for doing more specialized operations such as floating-point arithmetic. This manual describes packages and the Package Manager, the part of the Macintosh User Interface Toolbox that provides access to packages.

Summary of significant changes and additions since last draft:

- The documentation of the International Utilities Package and the Binary-Decimal Conversion Package has been added.

- There's a new feature in the Standard File Package routine SFGetFile, whereby the user can select a file name by pressing a key.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes packages and the Package Manager. The Macintosh packages include one for presenting the standard user interface when a file is to be saved or opened, and others for doing more specialzed operations such as floating-point arithmetic. The Package Manager is the part of the Macintosh User Interface Toolbox that provides access to packages. *** Eventually, this will become part of the comprehensive Inside Macintosh manual. ***

You should already be familiar with the Macintosh User Interface Guidelines, Lisa Pascal, the Macintosh Operating System's Memory Manager, and the Resource Manager. Using the various packages may require that you be familiar with other parts of the Toolbox and Operating System as well.

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 a discussion of the Package Manager and packages in general. This is followed by a series of sections on the individual packages. You'll only need to read the sections about the packages that interest you. Each section describes the package briefly, tells how its routines fit into the flow of your application program, and then gives detailed descriptions of the package's routines.

Finally, there are summaries of the Package Manager and the individual packages, for quick reference, followed by a glossary of terms used in this manual.

THE PACKAGE MANAGER

The Package Manager is the part of the Macintosh User Interface Toolbox that enables you to access packages. Packages are sets of data structures and routines that are stored as resources and brought into memory only when needed. They serve as extensions to the Macintosh Operating System and User Interface Toolbox, for the most part performing less common operations.

The Macintosh packages, which are stored in the system resource file, include the following:

- The Standard File Package, for presenting the standard user interface when a file is to be saved or opened.

- The Disk Initialization Package, for initializing and naming new disks. This package is called by the Standard File Package; you'll only need to call it in nonstandard situations.

- The International Utilities Package, for accessing country-dependent information such as the formats for numbers, currency, dates, and times.

- The Binary-Decimal Conversion Package, for converting integers to decimal strings and vice versa.

- The Floating-Point Arithmetic and Transcendental Functions Packages. *** These packages, which occupy a total of about 8.5K bytes, will be documented in a future draft of this manual. ***

Packages have the resource type 'PACK' and the following resource IDs:

```
CONST dskInit = 2;  {Disk Initialization}
      stdFile = 3;  {Standard File}
      flPoint = 4;  {Floating-Point Arithmetic}
      trFunc  = 5;  {Transcendental Functions}
      intUtil = 6;  {International Utilities}
      bdConv  = 7;  {Binary-Decimal Conversion}
```

Assembly-language note: All macros for calling the routines in a particular package expand to invoke one macro, _PackN, where N is the resource ID of the package. The package determines which routine to execute from the routine selector, an integer that's passed to it on the stack. For example, the routine selector for the Standard File Package procedure SFPutFile is 1, so invoking the macro _SFPutFile pushes 1 onto the stack and invokes _Pack3.

There are two Package Manager routines that you can call directly from Pascal: one that lets you access a specified package and one that lets you access all packages. The latter will already have been called when your application starts up, so normally you won't ever have to call the Package Manager yourself. Its procedures are described below for advanced programmers who may want to use them in unusual situations.

```
PROCEDURE InitPack (packID: INTEGER);
```

InitPack enables you to use the package specified by packID, which is the package's resource ID. (It gets a handle that will be used later to read the package into memory.)

```
PROCEDURE InitAllPacks;
```

InitAllPacks enables you to use all Macintosh packages (as though InitPack were called for each one). It will already have been called when your application starts up.

THE INTERNATIONAL UTILITIES PACKAGE

The International Utilities Package contains routines and data types that enable you to make your Macintosh application country-independent. Routines are provided for formatting dates and times and comparing strings in a way that's appropriate to the country where your application is being used. There's also a routine for testing whether to use the metric system of measurement. These routines access country-dependent information (stored in a resource file) that also tells how to format numbers and currency; you can access this information yourself for your own routines that may require it.

*** In the Inside Macintosh manual, the documentation of this package will be at the end of the volume that describes the User Interface Toolbox. ***

You should already be familiar with the Resource Manager, the Package Manager, and packages in general.

International Resources

Country-dependent information is kept in the system resource file in two resources of type 'INTL', with the resource IDs Ø and 1:

- International resource Ø contains the format for numbers, currency, and time, a short date format, and an indication of whether to use the metric system.

- International resource 1 contains a longer format for dates (spelling out the month and possibly the day of the week, with or without abbreviation) and a routine for localizing string comparison.

The system resource file released in each country contains the standard international resources for that country. Figure I-1 illustrates the standard formats for the United States, Great Britain, Italy, Germany, and France.

	United States	Great Britain	Italy	Germany	France
Numbers	1,234.56	1,234.56	1.234,56	1.234,56	1 234.56
List separator	;	,	;	;	;
Currency	$0.23	£0.23	L. 0,23	0,23 DM	0,23 F
	($0.45)	(£0.45)	L. -0,45	-0,45 DM	-0,45 F
	$345.00	£345	L. 345	325,00 DM	325 F
Time	9:05 AM	09:05	9:05	9.05 Uhr	9:05
	11:30 AM	11:30	11:30	11.30 Uhr	11:30
	11:20 PM	23:20	23:20	23.20 Uhr	23:20
	11:20:09 PM	23:20:00	23:20:09	23.20.09 Uhr	23:20:09
Short date	12/22/84	22/12/1984	22-12-1984	22.12.1984	22.12.84
	2/ 1/84	01/02/1984	1-02-1984	1.02.1984	1.02.84

		Unabbreviated	Abbreviated
Long date	United States	Wednesday, February 1, 1984	Wed, Feb 1, 1984
	Great Britain	Wednesday, February 1, 1984	Wed, Feb 1, 1984
	Italy	mercoledì 1 Febbraio 1984	mer 1 Feb 1984
	Germany	Mittwoch, 1. Februar 1984	Mit, 1. Feb 1984
	France	Mercredi 1 fevrier 1984	Mer 1 fev 1984

Figure I-1. Standard International Formats

The routines in the International Utilities Package use the information in these resources; for example, the routines for formatting dates and times yield strings that look like those shown in Figure I-1. Routines in other packages, in desk accessories, and in ROM also access the international resources when necessary, as should your own routines if they need such information.

In some cases it may be appropriate to store either or both of the international resources in the application's or document's resource file, to override those in the system resource file. For example, suppose an application creates documents containing currency amounts and gets the currency format from international resource Ø. Documents created by such an application should have their own copy of the international resource Ø that was used to create them, so that the unit of currency will be the same if the document is displayed on a Macintosh configured for another country.

Information about the exact components and structure of each international resource follows here; you can skip this if you intend only to call the formatting routines in the International Utilities Package and won't access the resources directly yourself.

International Resource Ø

The International Utilities Package contains the following data types for accessing international resource Ø:

```
TYPE IntlØHndl = ^IntlØPtr;  *** Following "Int" is the letter "l" ***
     IntlØPtr  = ^IntlØRec;
     IntlØRec  = PACKED RECORD
                   decimalPt:    CHAR; {decimal point character}
                   thousSep:     CHAR; {thousands separator}
                   listSep:      CHAR; {list separator}
                   currSym1:     CHAR; {currency symbol}
                   currSym2:     CHAR;
                   currSym3:     CHAR;
                   currFmt:      Byte; {currency format}
                   dateOrder:    Byte; {order of short date elements}
                   shortDateFmt: Byte; {short date format}
                   dateSep:      CHAR; {date separator}
                   timeCycle:    Byte; {Ø if 24-hour cycle, 255 if 12-hour}
                   timeFmt:      Byte; {time format}
                   mornStr:      PACKED ARRAY[1..4] OF CHAR;
                                  {trailing string for first 12-hour cycle}
                   eveStr:       PACKED ARRAY[1..4] OF CHAR;
                                  {trailing string for last 12-hour cycle}
                   timeSep:      CHAR; {time separator}
                   time1Suff:    CHAR; {trailing string for 24-hour cycle}
                   time2Suff:    CHAR;
                   time3Suff:    CHAR;
                   time4Suff:    CHAR;
                   time5Suff:    CHAR;
                   time6Suff:    CHAR;
                   time7Suff:    CHAR;
                   time8Suff:    CHAR;
                   metricSys:    Byte;   {255 if metric, Ø if not}
                   intlØVers:    INTEGER {version information}
                 END;
```

(note)

> A NUL character (ASCII code Ø) in a field of type CHAR means there's no such character. The currency symbol and the trailing string for the 24-hour cycle are separated into individual CHAR fields because of Pascal packing conventions. All strings include any required spaces.

The decimalPt, thousSep, and listSep fields define the number format. The *thousands separator* is the character that separates every three digits to the left of the decimal point. The *list separator* is the character that separates numbers, as when a list of numbers is entered by the user; it must be different from the decimal point character. If it's the same as the thousands separator, the user must not include the latter in entered numbers.

CurrSym1 through currSym3 define the currency symbol (only one character for the United States and Great Britain, but two for France and three for Italy and Germany). CurrFmt determines the rest of the currency format, as shown in Figure I-2. The decimal point character and thousands separator for currency are the same as in the number format.

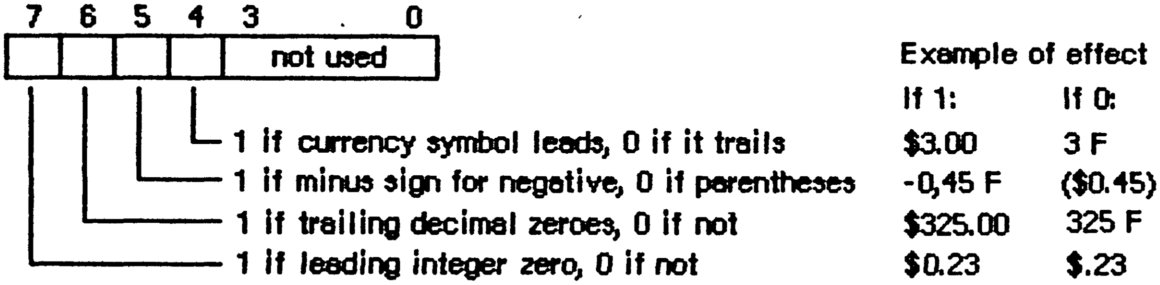

7 6 5 4 3 0
not used
Example of effect
If 1:
If 0:
1 if currency symbol leads, 0 if it trails
$3.00
3 F
1 if minus sign for negative, 0 if parentheses
-0,45 F
($0.45)
1 if trailing decimal zeroes, 0 if not
$325.00
325 F
1 if leading integer zero, 0 if not
$0.23
$.23

Figure I-2. CurrFmt Field

The following predefined constants are masks that can be used to set or test the bits in the currFmt field:

```
CONST currSymLead   = 16;  {set if currency symbol leads}
      currNegSym    = 32;  {set if minus sign for negative}
      currTrailingZ = 64;  {set if trailing decimal zeroes}
      currLeadingZ  = 128; {set if leading integer zero}
```

(note)
You can also apply the currency format's leading- and trailing-zero indicators to the number format if desired.

The dateOrder, shortDateFmt, and dateSep fields define the short date format. DateOrder indicates the order of the day, month, and year, with one of the following values:

```
CONST mdy = Ø;  {month day year}
      dmy = 1;  {day month year}
      ymd = 2;  {year month day}
```

ShortDateFmt determines whether to show leading zeroes in day and month numbers and whether to show the century, as illustrated in Figure I-3. DateSep is the character that separates the different parts of the date.

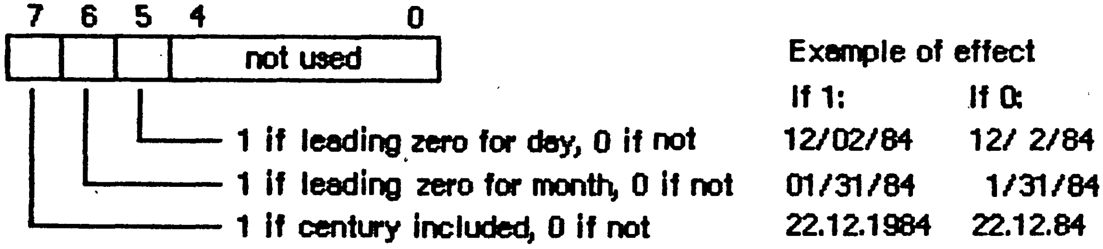

7 6 5 4 0
not used
Example of effect
If 1:
If 0:
1 if leading zero for day, 0 if not
12/02/84
12/ 2/84
1 if leading zero for month, 0 if not
01/31/84
1/31/84
1 if century included, 0 if not
22.12.1984
22.12.84

Figure I-3. ShortDateFmt Field

To set or test the bits in the shortDateFmt field, you can use the following predefined constants as masks:

```
CONST dayLeadingZ = 32;  {set if leading zero for day}
      mntLeadingZ = 64;  {set if leading zero for month}
      century     = 128; {set if century included}
```

The next several fields define the time format: the cycle (12 or 24 hours); whether to show leading zeroes (timeFmt, as shown in Figure I-4); a string to follow the time (two for 12-hour cycle, one for 24-hour); and the time separator character.

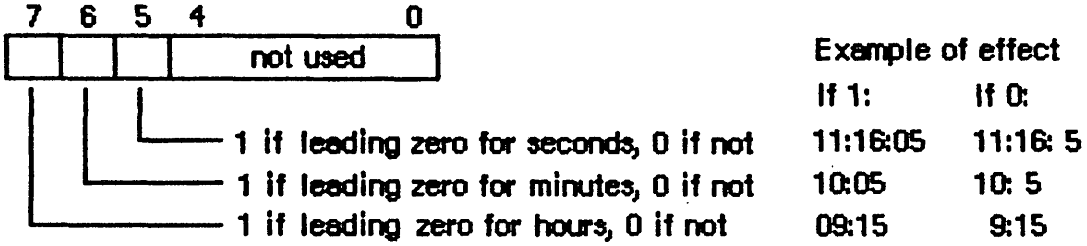

7
6
5
4
0
not used
Example of effect
If 1:
If 0:
1 if leading zero for seconds, 0 if not
11:16:05
11:16: 5
1 if leading zero for minutes, 0 if not
10:05
10: 5
1 if leading zero for hours, 0 if not
09:15
9:15

Figure I-4. TimeFmt Field

The following masks are available for setting or testing bits in the timeFmt field:

```
CONST secLeadingZ = 32;  {set if leading zero for seconds}
      minLeadingZ = 64;  {set if leading zero for minutes}
      hrLeadingZ  = 128; {set if leading zero for hours}
```

MetricSys indicates whether to use the metric system. The last field, intlØVers, contains a version number in its low-order byte and one of the following constants in its high-order byte:

```
CONST verUS      = Ø;
      verFrance  = 1;
      verBritain = 2;
      verGermany = 3;
      verItaly   = 4;
```

International Resource 1

The International Utilities Package contains the following data types for accessing international resource 1:

```
TYPE IntlIHndl = ^IntlIPtr; *** Following "Int" is the letter "l" ***
     Intl1Ptr  = ^Intl1Rec; *** Following "Intl" is the number "1" ***
     Intl1Rec  = PACKED RECORD
                   days:         ARRAY[1..7] OF STRING[15];  {day names}
                   months:       ARRAY[1..12] OF STRING[15]; {month names}
                   suppressDay:  Byte;   {Ø for day name, 255 for none}
                   longDateFmt:  Byte;   {order of long date elements}
                   dayleadingØ:  Byte;   {255 for leading Ø in day number}
                   abbrLen:      Byte;   {length for abbreviating names}
                   stØ:          PACKED ARRAY[1..4] OF CHAR; {strings }
                   st1:          PACKED ARRAY[1..4] OF CHAR; { for }
                   st2:          PACKED ARRAY[1..4] OF CHAR; { long }
                   st3:          PACKED ARRAY[1..4] OF CHAR; { date }
                   st4:          PACKED ARRAY[1..4] OF CHAR; { format}
                   intl1Vers:    INTEGER; {version information}
                   localRtn:     INTEGER  {routine for localizing string }
                                          { comparison; actually may be }
                                          { longer than one integer}
                 END;
```

All fields except the last two determine the long date format. The day names in the days array are ordered from Sunday to Saturday. (The month names are of course ordered from January to December.) As shown below, the longDateFmt field determines the order of the various parts of the date. StØ through st4 are strings (usually punctuation) that appear in the date.

longDateFmt	Long date format								
Ø	stØ	day name	st1	day	st2	month	st3	year	st4
255	stØ	day name	st1	month	st2	day	st3	year	st4

See Figure I-5 for examples of how the International Utilities Package formats dates based on these fields. The examples assume that the suppressDay and dayLeadingØ fields contain Ø. A suppressDay value of 255 causes the day name and st1 to be omitted, and a dayLeading value of 255 causes a Ø to appear before day numbers less than 1Ø.

longDateFmt	st0	st1	st2	st3	st4	Sample result
0	''	', '	'. '	' '	''	Mittwoch, 2. Februar 1984
255	''	', '	' '	', '	''	Wednesday, February 1, 1984

Figure I-5. Long Date Formats

AbbrLen is the number of characters to which month and day names should be abbreviated when abbreviation is desired.

The intl1Vers field contains version information with the same format as the intlØVers field of international resource Ø.

LocalRtn contains a routine that localizes the built-in character ordering (as described below under "International String Comparison").

International String Comparison

The International Utilities Package lets you compare strings in a way that accounts for diacritical marks and other special characters. The sort order built into the package may be localized through a routine stored in internatiomal resource 1.

The sort order is determined by a ranking of the entire Macintosh character set. The ranking can be thought of as a two-dimensional table. Each row of the table is a class of characters such as all A's (uppercase and lowercase, with and without diacritical marks). The characters are ordered within each row, but this ordering is secondary to the order of the rows themselves. For example, given that the rows for letters are ordered alphabetically, the following are all true under this scheme:

```
       'A' < 'a'
and   'Ab' < 'ab'
but   'Ac' > 'ab'
```

Even though 'A' < 'a' within the A row, 'Ac' > 'ab' because the order 'c' > 'b' takes precedence over the secondary ordering of the 'a' and the 'A'. In effect, the secondary ordering is ignored unless the comparison based on the primary ordering yields equality.

(note)

> The Pascal relational operators are used here for convenience only. String comparison in Pascal yields very different results, since it simply follows the ordering of the characters' ASCII codes.

When the strings being compared are of different lengths, each character in the longer string that doesn't correspond to a character in the shorter one compares "greater"; thus 'a' < 'ab'. This takes precedence over secondary ordering, so 'a' < 'Ab' even though 'A' < 'a'.

Besides letting you compare strings as described above, the International Utilities Package includes a routine that compares strings for equality without regard for secondary ordering. The effect on comparing letters, for example, is that diacritical marks are ignored and uppercase and lowercase are not distinguished.

Figure I-6 on the following page shows the two-dimensional ordering of the character set (from least to greatest as you read from top to bottom or left to right). The numbers on the left are ASCII codes corresponding to each row; ellipses (...) designate sequences of rows of just one character. Some codes do not correspond to rows (such as $61 through $7A, because lowercase letters are included in with their uppercase equivalents). See the Toolbox Event Manager manual for a table showing all the characters and their ASCII codes.

$00	ASCII NUL
...	
$1F	ASCII US
$20	space nonbreaking space
$21	!
$22	" « » “ ”
$23	#
$24	$
$25	%
$26	&
$27	' ‘ ’
$28	(
...	
$40	@
$41	A À Ä Ã Å a á à â ä ã å
$42	B b
$43	C Ç c ç
$45	E É e é è ê ë
$49	I i í ì î ï
$4E	N Ñ n ñ
$4F	O Ö Õ Ø o ó ò ô ö õ ø
$55	U Ü u ú ù û ü
$59	Y y ÿ
$5B	[
$5C	\
$5D	]
$5E	^
$5F	_
$60	`
$7B	{
$7C	|
$7D	}
$7E	~
$7F	ASCII DEL
$A0	†
...	
$AD	≠
$AE	Æ æ Œ œ (see remarks about ligatures)
$B0	∞
...	
$BD	Ω
$C0	¿
...	
$C9	…
$D0	–
$D2	—
$D6	÷
$D7	◊

letters not shown are like "B b"

Figure I-6. International Character Ordering

Characters combining two letters, as in the $AE row, are called ligatures. As shown in Figure I-7, they're actually expanded to the corresponding two letters, in the following sense:

- Primary ordering: The ligature is equal to the two-character sequence.
- Secondary ordering: The ligature is greater than the two-character sequence.

Standard:

AE Æ ae æ

OE Œ oe œ

Germany:

AE Ä Æ ae ä æ

OE Ö Œ oe ö œ

ss ß

UE Ü ue ü

Figure I-7. Ordering for Special Characters

Ligatures are ordered somewhat differently in Germany to accommodate umlauted characters (see Figure I-7). This is accomplished by means of the routine in international resource 1 for localizing the built-in character ordering. In the system resource file for Germany, this routine expands umlauted characters to the corresponding two letters (for example, "AE" for A-umlaut). The secondary ordering places the umlauted character between the two-character sequence and the ligature, if any. Likewise, the German double-s character expands to "ss".

In the system resource file for Great Britain, the localization routine in international resource 1 orders the pound currency sign between double quote and the pound weight sign (see Figure I-8). For the United States, France, and Italy, the localization routine does nothing.

$22 " « » “ ”
$A3 £
$23 #

Figure I-8. Special Ordering for Great Britain

Assembly-language note: The null localization routine consists of an RTS instruction.

*** Information on how to write your own localization routine is forthcoming. ***

Using the International Utilities Package

This section discusses how the routines in the International Utilities package 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.

The International Utilities Package is automatically read into memory from the system resource file when one of its routines is called. When a routine needs to access an international resource, it asks the Resource Manager to read the resource into memory. Together, the package and its resources occupy about 2K bytes.

As described in the *** not yet existing *** Operating System Utilities manual, you can get the date and time as a long integer from the utility routine ReadDateTime. If you need a string corresponding to the date or time, you can pass this long integer to the IUDateString or IUTimeString procedure in the International Utilities Package. These procedures determine the local format from the international resources read into memory by the Resource Manager (that is, resource type 'INTL' and resource ID Ø or 1). In some situations, you may need the format information to come instead from an international resource that you specify by its handle; if so, you can use IUDatePString or IUTimePString. This is useful, for example, if you want to use an international resource in a document's resource file after you've closed that file.

Applications that use measurements, such as on a ruler for setting margins and tabs, can call IUMetric to find out whether to use the metric system. This function simply returns the value of the corresponding field in international resource Ø. To access any other fields in an international resource--say, the currency format in international resource Ø--call IUGetIntl to get a handle to the resource. If you change any of the fields and want to write the changed resource to a resource file, the IUSetIntl procedure lets you do this.

To sort strings, you can use IUCompString or, if you're not dealing with Pascal strings, the more general IUMagString. These routines compare two strings and give their exact relationship, whether equal, less than, or greater than. Subtleties like diacritical marks and case differences are taken into consideration, as described above under "International String Comparison". If you need to know only whether two strings are equal, and want to ignore the subtleties, use IUEqualString (or the more general IUMagIDString) instead.

(note)

> The Operating System utility routine EqualString also compares two Pascal strings for equality. It's less sophisticated than IUEqualString in that it follows ASCII

order more strictly; for details, see the Operating System Utilities manual *** eventually ***.

International Utilities Package Routines

Assembly-language note: The macros for calling the International Utilities Package routines push one of the following routine selectors onto the stack and then invoke _Pack6:

Routine	Selector
IUDatePString	14
IUDateString	Ø
IUGetIntl	6
IUMagIDString	12
IUMagString	1Ø
IUMetric	4
IUTimePString	16
IUTimeString	2
IUSetIntl	8

PROCEDURE IUDateString (dateTime: LongInt; form: DateForm; VAR result: Str255);

Given a date and time as returned by the Operating System Utility routine ReadDateTime, IUDateString returns in the result parameter a string that represents the corresponding date. The form parameter has the following data type:

```
TYPE DateForm = (shortDate, longDate, abbrevDate);
```

ShortDate requests the short date format, longDate the long date, and abbrevDate the abbreviated long date. IUDateString determines the exact format from international resource Ø for the short date or 1 for the long date. See Figure I-1 above for examples of the standard formats. Notice that the short date contains a space in place of a leading zero when the format specifies "no leading zero", so the length of the result is always the same for short dates.

If the abbreviated long date is requested and the abbreviation length in international resource 1 is greater than the actual length of the name being abbreviated, IUDateString fills the abbreviation with NUL characters; the abbreviation length should not be greater than 15, the maximum name length.

PROCEDURE IUDatePString (dateTime: LongInt; form: DateForm; VAR result: Str255; intlParam: Handle);

IUDatePString is the same as IUDateString except that it determines the exact format of the date from the resource whose handle is passed in intlParam, overriding the resource that would otherwise be used.

PROCEDURE IUTimeString (dateTime: LongInt; wantSeconds: BOOLEAN; VAR result: Str255);

Given a date and time as returned by the Operating System Utility routine ReadDateTime, IUTimeString returns in the result parameter a string that represents the corresponding time of day. If wantSeconds is TRUE, seconds are included in the time; otherwise, only the hour and minute are included. IUTimeString determines the time format from international resource Ø. See Figure I-1 above for examples of the standard formats. Notice that the time contains a space in place of a leading zero when the format specifies "no leading zero", so the length of the result is always the same.

PROCEDURE IUTimePString (dateTime: LongInt; wantSeconds: BOOLEAN; VAR result: Str255; intlParam: Handle);

IUTimePString is the same as IUTimeString except that it determines the time format from the resource whose handle is passed in intlParam, overriding the resource that would otherwise be used.

FUNCTION IUMetric : BOOLEAN;

If international resource Ø specifies that the metric system is to be used, IUMetric returns TRUE; otherwise, it returns FALSE.

FUNCTION IUGetIntl (theID: INTEGER) : Handle;

IUGetIntl returns a handle to the international resource numbered theID (Ø or 1). It calls the Resource Manager function GetResource('INTL',theID). For example, if you want to access individual fields of international resource Ø, you can do the following:

```
VAR myHndl: Handle;
    intØ: IntlØHndl;
...
myHndl := IUGetIntl(Ø);
intØ := POINTER(ORD(myHndl));
```

PROCEDURE IUSetIntl (refNum: INTEGER; theID: INTEGER; intlParam: Handle);

In the resource file having the reference number refNum, IUSetIntl sets the international resource numbered theID (Ø or 1) to the data pointed to by intlParam. The data may be either an existing resource or data that hasn't yet been written to a resource file. IUSetIntl adds the resource to the specified file or replaces the resource if it's already there.

FUNCTION IUCompString (aStr,bStr: Str255) : INTEGER; [Pascal only]

IUCompString compares aStr and bStr as described above under "International String Comparison", taking both primary and secondary ordering into consideration. It returns one of the values listed below.

Result	Meaning	Example	
		aStr	bStr
-1	aStr is less than bStr	'Ab'	'ab'
Ø	aStr equals bStr	'Ab'	'Ab'
1	aStr is greater than bStr	'Ac'	'ab'

Assembly-language note: IUCompString was created for the convenience of Pascal programmers; there's no trap for it. It eventually calls IUMagString, which is what you should use from assembly language.

FUNCTION IUMagString (aPtr,bPtr: Ptr; aLen,bLen: INTEGER) : INTEGER;

IUMagString is the same as IUCompString (above) except that instead of comparing two Pascal strings, it compares the string defined by aPtr and aLen to the string defined by bPtr and bLen. The pointer points to the first character of the string (any byte in memory, not necessarily word-aligned), and the length specifies the number of characters in the string.

FUNCTION IUEqualString (aStr,bStr: Str255) : INTEGER; [Pascal only]

IUEqualString compares aStr and bStr for equality without regard for secondary ordering, as described above under "International String Comparison". If the strings are equal, it returns Ø; otherwise, it returns 1. For example, if the strings are 'Rose' and 'rose', IUEqualString considers them equal and returns Ø.

(note)

See also EqualString in the Operating System Utilities manual *** doesn't yet exist ***.

Assembly-language note: IUEqualString was created for the convenience of Pascal programmers; there's no trap for it. It eventually calls IUMagIDString, which is what you should use from assembly language.

FUNCTION IUMagIDString (aPtr,bPtr: Ptr; aLen,bLen: INTEGER) : INTEGER;

IUMagIDString is the same as IUEqualString (above) except that instead of comparing two Pascal strings, it compares the string defined by aPtr and aLen to the string defined by bPtr and bLen. The pointer points to the first character of the string (any byte in memory, not necessarily word-aligned), and the length specifies the number of characters in the string.

THE BINARY-DECIMAL CONVERSION PACKAGE

The Binary-Decimal Conversion Package contains only two routines: one converts an integer from its internal (binary) form to a string that represents its decimal (base 1Ø) value; the other converts a decimal string to the corresponding integer.

*** In the Inside Macintosh manual, the documentation of this package will be at the end of the volume that describes the User Interface Toolbox. ***

You should already be familiar with the Package Manager, and packages in general.

The Binary-Decimal Conversion Package is automatically read into memory when one of its routines is called; it occupies a total of about 2ØØ bytes. The routines are described below. They're register-based, so the Pascal form of each is followed by a box containing information needed to use the routine from assembly language. (For general information on using assembly language, see Programming Macintosh Applications in Assembly Language.)

Assembly-language note: The macros for calling the Binary-Decimal Conversion Package routines push one of the following routine selectors onto the stack and then invoke _Pack7:

Routine	Selector
NumToString	Ø
StringToNum	1

PROCEDURE NumToString (theNum: LongInt; VAR theString: Str255);

Trap macro	_NumToString	
On entry	AØ:	pointer to theString (length byte followed by characters)
	DØ:	theNum (long integer)
On exit	AØ:	pointer to theString

NumToString converts theNum to a string that represents its decimal value, and returns the result in theString. If the value is negative, the string begins with a minus sign; otherwise, the sign is omitted. Leading zeroes are suppressed, except that the value Ø produces 'Ø'.

For example:

theNum	theString
12	'12'
-23	'-23'
Ø	'Ø'

PROCEDURE StringToNum (theString: Str255; VAR theNum: LongInt);

Trap macro	_StringToNum	
On entry	AØ:	pointer to theString (length byte followed by characters)
On exit	DØ:	theNum (long integer)

Given a string representing a decimal integer, StringToNum converts it to the corresponding integer and returns the result in theNum. The string may begin with a plus or minus sign. For example:

theString	theNum
'12'	12
'-23'	-23
'-Ø'	Ø
'Ø55'	55

The magnitude of the integer is converted modulo 2^32, and the 32-bit result is negated if the string begins with a minus sign; integer overflow occurs if the magnitude is greater than 2^31-1. (Negation is done by taking the two's complement--reversing the state of each bit and then adding 1.) For example:

theString		theNum
'2147483648'	(magnitude is 2^31)	-2147483648
'-2147483648'		-2147483648
'4294967295'	(magnitude is 2^32-1)	-1
'-4294967295'		1

StringToNum doesn't actually check whether the characters in the string are between 'Ø' and '9'; instead, since the ASCII codes for 'Ø' through '9' are $3Ø through $39, it just masks off the last four bits and uses them as a digit. For example, '2:' is converted to the number 3Ø because the ASCII code for ':' is $3A. Leading spaces before the first digit are treated as zeroes, since the ASCII code for a space is $2Ø. Given that the ASCII codes for 'C', 'A', and 'T' are $43, $41, and $54, respectively, consider the following examples:

theString	theNum
'CAT'	314
'+CAT'	314
'-CAT'	-314

THE STANDARD FILE PACKAGE

The Standard File Package provides the standard user interface for specifying a file to be saved or opened. It allows the file to be on a disk in any drive connected to the Macintosh, and lets a currently inserted disk be ejected so that another one can be inserted.

*** In the Inside Macintosh manual, the documentation of this package will be at the end of the volume that describes the Toolbox. ***

You should already be familiar with the following:

- the basic concepts and structures behind QuickDraw, particularly points and rectangles
- the Toolbox Event Manager
- the Dialog Manager, especially the ModalDialog procedure
- the Package Manager and packages in general

About the Standard File Package

Standard Macintosh applications should have a File menu from which the user can save and open documents, via the Save, Save As, and Open commands. In response to these commands, the application can call the Standard File Package to find out the document name and let the user switch disks if desired. As described below, a dialog box is presented for this purpose. (More details and illustrations are given later in the descriptions of the individual routines.)

When the user chooses Save As, or Save when the document is untitled, the application needs a name for the document. The corresponding dialog box lets the user enter the document name and click a button labeled "Save" (or just click "Cancel" to abort the command). By convention, the dialog box comes up displaying the current document name, if any, so the user can edit it.

In response to an Open command, the application needs to know which document to open. The corresponding dialog box displays the names of all documents that might be opened, and the user chooses one by clicking it and then clicking a button labeled "Open". A vertical scroll bar allows scrolling through the names if there are more than can be shown at once.

Both of these dialog boxes let the user:

- insert a disk in an external drive connected to the Macintosh
- eject a disk from either drive and insert another

- initialize and name an inserted disk that's uninitialized

- switch from one drive to another

On the right in the dialog box, separated from the rest of the box by a gray line, there's a disk name with one or two buttons below it; Figure S-1 shows what this looks like when an external drive is connected to the Macintosh but currently has no disk in it. Notice that the Drive button is inactive (dimmed). After the user inserts a disk in the external drive (and, if necessary, initializes and names it), the Drive button becomes active. If there's no external drive, the Drive button isn't displayed at all.

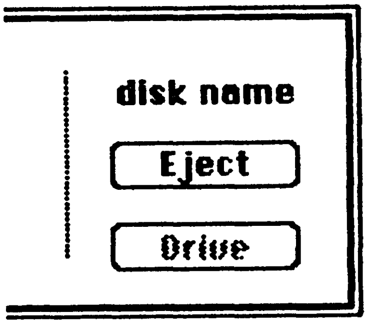


Figure S-1. Partial Dialog Box

The disk name displayed in the dialog box is the name of the current disk, initially the disk you used to start up the Macintosh. The user can click Eject to eject the current disk and insert another, which then becomes the current disk. If there's an external drive, clicking the Drive button changes the current disk from the one in the external drive to the one in the internal drive or vice versa. The Drive button is inactive whenever there's only one disk inserted.

If an uninitialized or otherwise unreadable disk is inserted, the Standard File Package calls the Disk Initialization Package to provide the standard user interface for initializing and naming a disk.

Using the Standard File Package

This section discusses how the routines in the Standard File Package 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.

The Standard File Package and the resources it uses are automatically read into memory when one of its routines is called. It in turn reads the Disk Initialization Package into memory if a disk is ejected; together they occupy about 6.5K bytes.

Call SFPutFile when your application is to save to a file and needs to get the name of the file from the user. Standard applications should do this when the user chooses Save As from the File menu, or Save when the document is untitled. SFPutFile displays a dialog box allowing the user to enter a file name.

Similarly, SFGetFile is useful whenever your application is to open a file and needs to know which one, such as when the user chooses the Open command from a standard application's File menu. SFGetFile displays a dialog box with a list of file names to choose from.

You pass these routines a reply record, as shown below, and they fill it with information about the user's reply.

```
TYPE SFReply = RECORD
                 good:     BOOLEAN;     {FALSE if ignore command}
                 copy:     BOOLEAN;     {not used}
                 fType:    OSType;      {file type or not used}
                 vRefNum:  INTEGER;     {volume reference number}
                 version:  INTEGER;     {file's version number}
                 fName:    STRING[63]   {file name}
               END;
```

The first field of this record determines whether the file operation should take place or the command should be ignored (because the user clicked the Cancel button in the dialog box). The fType field is used by SFGetFile to store the file's type. The vRefNum, version, and fName fields identify the file chosen by the user; the application passes their values on to the File Manager routine that does the actual file operation. VRefNum contains the volume reference number of the volume containing the file. Currently the version field always contains Ø; the use of nonzero version numbers is not supported by this package. For more information on files, volumes, and file operations, see the File Manager manual *** doesn't yet exist ***.

Both SFPutFile and SFGetFile allow you to use a nonstandard dialog box; two additional routines, SFPPutFile and SFPGetFile, provide an even more convenient and powerful way of doing this.

Standard File Package Routines

Assembly-language note: The macros for calling the Standard File Package routines push one of the following routine selectors onto the stack and then invoke _Pack3:

Routine	Selector
SFGetFile	2
SFPGetFile	4
SFPPutFile	3
SFPutFile	1

```
PROCEDURE SFPutFile (where: Point; prompt: Str255; origName: Str255;
             dlgHook: ProcPtr; VAR reply: SFReply);
```

SFPutFile displays a dialog box allowing the user to specify a file to which data will be written (as during a Save or Save As command). It then repeatedly gets and handles events until the user either confirms the command after entering an appropriate file name or aborts the command by clicking Cancel in the dialog. It reports the user's reply by filling the fields of the reply record specified by the reply parameter, as described above; the fType field of this record isn't used.

The general appearance of the standard SFPutFile dialog box is shown in Figure S-2. The where parameter specifies the location of the top left corner of the dialog box in global coordinates. The prompt parameter is a line of text to be displayed as a statText item in the dialog box, where shown in Figure S-2. The origName parameter contains text that appears as an enabled, selected editText item; for the standard document-saving commands, it should be the current name of the document, or the empty string (to display an insertion point) if the document hasn't been named yet.

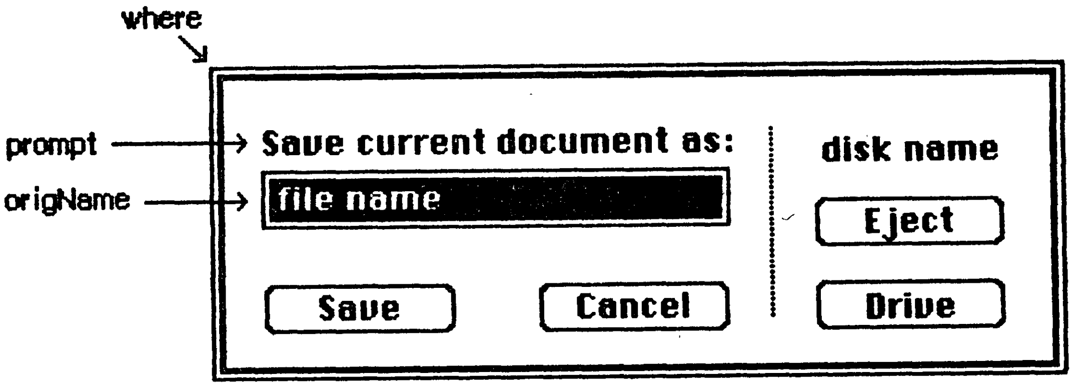


Figure S-2. Standard SFPutFile Dialog

If you want to use the standard SFPutFile dialog box, pass NIL for dlgHook; otherwise, see the information for advanced programmers below.

SFPutFile repeatedly calls the Dialog Manager procedure ModalDialog. When an event involving an enabled dialog item occurs, ModalDialog handles the event and returns the item number, and SFPutFile responds as follows:

- If the Eject or Drive button is clicked, or a disk is inserted, SFPutFile responds as described above under "About the Standard File Package".

- Text entered into the editText item is stored in the fName field of the reply record. (SFPutFile keeps track of whether there's currently any text in the item, and makes the Save button inactive if not.)

- If the Save button is clicked, SFPutFile determines whether the file name in the fName field of the reply record is appropriate. If so, it returns control to the application with the first field of the reply record set to TRUE; otherwise, it responds accordingly, as described below.

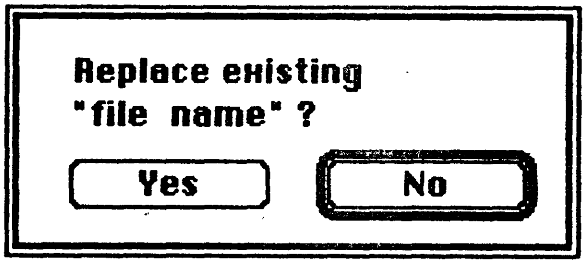
- If the Cancel button in the dialog is clicked, SFPutFile returns control to the application with the first field of the reply record set to FALSE.

(note)

Notice that disk insertion is one of the user actions listed above, even though ModalDialog normally ignores disk-inserted events. The reason this works is that SFPutFile calls ModalDialog with a filterProc function that checks for a disk-inserted event and returns a "fake", very large item number if one occurs; SFPutFile recognizes this item number as an indication that a disk was inserted.

The situations that may cause an entered name to be inappropriate, and SFPutFile's response to each, are as follows:

- If a file with the specified name already exists on the disk and is different from what was passed in the origName parameter, the alert in Figure S-3 is displayed. If the user clicks Yes, the file name is appropriate.



Replace existing
"file name" ?

Figure S-3. Alert for Existing File

- If the disk to which the file should be written is locked, the alert in Figure S-4 is displayed. If a system error occurs, a similar alert is displayed, with a corresponding message explaining the problem.

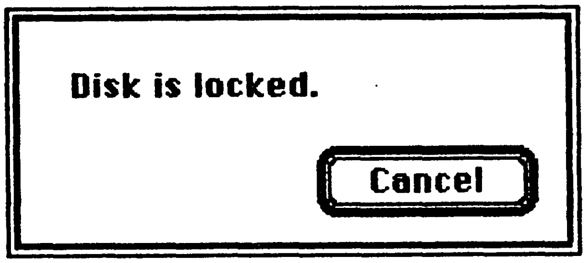

Disk is locked.

Figure S-4. Alert for Locked Disk

(note)

> The user may specify a disk name (preceding the file name and separated from it by a colon). If the disk isn't currently in a drive, an alert similar to the one in Figure S-4 is displayed. The ability to specify a disk name is supported for historical reasons only; users should not be encouraged to do it.

After the user clicks No or Cancel in response to one of these alerts, SFPutFile dismisses the alert box and continues handling events (so a different name may be entered).

Advanced programmers: You can create your own dialog box rather than use the standard SFPutFile dialog. To do this, you must provide your own dialog template and store it in your application's resource file with the same resource ID that the standard template has in the system resource file:

```
CONST putDlgID = -3999;  {SFPutFile dialog template ID}
```

(note)

> The SFPPutFile procedure, described below, lets you use any resource ID for your nonstandard dialog box.

Your dialog template must specify that the dialog window be invisible, and your dialog must contain all the standard items, as listed below. The appearance and location of these items in your dialog may be different. You can make an item "invisible" by giving it a display rectangle that's off the screen. The display rectangle for each item in the standard dialog box is given below. The rectangle for the standard dialog box itself is (Ø, Ø, 3Ø4, 1Ø4).

Item number	Item	Standard display rectangle
1	Save button	(12, 74, 82, 92)
2	Cancel button	(114, 74, 184, 92)
3	Prompt string (statText)	(12, 12, 184, 28)
4	UserItem for disk name	(2ø9, 16, 295, 34)
5	Eject button	(217, 43, 287, 61)
6	Drive button	(217, 74, 287, 92)
7	EditText item for file name	(14, 34, 182, 5ø)
8	UserItem for gray line	(2øø, 16, 2ø1, 88)

(note)

> Remember that the display rectangle for any "invisible" item must be at least about 2ø pixels wide. *** This will be discussed in a future draft of the Dialog Manager manual. ***

If your dialog has additional items beyond the the standard ones, or if you want to handle any of the standard items in a nonstandard manner, you must write your own dlgHook function and point to it with dlgHook. Your dlgHook function should have two parameters and return an integer value. For example, this is how it would be declared if it were named MyDlg:

```
FUNCTION MyDlg (item: INTEGER; theDialog: DialogPtr) : INTEGER;
```

Immediately after calling ModalDialog, SFPutFile calls your dlgHook function, passing it the item number returned by ModalDialog and a pointer to the dialog record describing your dialog box. Using these two parameters, your dlgHook function should determine how to handle the event. There are predefined constants for the item numbers of standard enabled items, as follows:

```
CONST putSave   = 1;  {Save button}
      putCancel = 2;  {Cancel button}
      putEject  = 5;  {Eject button}
      putDrive  = 6;  {Drive button}
      putName   = 7;  {editText item for file name}
```

ModalDialog also returns the "fake" item number 1øø when a disk-inserted event occurs, as detected by its filterProc function.

After handling the event (or, perhaps, after ignoring it) the dlgHook function must return an item number to SFPutFile. If the item number is one of those listed above, SFPutFile responds in the standard way; otherwise, it does nothing.

(note)

> For advanced programmers who want to change the appearance of the alerts displayed when an inappropriate file name is entered, the resource IDs of those alerts in the system resource file are listed below.

Alert	Resource ID
Existing file	-3996
Locked disk	-3997
System error	-3995
Disk not found	-3994

```
PROCEDURE SFPPutFile (where: Point; prompt: Str255; origName: Str255;
            dlgHook: ProcPtr; VAR reply: SFReply; dlgID: INTEGER;
            filterProc: ProcPtr);
```

SFPPutFile is an alternative to SFPutFile for advanced programmers who want to use a nonstandard dialog box. It's the same as SFPutFile except for the two additional parameters dlgID and filterProc.

DlgID is the resource ID of the dialog template to be used instead of the standard one (so you can use whatever ID you wish rather than the same one as the standard).

The filterProc parameter determines how ModalDialog will filter events when called by SFPPutFile. If filterProc is NIL, ModalDialog does the standard filtering that it does when called by SFPutFile; otherwise, filterProc should point to a function for ModalDialog to execute **after** doing the standard filtering. The function must be the same as one you'd pass directly to ModalDialog in its filterProc parameter. (See the Dialog Manager manual for more information.)

```
PROCEDURE SFGetFile (where: Point; prompt: Str255; fileFilter: ProcPtr;
            numTypes: INTEGER; typeList: SFTypeList; dlgHook: ProcPtr;
            VAR reply: SFReply);
```

SFGetFile displays a dialog box listing the names of a specific group of files from which the user can select one to be opened (as during an Open command). It then repeatedly gets and handles events until the user either confirms the command after choosing a file name or aborts the command by clicking Cancel in the dialog. It reports the user's reply by filling the fields of the reply record specified by the reply parameter, as described above under "Using the Standard File Package".

The general appearance of the standard SFGetFile dialog box is shown in Figure S-5. File names are sorted in order of the ASCII codes of their characters, ignoring diacritical marks and mapping lowercase characters to their uppercase equivalents. If there are more file names than can be displayed at one time, the scroll bar is active; otherwise, the scroll bar is inactive.

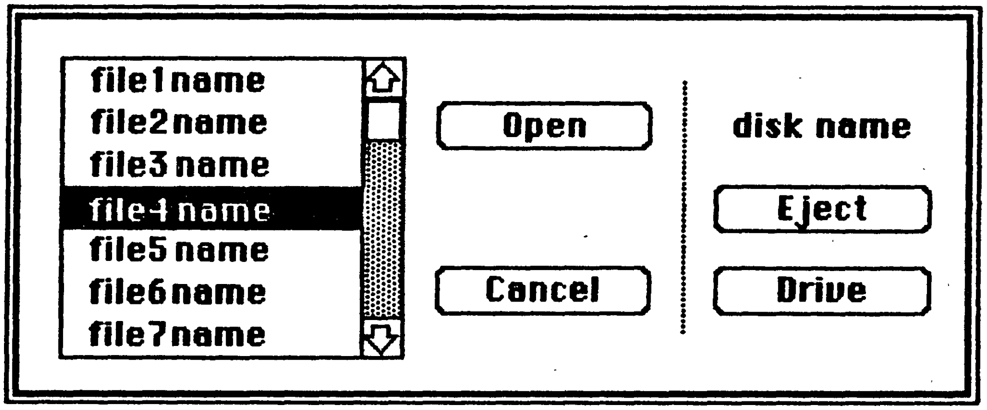


Figure S-5. Standard SFGetFile Dialog

The where parameter specifies the location of the top left corner of the dialog box in global coordinates. The prompt parameter is ignored; it's there for historical purposes only.

The fileFilter, numTypes, and typeList parameters determine which files appear in the dialog box. SFGetFile first looks at numTypes and typeList to determine what types of files to display, then it executes the function pointed to by fileFilter (if any) to do additional filtering on which files to display. File types are discussed in the manual The Structure of a Macintosh Application. For example, if the application is concerned only with pictures, you won't want to display the names of any text files.

Pass -1 for numTypes to display all types of files; otherwise, pass the number of file types you want to display, and pass the types themselves in typeList. The SFTypeList data type is defined as follows:

```
TYPE SFTypeList = ARRAY [Ø..3] OF OSType;
```

(note)

> This array is declared for a reasonable maximum number of types (four). If you need to specify more than four types, declare your own array type with the desired number of entries (and use the @ operator to pass a pointer to it).

If fileFilter isn't NIL, SFGetFile executes the function it points to for each file, to determine whether the file should be displayed. The fileFilter function has one parameter and returns a Boolean value. For example:

```
FUNCTION MyFileFilter (paramBlock: ParmBlkPtr) : BOOLEAN;
```

SFGetFile passes this function the file information it gets by calling the File Manager procedure PBGetFInfo (see the *** forthcoming *** File Manager manual for details). The function selects which files should appear in the dialog by returning FALSE for every file that should be shown and TRUE for every file that shouldn't be shown.

(note)

> As described in the File Manager manual, a flag can be set that tells the Finder not to display a particular file's icon on the desktop; this has no effect on whether SFGetFile will list the file name.

If you want to use the standard SFGetFile dialog box, pass NIL for dlgHook; otherwise, see the information for advanced programmers below.

Like SFPutFile, SFGetFile repeatedly calls the Dialog Manager procedure ModalDialog. When an event involving an enabled dialog item occurs, ModalDialog handles the event and returns the item number, and SFGetFile responds as follows:

- If the Eject or Drive button is clicked, or a disk is inserted, SFGetFile responds as described above under "About the Standard File Package".
- If clicking or dragging occurs in the scroll bar, the contents of the dialog box are redrawn accordingly.
- If a file name is clicked, it's selected and stored in the fName field of the reply record. (SFGetFile keeps track of whether a file name is currently selected, and makes the Open button inactive if not.)
- If the Open button is clicked, SFGetFile returns control to the application with the first field of the reply record set to TRUE.
- If a file name is double-clicked, SFGetFile responds as if the user clicked the file name and then the Open button.
- If the Cancel button in the dialog is clicked, SFGetFile returns control to the application with the first field of the reply record set to FALSE.

If a key (other than a modifier key) is pressed, SFGetFile selects the first file name starting with the character typed. If no file name starts with that character, it selects the first file name starting with a character whose ASCII code is greater than the character typed.

Advanced programmers: You can create your own dialog box rather than use the standard SFGetFile dialog. To do this, you must provide your own dialog template and store it in your application's resource file with the same resource ID that the standard template has in the system resource file:

```
        CONST getDlgID = -4ØØØ;  {SFGetFile dialog template ID}
```

(note)

> The SFPGetFile procedure, described below, lets you use any resource ID for your nonstandard dialog box.

Your dialog template must specify that the dialog window be invisible, and your dialog must contain all the standard items, as listed below. The appearance and location of these items in your dialog may be different. You can make an item "invisible" by giving it a display rectangle that's off the screen. The display rectangle for each in the standard dialog box is given below. The rectangle for the standard dialog box itself is (0, 0, 348, 136).

Item number	Item	Standard display rectangle
1	Open button	(152, 28, 232, 46)
2	Invisible button	(1152, 59, 1232, 77)
3	Cancel button	(152, 90, 232, 108)
4	UserItem for disk name	(248, 28, 344, 46)
5	Eject button	(256, 59, 336, 77)
6	Drive button	(256, 90, 336, 108)
7	UserItem for file name list	(12, 11, 125, 125)
8	UserItem for scroll bar	(124, 11, 140, 125)
9	UserItem for gray line	(244, 20, 245, 116)
10	Invisible text (statText)	(1044, 20, 1145, 116)

If your dialog has additional items beyond the the standard ones, or if you want to handle any of the standard items in a nonstandard manner, you must write your own dlgHook function and point to it with dlgHook. Your dlgHook function should have two parameters and return an integer value. For example, this is how it would be declared if it were named MyDlg:

```
FUNCTION MyDlg (item: INTEGER; theDialog: DialogPtr) : INTEGER;
```

Immediately after calling ModalDialog, SFGetFile calls your dlgHook function, passing it the item number returned by ModalDialog and a pointer to the dialog record describing your dialog box. Using these two parameters, your dlgHook function should determine how to handle the event. There are predefined constants for the item numbers of standard enabled items, as follows:

```
CONST getOpen   = 1;  {Open button}
      getCancel = 3;  {Cancel button}
      getEject  = 5;  {Eject button}
      getDrive  = 6;  {Drive button}
      getNmList = 7;  {userItem for file name list}
      getScroll = 8;  {userItem for scroll bar}
```

ModalDialog also returns "fake" item numbers in the following situations, which are detected by its filterProc function:

- When a disk-inserted event occurs, it returns 100.

- When a key-down event occurs, it returns 1000 plus the ASCII code of the character.

After handling the event (or, perhaps, after ignoring it) your dlgHook function must return an item number to SFGetFile. If the item number is one of those listed above, SFGetFile responds in the standard way;

otherwise, it does nothing.

```
PROCEDURE SFPGetFile (where: Point; prompt: Str255; fileFilter:
            ProcPtr; numTypes: INTEGER; typeList: SFTypeList; dlgHook:
            ProcPtr; VAR reply: SFReply; dlgID: INTEGER; filterProc:
            ProcPtr);
```

SFPGetFile is an alternative to SFGetFile for advanced programmers who want to use a nonstandard dialog box. It's the same as SFGetFile except for the two additional parameters dlgID and filterProc.

DlgID is the resource ID of the dialog template to be used instead of the standard one (so you can use whatever ID you wish rather than the same one as the standard).

The filterProc parameter determines how ModalDialog will filter events when called by SFPGetFile. If filterProc is NIL, ModalDialog does the standard filtering that it does when called by SFGetFile; otherwise, filterProc should point to a function for ModalDialog to execute **after** doing the standard filtering. Note, however, that the standard filtering will detect key-down events only if the dialog template ID is the standard one.

THE DISK INITIALIZATION PACKAGE

The Disk Initialization Package provides routines for initializing disks to be accessed with the Macintosh Operating System's File Manager and Disk Driver. A single routine lets you easily present the standard user interface for initializing and naming a disk; the Standard File Package calls this routine when the user inserts an uninitialized disk. You can also use the Disk Initialization Package to perform each of the three steps of initializing a disk separately if desired.

*** In the Inside Macintosh manual, the documentation of this package will be at the end of the volume that describes the Operating System. ***

You should already be familiar with the following:

- the basic concepts and structures behind QuickDraw, particularly points

- the Toolbox Event Manager

- the File Manager *** the File Manager manual doesn't yet exist ***

- the Package Manager and packages in general

Using the Disk Initialization Package

This section discusses how the routines in the Disk Initialization package 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.

The Disk Initialization Package and the resources it uses are automatically read into memory from the system resource file when one of the routines in the package is called. Together, the package and its resources occupy about 2.5K bytes. If the disk containing the system resource file isn't currently in a Macintosh disk drive, the user will be asked to switch disks and so may have to remove the one to be initialized. To avoid this, you can use the DILoad procedure, which explicitly reads the necessary resources into memory and makes them unpurgeable. You would need to call DILoad before explicitly ejecting the system disk or before any situations where it may be switched with another disk (except for situations handled by the Standard File Package, which calls DILoad itself).

(note)

> The resources used by the Disk Initialization Package consist of a single dialog and its associated items, even though the package may present what seem to be a number of different dialogs. A special technique was used to allow the single dialog to contain all possible dialog items with only some of them visible at one time. ***

This technique will be documented in the next draft of the Dialog Manager manual. ***

When you no longer need to have the Disk Initialization Package in memory, call DIUnload. The Standard File Package calls DIUnload before returning.

When a disk-inserted event occurs, the system attempts to mount the volume (by calling the File Manager function PBMountVol) and returns PBMountVol's result code in the high-order word of the event message. In response to such an event, your application can examine the result code in the event message and call DIBadMount if an error occurred (that is, if the volume could not be mounted). If the error is one that can be corrected by initializing the disk, DIBadMount presents the standard user interface for initializing and naming the disk, and then mounts the volume itself. For other errors, it justs ejects the disk; these errors are rare, and may reflect a problem in your program.

(note)

> Disk-inserted events during standard file saving and opening are handled by the Standard File Package. You'll call DIBadMount only in other, less common situations (for example, if your program explicitly ejects disks, or if you want to respond to the user's inserting an uninitialized disk when not expected).

Disk initialization consists of three steps, each of which can be performed separately by the functions DIFormat, DIVerify, and DIZero. Normally you won't call these in a standard application, but they may be useful in special utility programs that have a nonstandard interface.

Disk Initialization Package Routines

Assembly-language note: The macros for calling the Disk Initialization Package routines push one of the following routine selectors onto the stack and then invoke _Pack2:

Routine	Selector
DIBadMount	Ø
DIFormat	6
DILoad	2
DIUnload	4
DIVerify	8
DIZero	1Ø

PROCEDURE DILoad;

DILoad reads the Disk Initialization Package, and its associated dialog and dialog items, from the system resource file into memory and makes them unpurgeable.

(note)

> DIFormat, DIVerify, and DIZero don't need the dialog, so if you use only these routines you can call the Resource Manager function GetResource to read just the package resource into memory (and the Memory Manager procedure HNoPurge to make it unpurgeable).

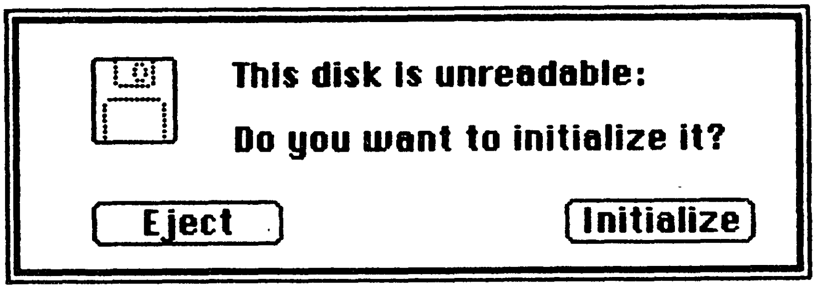
PROCEDURE DIUnload;

DIUnload makes the Disk Initialization Package (and its associated dialog and dialog items) purgeable.

FUNCTION DIBadMount (where: Point; evtMessage: LongInt) : INTEGER;

Call DIBadMount when a disk-inserted event occurs if the result code in the high-order word of the associated event message indicates an error (that is, the result code is other than noErr). Given the event message in evtMessage, DIBadMount evaluates the result code and either ejects the disk or lets the user initialize and name it. The low-order word of the event message contains the drive number. The where parameter specifies the location (in global coordinates) of the top left corner of the dialog box displayed by DIBadMount.

If the result code passed is extFSErr, mFulErr, nsDrvErr, paramErr, or volOnLinErr, DIBadMount simply ejects the disk from the drive and returns the result code. If the result code ioErr, badMDBErr, or noMacDskErr is passed, the error can be corrected by initializing the disk; DIBadMount displays a dialog box that describes the problem and asks whether the user wants to initialize the disk. For the result code ioErr, the dialog box shown in Figure D-1 is displayed. (This happens if the disk is brand new.) For badMDBErr and noMacDskErr, DIBadMount displays a similar dialog box in which the description of the problem is "This disk is damaged" and "This is not a Macintosh disk", respectively.



This disk is unreadable:

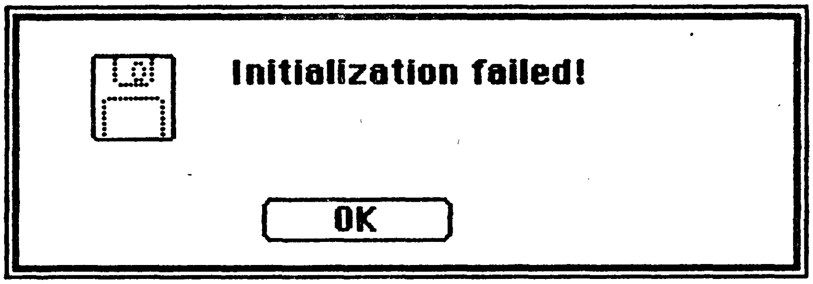
Do you want to initialize it?

Figure D-1. Disk Initialization Dialog for IOErr

(note)

> Before presenting the disk initialization dialog, DIBadMount checks whether the drive contains an already mounted volume; if so, it ejects the disk and returns 2 as its result. This will happen rarely and may reflect an error in your program (for example, you forgot to call DILoad and the user had to switch to the disk containing the system resource file).

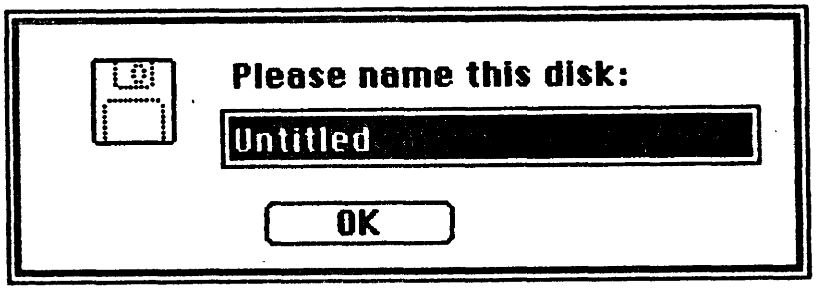
If the user responds to the disk initialization dialog by clicking the Eject button, DIBadMount ejects the disk and returns 1 as its result. If the Initialize button is clicked, a box displaying the message "Initializing disk..." appears, and DIBadMount attempts to initialize the disk. If initialization fails, the disk is ejected and the user is informed as shown in Figure D-2; after the user clicks OK, DIBadMount returns a negative result code ranging from firstDskErr to lastDskErr, indicating that a low-level disk error occurred.



Initialization failed!

Figure D-2. Initialization Failure Dialog

If the disk is successfully initialized, the dialog box in Figure D-3 appears. After the user names the disk and clicks OK, DIBadMount mounts the volume by calling the File Manager function PBMountVol and returns PBMountVol's result code (noErr if no error occurs).



Please name this disk:

Untitled

Figure D-3. Dialog for Naming a Disk

<u>Result codes</u>	noErr	No error
	extFSErr	External file system
	mFulErr	Memory full
	nsDrvErr	No such drive
	paramErr	Bad drive number
	volOnLinErr	Volume already on-line
	firstDskErr through lastDskErr	Low-level disk error
<u>Other results</u>	1	User clicked Eject
	2	Mounted volume in drive

FUNCTION DIFormat (drvNum: INTEGER) : OSErr;

DIFormat formats the disk in the drive specified by the given drive number and returns a result code indicating whether the formatting was completed successfully or failed. Formatting a disk consists of writing special information onto it so that the Disk Driver can read from and write to the disk.

<u>Result codes</u>	noErr	No error
	firstDskErr through lastDskErr	Low-level disk error

FUNCTION DIVerify (drvNum: INTEGER) : OSErr;

DIVerify verifies the format of the disk in the drive specified by the given drive number; it reads each bit from the disk and returns a result code indicating whether all bits were read successfully or not.

<u>Result codes</u>	noErr	No error
	firstDskErr through lastDskErr	Low-level disk error

FUNCTION DIZero (drvNum: INTEGER; volName: Str255) : OSErr;

On the unmounted volume in the drive specified by the given drive number, DIZero writes the volume information, a block map, and a file directory as for a volume with no files; the volName parameter specifies the volume name to be included in the volume information. This is the last step in initialization (after formatting and verifying) and makes any files that are already on the volume permanently inaccessible. If the operation fails, DIZero returns a result code indicating that a low-level disk error occurred; otherwise, it mounts the volume by calling the File Manager function PBMountVol and returns PBMountVol's result code (noErr if no error occurs).

Result codes	noErr	No error
	badMDBErr	Bad master directory block
	extFSErr	External file system
	ioErr	Disk I/O error
	mFulErr	Memory full
	noMacDskErr	Not a Macintosh volume
	nsDrvErr	No such drive
	paramErr	Bad drive number
	volOnLinErr	Volume already on-line
	firstDskErr through lastDskErr	Low-level disk error

SUMMARY OF THE PACKAGE MANAGER

Constants

```
CONST { Resource IDs for packages }

      dskInit = 2;  {Disk Initialization}
      stdFile = 3;  {Standard File}
      flPoint = 4;  {Floating-Point Arithmetic}
      trFunc  = 5;  {Transcendental Functions}
      intUtil = 6;  {International Utilities}
      bdConv  = 7;  {Binary-Decimal Conversion}
```

Routines

```
PROCEDURE InitPack (packID: INTEGER);
PROCEDURE InitAllPacks;
```

Assembly-Language Information

Constants

```
; Resource IDs for packages

dskInit     .EQU     2  ;Disk Initialization
stdFile     .EQU     3  ;Standard File
flPoint     .EQU     4  ;Floating-Point Arithmetic
trFunc      .EQU     5  ;Transcendental Functions
intUtil     .EQU     6  ;International Utilities
bdConv      .EQU     7  ;Binary-Decimal Conversion
```

SUMMARY OF THE INTERNATIONAL UTILITIES PACKAGE

Constants

```
CONST { Masks for currency format }

      currSymLead    = 16;  {set if currency symbol leads}
      currNegSym     = 32;  {set if minus sign for negative}
      currTrailingZ  = 64;  {set if trailing decimal zeroes}
      currLeadingZ   = 128; {set if leading integer zero}

      { Order of short date elements }

      mdy = Ø;  {month day year}
      dmy = 1;  {day month year}
      ymd = 2;  {year month day}

      { Masks for short date format }

      dayLeadingZ = 32;  {set if leading zero for day}
      mntLeadingZ = 64;  {set if leading zero for month}
      century     = 128; {set if century included}

      { Masks for time format }

      secLeadingZ = 32;  {set if leading zero for seconds}
      minLeadingZ = 64;  {set if leading zero for minutes}
      hrLeadingZ  = 128; {set if leading zero for hours}

      { High-order byte of version information }

      verUS      = Ø;
      verFrance  = 1;
      verBritain = 2;
      verGermany = 3;
      verItaly   = 4;
```

Data Types

```
TYPE IntlØHndl = ^IntlØPtr;
     IntlØPtr  = ^IntlØRec;
```

```
Intl0Rec  = PACKED RECORD
              decimalPt:    CHAR; {decimal point character}
              thousSep:     CHAR; {thousands separator}
              listSep:      CHAR; {list separator}
              currSym1:     CHAR; {currency symbol}
              currSym2:     CHAR;
              currSym3:     CHAR;
              currFmt:      Byte; {currency format}
              dateOrder:    Byte; {order of short date elements}
              shortDateFmt: Byte; {short date format}
              dateSep:      CHAR; {date separator}
              timeCycle:    Byte; {Ø if 24-hour cycle, 255 if 12-hour}
              timeFmt:      Byte; {time format}
              mornStr:      PACKED ARRAY[1..4] OF CHAR;
                             {trailing string for first 12-hour cycle}
              eveStr:       PACKED ARRAY[1..4] OF CHAR;
                             {trailing string for last 12-hour cycle}
              timeSep:      CHAR; {time separator}
              time1Suff:    CHAR; {trailing string for 24-hour cycle}
              time2Suff:    CHAR;
              time3Suff:    CHAR;
              time4Suff:    CHAR;
              time5Suff:    CHAR;
              time6Suff:    CHAR;
              time7Suff:    CHAR;
              time8Suff:    CHAR;
              metricSys:    Byte;   {255 if metric, Ø if not}
              intlØVers:    INTEGER {version information}
            END;

Intl1Hndl = ^Intl1Ptr;
Intl1Ptr  = ^Intl1Rec;
Intl1Rec  = PACKED RECORD
              days:        ARRAY[1..7] OF STRING[15];  {day names}
              months:      ARRAY[1..12] OF STRING[15]; {month names}
              suppressDay: Byte;   {Ø for day name, 255 for none}
              longDateFmt: Byte;   {order of long date elements}
              dayleadingØ: Byte;   {255 for leading Ø in day number}
              abbrLen:     Byte;   {length for abbreviating names}
              stØ:         PACKED ARRAY[1..4] OF CHAR; {strings }
              st1:         PACKED ARRAY[1..4] OF CHAR; { for }
              st2:         PACKED ARRAY[1..4] OF CHAR; { long }
              st3:         PACKED ARRAY[1..4] OF CHAR; { date }
              st4:         PACKED ARRAY[1..4] OF CHAR; { format}
              intl1Vers:   INTEGER; {version information}
              localRtn:    INTEGER  {routine for localizing string }
                                    { comparison; actually may be }
                                    { longer than one integer}
            END;

DateForm = (shortDate, longDate, abbrevDate);
```

Routines

```
PROCEDURE IUDateString  (dateTime: LongInt; form: DateForm; VAR result:
                         Str255);
PROCEDURE IUDatePString (dateTime: LongInt; form: DateForm; VAR result:
                         Str255; intlParam: Handle);
PROCEDURE IUTimeString  (dateTime: LongInt; wantSeconds: BOOLEAN; VAR
                         result: Str255);
PROCEDURE IUTimePString (dateTime: LongInt; wantSeconds: BOOLEAN; VAR
                         result: Str255; intlParam: Handle);
FUNCTION  IUMetric :     BOOLEAN;
FUNCTION  IUGetIntl     (theID: INTEGER) : Handle;
PROCEDURE IUSetIntl     (refNum: INTEGER; theID: INTEGER; intlParam:
                         Handle);
FUNCTION  IUCompString   (aStr,bStr: Str255) : INTEGER;  [Pascal only]
FUNCTION  IUMagString    (aPtr,bPtr: Ptr; aLen,bLen: INTEGER) : INTEGER;
FUNCTION  IUEqualString  (aStr,bStr: Str255) : INTEGER;  [Pascal only]
FUNCTION  IUMagIDString  (aPtr,bPtr: Ptr; aLen,bLen: INTEGER) : INTEGER;
```

Assembly-Language Information

Constants

```
; Currency format

currSymLead     .EQU    4    ;set if currency symbol leads
currNegSym      .EQU    5    ;set if minus sign for negative
currTrailingZ   .EQU    6    ;set if trailing decimal zeroes
currLeadingZ    .EQU    7    ;set if leading integer zero

; Order of short date elements

mdy             .EQU    Ø    ;month day year
dmy             .EQU    1    ;day month year
ymd             .EQU    2    ;year month day

; Short date format

dayLeadingZ     .EQU    5    ;set if leading zero for day
mntLeadingZ     .EQU    6    ;set if leading zero for month
century         .EQU    7    ;set if century included

; Time format

secLeadingZ     .EQU    5    ;set if leading zero for seconds
minLeadingZ     .EQU    6    ;set if leading zero for minutes
hrLeadingZ      .EQU    7    ;set if leading zero for hours
```

```
; High-order byte of version information

verUS           .EQU    Ø
verFrance       .EQU    1
verBritain      .EQU    2
verGermany      .EQU    3
verItaly        .EQU    4

; Date form for IUDateString and IUDatePString

shortDate       .EQU    Ø    ;short form of date
longDate        .EQU    1    ;long form of date
abbrevDate      .EQU    2    ;abbreviated long form
```

International Resource Ø Data Structure

decimalPt	Decimal point character
thousSep	Thousands separator
listSep	List separator
currSym	Currency symbol
currFmt	Currency format
dateOrder	Order of short date elements
shortDateFmt	Short date format
dateSep	Date separator
timeCycle	Ø if 24-hour cycle, 255 if 12-hour
timeFmt	Time format
mornStr	Trailing string for first 12-hour cycle
eveStr	Trailing string for last 12-hour cycle
timeSep	Time separator
timeSuff	Trailing string for 24-hour cycle
metricSys	255 if metric, Ø if not
intlØVers	Version information

International Resource 1 Data Structure

days	Day names
months	Month names
suppressDay	Ø for day name, 255 for none
longDateFmt	Order of long date elements
dayleadingØ	255 for leading Ø in day number
abbrLen	Length for abbreviating names
stØ	Strings for long date format
st1	
st2	
st3	
st4	
intl1Vers	Version information
localRtn	Comparison localization routine

Routine Selectors

Routine	Selector
IUDatePString	14
IUDateString	∅
IUGetIntl	6
IUMagIDString	12
IUMagString	1∅
IUMetric	4
IUSetIntl	8
IUTimePString	16
IUTimeString	2

SUMMARY OF THE BINARY-DECIMAL CONVERSION PACKAGE

Routines

```
PROCEDURE NumToString (theNum: LongInt; VAR theString: Str255);
PROCEDURE StringToNum (theString: Str255; VAR theNum: LongInt);
```

Assembly-Language Information

Routine Selectors

Routine	Selector
NumToString	Ø
StringToNum	1

SUMMARY OF THE STANDARD FILE PACKAGE

Constants

```
CONST = putDlgID = -3999;  {SFPutFile dialog template ID}

        { Item numbers of enabled items in SFPutFile dialog }

        putSave   = 1;  {Save button}
        putCancel = 2;  {Cancel button}
        putEject  = 5;  {Eject button}
        putDrive  = 6;  {Drive button}
        putName   = 7;  {editText item for file name}

        getDlgID = -4ØØØ;  {SFGetFile dialog template ID}

        { Item numbers of enabled items in SFGetFile dialog }

        getOpen   = 1;  {Open button}
        getCancel = 3;  {Cancel button}
        getEject  = 5;  {Eject button}
        getDrive  = 6;  {Drive button}
        getNmList = 7;  {userItem for file name list}
        getScroll = 8;  {userItem for scroll bar}
```

Data Types

```
TYPE SFReply = RECORD
                 good:    BOOLEAN;     {FALSE if ignore command}
                 copy:    BOOLEAN;     {not used}
                 fType:   OSType;      {file type or not used}
                 vRefNum: INTEGER;     {volume reference number}
                 version: INTEGER;     {file's version number}
                 fName:   STRING[63]   {file name}
               END;

     SFTypeList = ARRAY [Ø..3] OF OSType;
```

Routines

```
PROCEDURE SFPutFile  (where: Point; prompt: Str255; origName: Str255;
                      dlgHook: ProcPtr; VAR reply: SFReply);
PROCEDURE SFPPutFile (where: Point; prompt: Str255; origName: Str255;
                      dlgHook: ProcPtr; VAR reply: SFReply; dlgID:
                      INTEGER; filterProc: ProcPtr);
PROCEDURE SFGetFile  (where: Point; prompt: Str255; fileFilter:
                      ProcPtr; numTypes: INTEGER; typeList: SFTypeList;
                      dlgHook: ProcPtr; VAR reply: SFReply);
```

```
PROCEDURE SFPGetFile (where: Point; prompt: Str255; fileFilter:
                      ProcPtr; numTypes: INTEGER; typeList: SFTypeList;
                      dlgHook: ProcPtr; VAR reply: SFReply; dlgID:
                      INTEGER; filterProc: ProcPtr);
```

DlgHook Function

```
FUNCTION MyDlg (item: INTEGER; theDialog: DialogPtr) : INTEGER;
```

FileFilter Function

```
FUNCTION MyFileFilter (paramBlock: ParmBlkPtr) : BOOLEAN;
```

Standard SFPutFile Items

Item number	Item	Standard display rectangle
1	Save button	(12, 74, 82, 92)
2	Cancel button	(114, 74, 184, 92)
3	Prompt string (statText)	(12, 12, 184, 28)
4	UserItem for disk name	(209, 16, 295, 34)
5	Eject button	(217, 43, 287, 61)
6	Drive button	(217, 74, 287, 92)
7	EditText item for file name	(14, 34, 182, 50)
8	UserItem for gray line	(200, 16, 201, 88)

Resource IDs of SFPutFile Alerts

Alert	Resource ID
Existing file	-3996
Locked disk	-3997
System error	-3995
Disk not found	-3994

Standard SFGetFile Items

Item number	Item	Standard display rectangle
1	Open button	(152, 28, 232, 46)
2	Invisible button	(1152, 59, 1232, 77)
3	Cancel button	(152, 90, 232, 108)
4	UserItem for disk name	(248, 28, 344, 46)
5	Eject button	(256, 59, 336, 77)
6	Drive button	(256, 90, 336, 108)
7	UserItem for file name list	(12, 11, 125, 125)
8	UserItem for scroll bar	(124, 11, 140, 125)
9	UserItem for gray line	(244, 20, 245, 116)
10	Invisible text (statText)	(1044, 20, 1145, 116)

Assembly-Language Information

Constants

```
putDlgID        .EQU     -3999  ;SFPutFile dialog template ID

; Item numbers of enabled items in SFPutFile dialog

putSave         .EQU      1     ;Save button
putCancel       .EQU      2     ;Cancel button
putEject        .EQU      5     ;Eject button
putDrive        .EQU      6     ;Drive button
putName         .EQU      7     ;editText item for file name

getDlgID        .EQU     -4ØØØ  ;SFGetFile dialog template ID

; Item numbers of enabled items in SFGetFile dialog

getOpen         .EQU      1     ;Open button
getCancel       .EQU      3     ;Cancel button
getEject        .EQU      5     ;Eject button
getDrive        .EQU      6     ;Drive button
getNmList       .EQU      7     ;userItem for file name list
getScroll       .EQU      8     ;userItem for scroll bar
```

Reply Record Data Structure

rGood	FALSE if ignore command
rType	File type
rVolume	Volume reference number
rVersion	File's version number
rName	File name

Routine Selectors

Routine	Selector
SFGetFile	2
SFPGetFile	4
SFPPutFile	3
SFPutFile	1

SUMMARY OF THE DISK INITIALIZATION PACKAGE

Routines

```
PROCEDURE DILoad;
PROCEDURE DIUnload;
FUNCTION  DIBadMount (where: Point; evtMessage: LongInt) : INTEGER;
FUNCTION  DIFormat   (drvNum: INTEGER) : OsErr;
FUNCTION  DIVerify   (drvNum: INTEGER) : OsErr;
FUNCTION  DIZero     (drvNum: INTEGER; volName: Str255) : OSErr;
```

Assembly-Language Information

Routine Selectors

Routine	Selector
DIBadMount	Ø
DIFormat	6
DILoad	2
DIUnload	4
DIVerify	8
DIZero	1Ø

Result Codes

Name	Value	Meaning
badMDBErr	-6Ø	Bad master directory block
extFSErr	-58	External file system
firstDskErr	-84	First of the range of low-level disk errors
ioErr	-36	Disk I/O error
lastDskErr	-64	Last of the range of low-level disk errors
mFulErr	-41	Memory full
noErr	Ø	No error
noMacDskErr	-57	Not a Macintosh disk
nsDrvErr	-56	No such drive
paramErr	-5Ø	Bad drive number
volOnLinErr	-55	Volume already on-line

GLOSSARY

ligature: A character that combines two letters.

list separator: The character that separates numbers, as when a list of numbers is entered by the user.

package: A set of data structures and routines that's stored as a resource and brought into memory only when needed.

routine selector: An integer that's pushed onto the stack before the _PackN macro is invoked, to identify which routine to execute. (N is the resource ID of a package; all macros for calling routines in the package expand to invoke _PackN.)

thousands separator: The character that separates every three digits to the left of the decimal point.

MACINTOSH USER EDUCATION

The Memory Manager: A Programmer's Guide /MEM.MGR/MEM

See Also: Inside Macintosh: A Road Map
Macintosh Memory Management: An Introduction
Programming Macintosh Applications in Assembly Language
The Resource Manager: A Programmer's Guide
The Segment Loader: A Programmer's Guide
Putting Together a Macintosh Application

Modification History:	First Draft (ROM 7)	Steve Chernicoff	1Ø/1Ø/83
	Second Draft	Bradley Hacker	1Ø/9/84

ABSTRACT

This manual describes the Memory Manager, the part of the Macintosh Operating System that controls the dynamic allocation of memory on the heap.

Summary of significant changes and additions since first draft:

- Important information about handle usage has been added (page 1Ø).

- The discussion of memory organization has been moved here (page 15) from the manual Programming Macintosh Applications in Assembly Language. It now includes a Lisa running MacWorks. All memory maps, or portions of them shown separately, place high memory at the top; other manuals will be changed to match this.

- The procedures MaxApplZone and MoreMasters have been added (page 3Ø).

- The descriptions of the routines InitZone, CompactMem, ResrvMem, and PurgeMem have been changed (pages 28 and 4Ø).

- Notes for assembly-language programmers are now brought up where appropriate rather than in a separate section at the end.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Memory Manager, the part of the Macintosh Operating System that controls the dynamic allocation of memory space on the heap. *** Eventually it will become part of the comprehensive Inside Macintosh manual. ***

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal and the information in the following manuals:

- Inside Macintosh: A Road Map
- Macintosh Memory Management: An Introduction
- Programming Macintosh Applications in Assembly Language, if you're using assembly language

ABOUT THE MEMORY MANAGER

Using the Memory Manager, your program can maintain one or more independent areas of heap memory (called heap zones) and use them to allocate blocks of memory of any desired size. Unlike stack space, which is always allocated and released in strict LIFO (last-in-first-out) order, blocks on the heap can be allocated and released in any order, according to your program's needs. So instead of growing and shrinking in an orderly way like the stack, the heap tends to become fragmented into a patchwork of allocated and free blocks, as shown in Figure 1. The Memory Manager does all the necessary "housekeeping" to keep track of the blocks as it allocates and releases them.

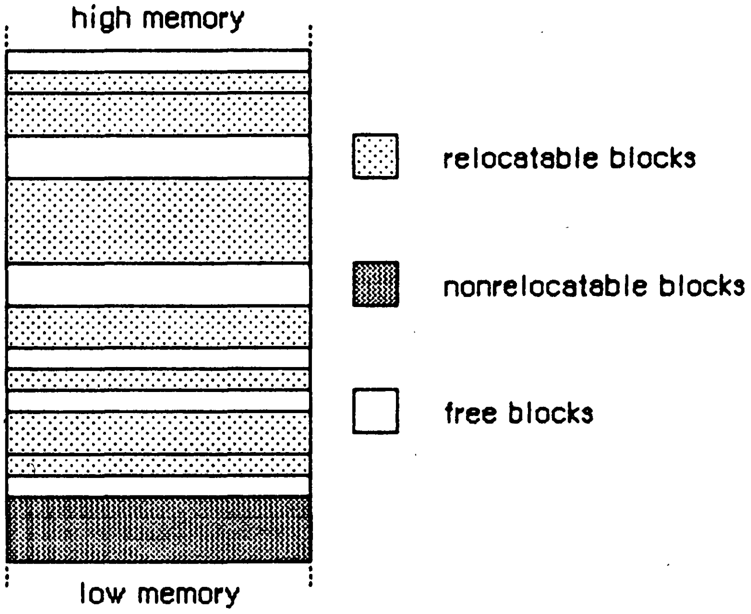


Figure 1. A Fragmented Heap

The Memory Manager always maintains at least two heap zones: a system heap zone, reserved for the system's own use, and an application heap zone for use by your program. The system heap zone is initialized to a fixed size when the system is started up (16.5K on a 128K Macintosh and 46K on a 512K Macintosh or a Lisa). Objects in this zone remain allocated even when one application terminates and another is started up. In contrast, the application heap zone is automatically reinitialized at the start of each new application program, and the contents of any previous application zone are lost.

Assembly-language note: If desired, you can prevent the application heap zone from being reinitialized when an application starts up; see the discussion of the Chain procedure in the Segment Loader manual for details.

The initial size of the application zone is 6K bytes, but it can grow as needed. Your program can create additional heap zones if it chooses, either by subdividing this original application zone or by allocating space on the stack for more heap zones.

(note)

In this manual, unless otherwise stated, the term "application heap zone" (or just "application zone") always refers to the original application heap zone provided by the system, before any subdivision.

Various parts of the Macintosh Operating System and Toolbox also use space in the application heap zone. For instance, your program's code typically resides in the application zone, in space reserved for it at the request of the Segment Loader. Similarly, the Resource Manager requests space in the application zone to hold resources it has read into memory from a resource file. Toolbox routines that create new entities of various kinds, such as NewWindow, NewControl, and NewMenu, also call the Memory Manager to allocate the space they need.

At any given time, there's one current heap zone, to which most Memory Manager operations implicitly apply. You can control which heap zone is current by calling a Memory Manager procedure. Whenever the system needs to access its own (system) heap zone, it saves the setting of the current heap zone and restores it later.

Space within a heap zone is divided up into contiguous pieces called blocks. The blocks in a zone fill it completely: every byte in the zone is part of exactly one block, which may be either allocated (reserved for use) or free (available for allocation). Each block has a block header for the Memory Manager's own use, followed by the block's contents, the area available for use by your application or the system (see Figure 2). There may also be some unused bytes at the end of the block, beyond the end of the contents. A block can be of any size, limited only by the size of the heap zone itself.

Assembly-language note: Blocks are always aligned on even word boundaries, so you can access them with word (.W) and long-word (.L) instructions.

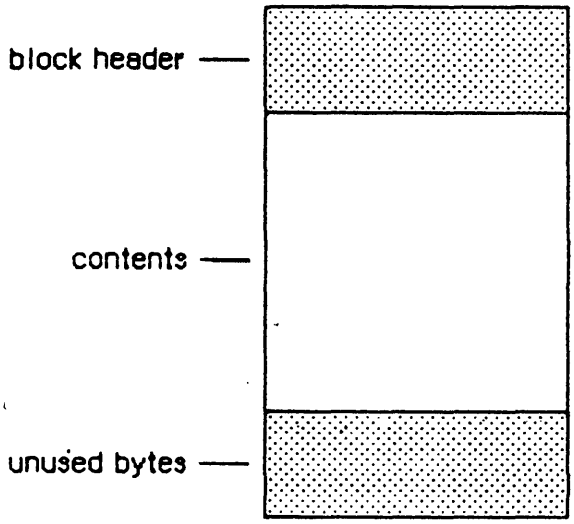


Figure 2. A Block

An allocated block may be relocatable or nonrelocatable. Relocatable blocks can be moved around within the heap zone to create space for other blocks; nonrelocatable blocks can never be moved. These are permanent properties of a block. If relocatable, a block may be locked or unlocked; if unlocked, it may be purgeable or unpurgeable. These attributes can be set and changed as necessary. Locking a relocatable block prevents it from being moved. Making a block purgeable allows the Memory Manager to remove it from the heap zone, if necessary, to make room for another block. (Purging of blocks is discussed further below under "How Heap Space Is Allocated".) A newly allocated relocatable block is initially unlocked and unpurgeable.

POINTERS AND HANDLES

Relocatable and nonrelocatable blocks are referred to in different ways: nonrelocatable blocks by pointers, relocatable blocks by handles. When the Memory Manager allocates a new block, it returns a pointer or handle to the contents of the block (not to the block's header) depending on whether the block is nonrelocatable (Figure 3) or relocatable (Figure 4).

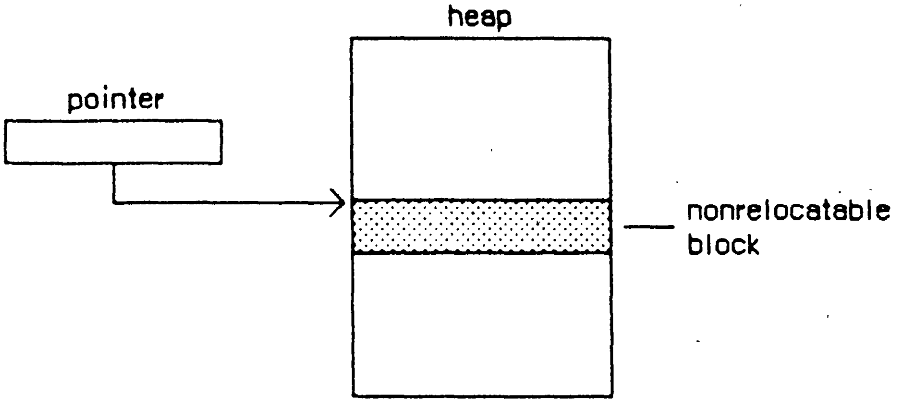


Figure 3. A Pointer to a Nonrelocatable Block

A pointer to a nonrelocatable block never changes, since the block itself can't move. A pointer to a relocatable block can change value, however, since the block can move. For this reason, the Memory Manager maintains a single nonrelocatable master pointer to each relocatable block. The master pointer is created at the same time as the block and set to point to it. When you allocate a relocatable block, the Memory Manager returns a pointer to the master pointer, called a handle to the block (see Figure 4). If the Memory Manager later has to move the block, it has only to update the master pointer to point to the block's new location.

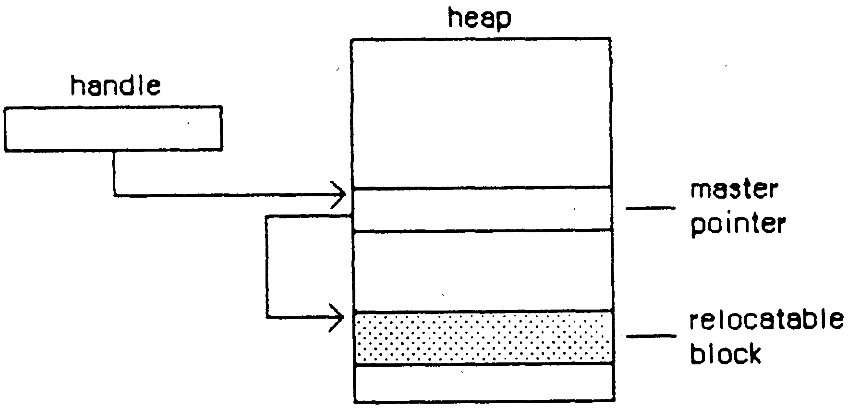


Figure 4. A Handle to a Relocatable Block

HOW HEAP SPACE IS ALLOCATED

The Memory Manager allocates space for relocatable blocks according to a "first fit" strategy. It looks for a free block of at least the requested size, scanning forward from the end of the last block allocated and "wrapping around" from the end of the zone to the beginning if necessary. As soon as it finds a free block big enough, it allocates the requested number of bytes from that block.

If a single free block can't be found that's big enough, the Memory Manager compacts the heap zone: moves allocated blocks together in order to collect the free space into a single larger free block. Only relocatable, unlocked blocks are moved. The compaction continues until either a free block of at least the requested size has been created or the entire heap zone has been compacted. Figure 5 illustrates what happens when the entire heap must be compacted to create a large enough free block.

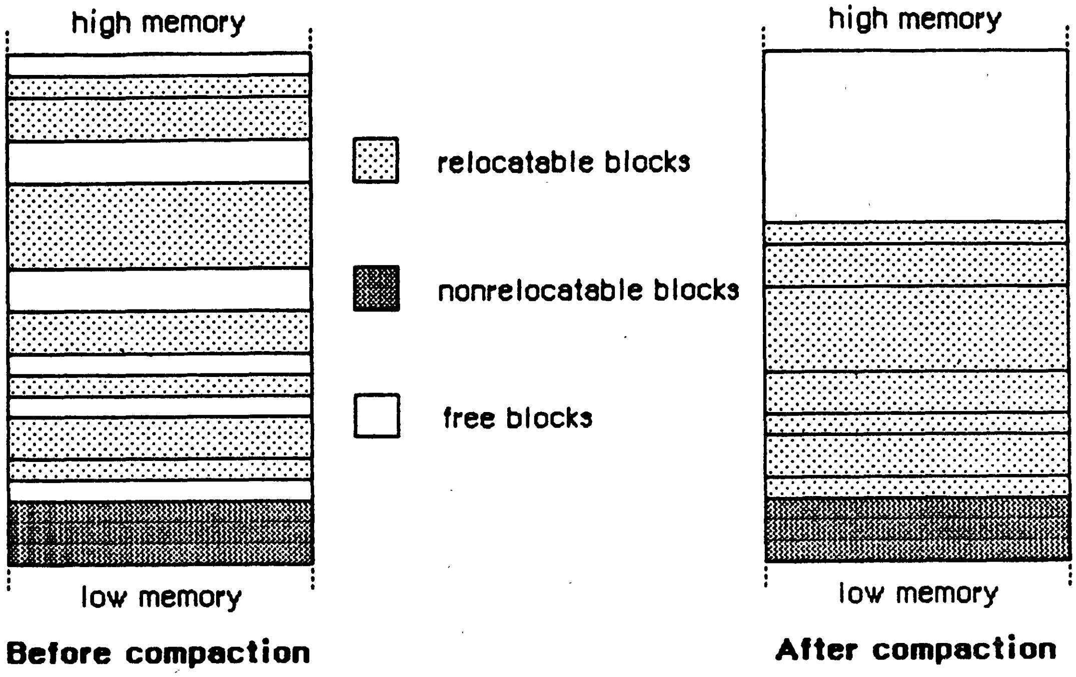


Figure 5. Heap Compaction

Notice that nonrelocatable blocks (and relocatable ones that are temporarily locked) interfere with the compaction process by forming immovable "islands" in the heap. This can prevent free blocks from being collected together and lead to fragmentation of the available free space, as shown in Figure 6. To minimize this problem, the Memory Manager tries to keep all the nonrelocatable blocks together at the bottom of the heap zone. When you allocate a nonrelocatable block, the Memory Manager will try to make room for the new block near the bottom of the zone, by moving other blocks upward, expanding the zone, or purging blocks from it (see below).

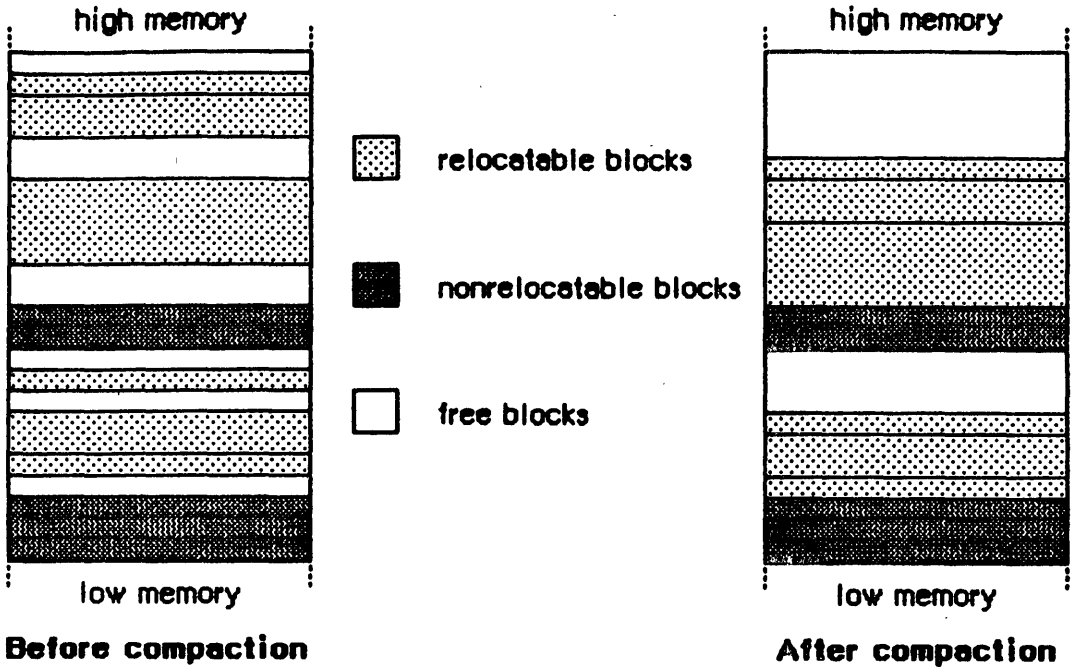


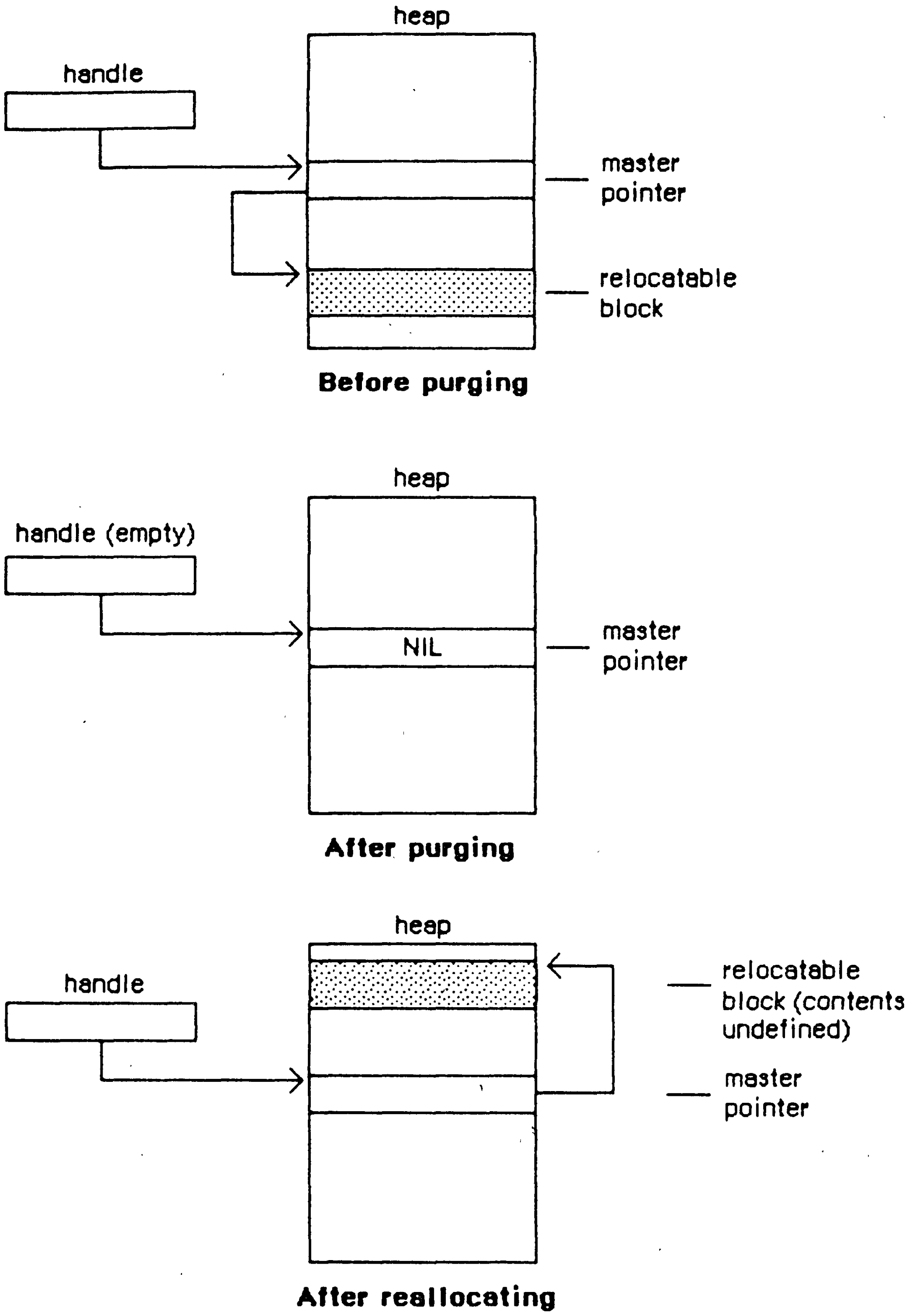
Figure 6. Fragmentation of Free Space

(warning)

Whenever possible, use relocatable instead of nonrelocatable blocks. If you must use nonrelocatable blocks, allocate them early in the program so they will be placed near the bottom of the heap.

If the Memory Manager can't satisfy the allocation request after compacting the entire heap zone, it next tries expanding the zone by the requested number of bytes (rounded up to the nearest 1K bytes). Only the original application zone can be expanded, and only up to a certain limit (discussed more fully under "The Stack and the Heap", below). If any other zone is current, or if the application zone has already reached or exceeded its limit, this step is skipped.

Next the Memory Manager tries to free space by purging blocks from the zone. Only relocatable blocks can be purged, and then only if they're explicitly marked as unlocked and purgeable. Purging a block removes it from its heap zone and frees the space it occupies. The space occupied by the block's master pointer itself remains allocated, but the master pointer is set to NIL. Any handles to the block now point to a NIL master pointer, and are said to be empty. If your program later needs to refer to the purged block, it must detect that the handle has become empty and ask the Memory Manager to reallocate the block. This operation updates the master pointer (see Figure 7).



Before purging

After purging

After reallocating

Figure 7. Purging and Reallocating a Block

(warning)

> Reallocating a block recovers only its space, not its contents (which were lost when the block was purged). It's up to your program to reconstitute the block's contents.

Finally, if all else fails, the Memory Manager calls the grow zone function, if any, for the current heap zone. This is an optional routine that an application can provide to take any last-ditch measures to try to "grow" the zone by freeing some space in it. The grow zone function can try to create additional free space by purging blocks that were previously marked unpurgeable, unlocking previously locked blocks, and so on. The Memory Manager will call the grow zone function repeatedly, compacting the heap again after each call, until either it finds the space it's looking for or the grow zone function has exhausted all possibilities. In the latter case, the Memory Manager will finally give up and report that it's unable to satisfy the allocation request.

Dereferencing a Handle

Accessing a block by double indirection, through its handle instead of through its master pointer, requires an extra memory reference. For efficiency, you may sometimes want to dereference the handle--that is, make a copy of the block's master pointer, and then use that pointer to access the block by single indirection. But **be careful!** Any operation that allocates space from the heap may cause the underlying block to be moved or purged. In that event, the master pointer itself will be correctly updated, but your copy of it will be left dangling.

One way to avoid this common type of program bug is to lock the block before dereferencing its handle. For example:

```
VAR aPointer: Ptr;
    aHandle: Handle;

BEGIN
. . . ;
aHandle := NewHandle( . . . );    {create a relocatable block}
. . . ;
HLock(aHandle);                   {lock before dereferencing}
aPointer := aHandle^;             {dereference handle}
WHILE . . . DO
  BEGIN
  ...aPointer^...                 {use simple pointer}
  END;
HUnlock(aHandle);                 {unlock block when finished}
. . .
END
```

Assembly-language note: To dereference a handle in assembly language, just copy the master pointer into an address register and use it to access the block by single indirection.

Remember, however, that when you lock a block it becomes an "island" in the heap that may interfere with compaction and cause free space to become fragmented. It's recommended that you use this technique only in parts of your program where efficiency is critical, such as inside tight inner loops that are executed many times (and that don't allocate other blocks).

(warning)

Don't forget to unlock the block again when you're through with the dereferenced handle.

Instead of locking the block, you can update your copy of the master pointer after any "dangerous" operation (one that can invalidate the pointer by moving or purging the block it points to). Memory Manager routines that can move or purge blocks in the heap are NewHandle, NewPtr, SetHandleSize, SetPtrSize, ReallocHandle, ResrvMem, CompactMem, PurgeMem, and MaxMem. Since these routines can be called indirectly from other Operating System or Toolbox routines, you should assume that any call to the Operating System or Toolbox can potentially leave your dereferenced pointer dangling.

The Pascal compiler frequently dereferences handles during its normal operation. You should take care to write code that will not require the compiler to deference handles in the following cases:

- Use of the WITH statement with a handle, such as

  ```
  WITH aHandle^^ DO . . .
  ```

- Assigning the result of a function that can move or purge blocks to a field in a record referred to by a handle, such as

  ```
  aHandle^^.field := NewHandle(...)
  ```

  A problem may arise because the compiler generates code that dereferences the handle before calling NewHandle--and NewHandle may move the block containing the field.

- Passing an argument of more than four bytes referred to by a handle, to a routine that can move or purge a block or to any routine in a package or another segment. For example:

  ```
  TEUpdate(aHandle^^.box)
  ```

  or

```
        DrawString(aHandle^^.msg)
```

You can avoid having the compiler generate and use dangling pointers by locking a block before you use its handle in the above situations. Or, you can use temporary variables, as in the following:

```
     temp := NewHandle(...);
     aHandle^^.field := temp
```

THE STACK AND THE HEAP

The LIFO (last-in-first-out) nature of the stack makes it particularly convenient for memory allocation connected with the activation and deactivation of routines (procedures and functions). Each time a routine is called, space is allocated for a stack frame. The stack frame holds the routine's parameters, local variables, and return address. Upon exit from the routine, the stack frame is released, restoring the stack to the same state it was in when the routine was called.

In Pascal, all stack management is done by the compiler. When you call a routine, the compiler generates code to reserve space if necessary for a function result, place the parameter values and return link on the stack, and jump to the routine. The routine can then allocate space on the stack for its own local variables.

Before returning, the routine releases the stack space occupied by its local variables, return link, and parameters. If the routine is a function, it leaves its result on the stack for the calling program.

> Assembly-language note: In assembly language, you control the allocation and release of stack space explicitly by manipulating the stack pointer (register A7, also referred to by the standard symbol SP). Decreasing the stack pointer allocates stack space; increasing it releases stack space. Certain machine instructions --notably JSR (Jump to Subroutine), BSR (Branch to Subroutine), and RTS (Return from Subroutine)--also implicitly manipulate the stack pointer.

The application heap zone and the application stack share the same area in memory, growing toward each other from opposite ends (see Figure 8). Naturally it would be disastrous for either to grow so far that it collides with the other. To help prevent such collisions, the Memory Manager enforces a limit on how far the application heap zone can grow toward the stack. Your program can set this application heap limit to control the allotment of available space between the stack and the heap.

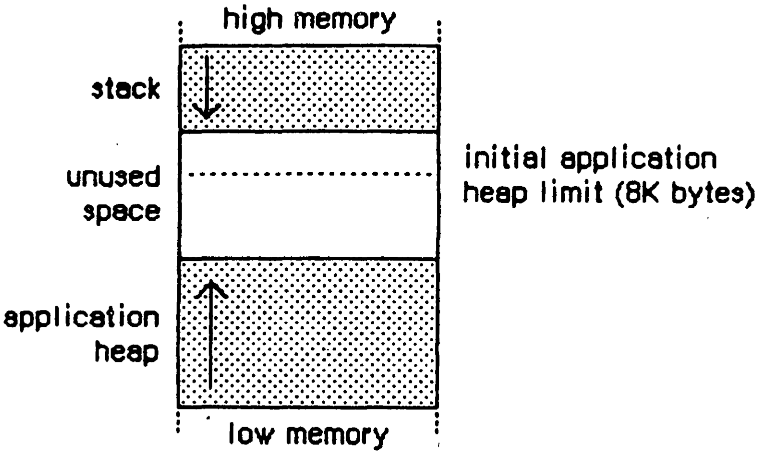


Figure 8. The Stack and the Heap

The application heap limit marks the boundary between the space available for the application heap zone and the space reserved exclusively for the stack. At the start of each application program, the limit is initialized to allow 8K bytes for the stack. Depending on your program's needs, you can adjust the limit to allow more heap space at the expense of the stack or vice versa.

Notice that the limit applies only to expansion of the **heap**; it has no effect on how far the **stack** can expand. Although the heap can never expand beyond the limit into space reserved for the stack, there's nothing to prevent the stack from crossing the limit. It's up to you to set the limit low enough to allow for the maximum stack depth your program will ever need.

(note)

> Regardless of the limit setting, the application zone is never allowed to grow to within 1K of the current end of the stack. This gives a little extra protection in case the stack is approaching the boundary or has crossed over onto the heap's side, and allows some safety margin for the stack to expand even further.

To help detect collisions between the stack and the heap, a "stack sniffer" routine is run sixty times a second, during the Macintosh's vertical retrace interrupt. This routine compares the current ends of the stack and the heap and invokes the System Error Handler in case of a collision.

Assembly-language note: The System Error Handler moves the top long word off the stack, sets the top of the stack to the bottom of the stack, and then restores the top long word.

The stack sniffer can't prevent collisions, only detect them after the fact: a lot of computation can take place in a sixtieth of a second. In fact, the stack can easily expand into the heap, overwrite it, and then shrink back again before the next activation of the stack sniffer, escaping detection completely. The stack sniffer is useful mainly during software development; the alert box the System Error Handler displays can be confusing to your program's end user. Its purpose is to warn you, the programmer, that your program's stack and heap are colliding, so that you can adjust the heap limit to correct the problem before the user ever encounters it.

> Assembly-language note: A number of global variables and constants control the size of the heap and stack. The initial and minimum sizes of the application heap are given by the global constants appZoneSize and minZone, respectively. The default size of the stack is given by the global constant dfltStackSize; it's moved into the global variable DefltStack when the system starts up. The minimum size of the stack is specified by the global constant mnStackSize; it's moved into the global variable MinStack when the system starts up.

GENERAL-PURPOSE DATA TYPES

The Memory Manager includes a number of type definitions for general-purpose use. The types listed below are explained in Macintosh Memory Management: An Introduction.

```
TYPE SignedByte = -128..127;
     Byte       = Ø..255;
     Ptr        = ^SignedByte;
     Handle     = ^Ptr;

     Str255       = STRING[255];
     StringPtr    = ^Str255;
     StringHandle = ^StringPtr;

     ProcPtr = Ptr;

     Fixed = LONGINT;
```

*** (Correction to be made to the Memory Management Introduction manual: Bit 15 of the high-order word of a fixed-point number is the sign bit.) ***

For specifying the sizes of blocks on the heap, the Memory Manager defines a special type called Size:

```
TYPE Size = LONGINT;
```

All Memory Manager routines that deal with block sizes expect parameters of type Size or return them as results. To specify a size bigger than any existing block, you can use the following constant:

```
CONST maxSize = $800000;
```

MEMORY ORGANIZATION

This section discusses the organization of the Macintosh memory and Lisa memory when running MacWorks. You'll need this information if you want to use the available memory efficiently.

The organization of the Macintosh RAM is shown in Figure 9 on the following page. The variable names listed on the right in the figure refer to global variables for use by assembly-language programmers.

> Assembly-language note: The global variables not shown in parentheses are constants that are equated directly to a memory address; those in parentheses are variables containing long-word pointers that in turn point to an address. Names identified as marking the end of an area actually refer to the address **following** the last byte in that area.

The lowest 2816 bytes are used for system globals and the trap dispatch table. Immediately following this are the system heap and the application space. The application space is the memory available for dynamic allocation by applications. Most of the application 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 remainder of the application space is occupied by global variables belonging to QuickDraw, the application's global variables, parameters passed to the application by the Finder, and the jump table. All of these are explained in the Segment Loader manual.

> Assembly-language note: The starting address and default size of the system heap are given by the global constants heapStart and sysZoneSize, respectively.

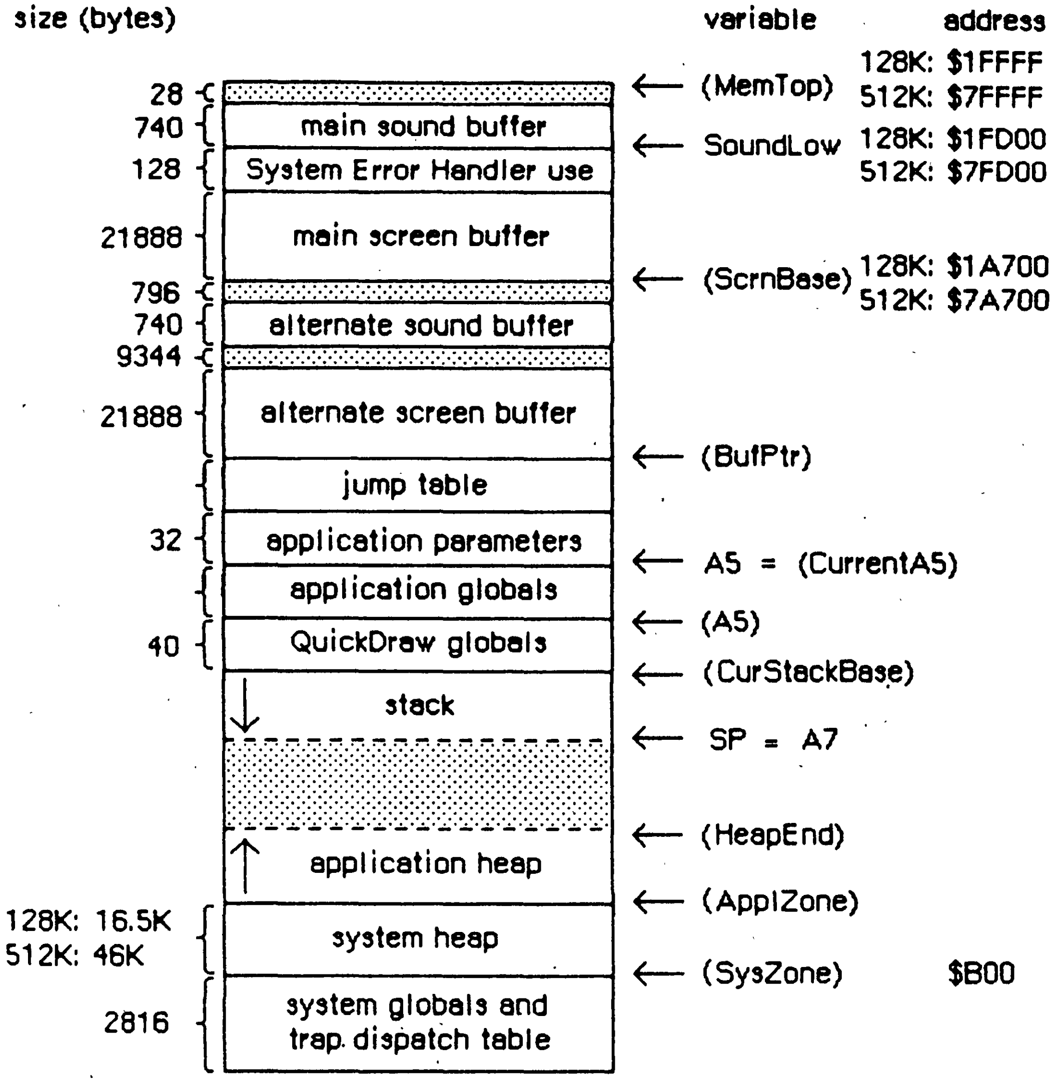


Figure 9. Macintosh RAM Organization

At (almost) the very end of memory are the main sound buffer, used by the Sound Driver to control the sounds emitted by the built-in speaker and the Disk Driver to control disk motor speed, and the main screen buffer, which holds the bit image to be displayed on the Macintosh screen. The area between the main screen and sound buffers is used by the System Error Handler. Note that the addresses of these buffers are different for different-sized computers.

There are alternate screen and sound buffers for special applications. If you use either or both of these, the space available for use by your application is reduced accordingly. The Segment Loader provides a routine for specifying that an alternate screen or sound buffer will be used.

The memory organization of a Lisa running MacWorks is shown in Figure 1Ø.

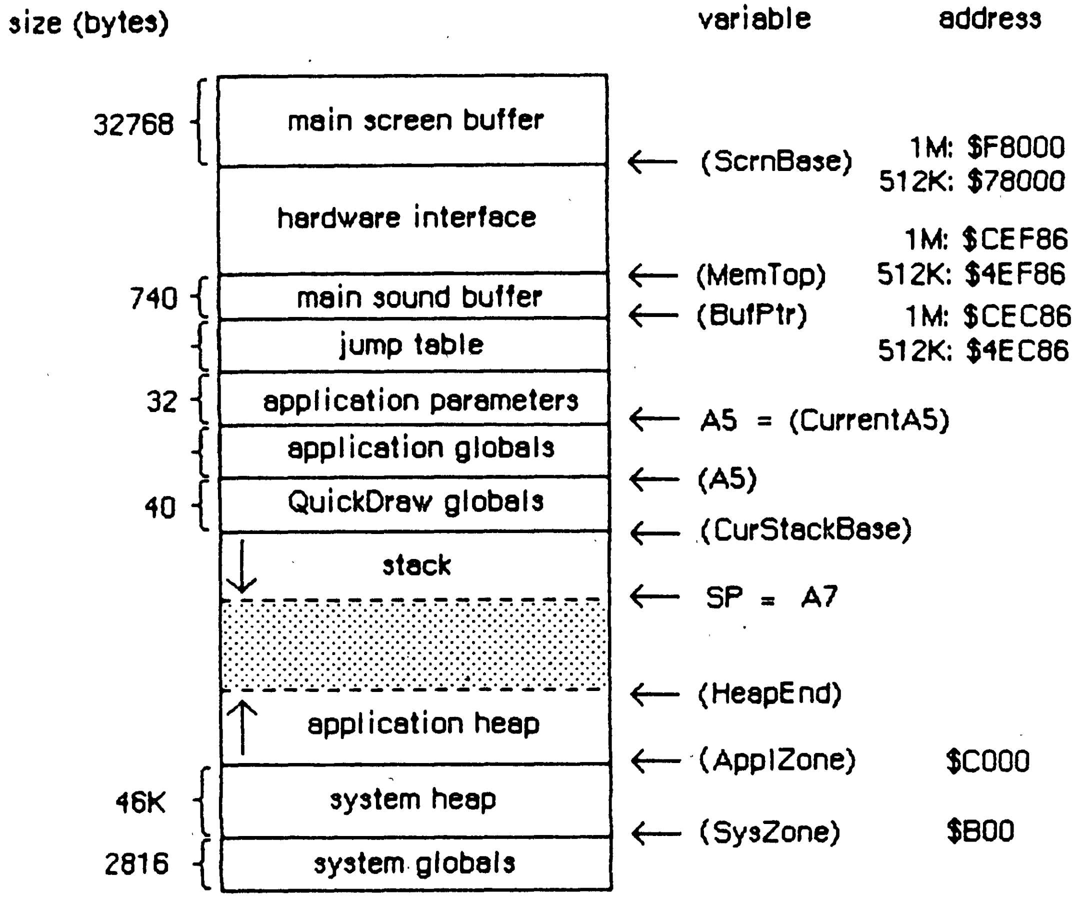


Figure 1Ø. Lisa RAM Organization

MEMORY MANAGER DATA STRUCTURES

This section discusses the internal data structures of the Memory Manager. You don't need to know this information if you're just using the Memory Manager routinely to allocate and release blocks of memory from the application heap zone.

Structure of Heap Zones

Each heap zone begins with a 52-byte zone header and ends with a 12-byte zone trailer (see Figure 11). The header contains all the information the Memory Manager needs about that heap zone; the trailer is just a minimum-size free block (described in the next section) placed at the end of the zone as a marker. All the remaining space between the header and trailer is available for allocation.

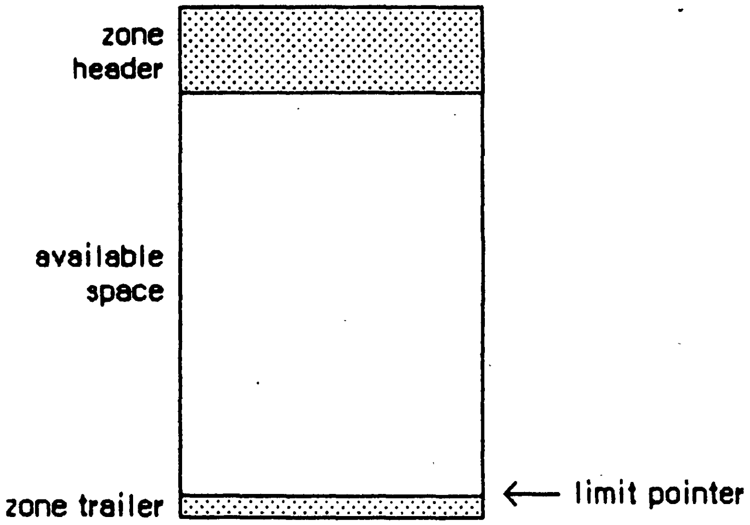


Figure 11. Structure of a Heap Zone

In Pascal, a heap zone is defined as a zone record of type Zone. It's always referred to with a zone pointer of type THz ("the heap zone"):

```
       TYPE THz  = ^Zone;

            Zone = RECORD
                     bkLim:       Ptr;      {limit pointer}
                     purgePtr:    Ptr;      {used internally}
                     hFstFree:    Ptr;      {first free master pointer}
                     zcbFree:     LONGINT;  {number of free bytes}
                     gzProc:      ProcPtr;  {grow zone function}
                     moreMast:    INTEGER;  {master pointers to allocate}
                     flags:       INTEGER;  {used internally}
                     cntRel:      INTEGER;  {relocatable blocks}
                     maxRel:      INTEGER;  {maximum cntRel value}
                     cntNRel:     INTEGER;  {nonrelocatable blocks}
                     maxNRel:     INTEGER;  {maximum maxRel value}
                     cntEmpty:    INTEGER;  {empty master pointers}
                     cntHandles:  INTEGER;  {total master pointers}
                     minCBFree:   LONGINT;  {minimum zcbFree value}
                     purgeProc:   ProcPtr;  {purge warning procedure}
                     sparePtr:    Ptr;      {used internally}
                     allocPtr:    Ptr;      {used internally}
                     heapData:    INTEGER   {first usable byte in zone}
                   END;
```

(warning)

> The fields of the zone header are for the Memory Manager's own internal use. You can examine the contents of the zone's fields, but in general it doesn't make sense for your program to try to change them. The few exceptions are noted below in the discussions of the specific fields.

BkLim is a pointer to the zone's trailer block. Since the trailer is the last block in the zone, this constitutes a limit pointer to the byte **following** the last byte of usable space in the zone.

PurgePtr and allocPtr are "roving pointers" that the Memory Manager maintains for its own internal use. When scanning the zone for a free block to satisfy an allocation request for a relocatable block, the Memory Manager begins at the block pointed to by allocPtr. When purging blocks from the zone, it starts from the block pointed to by purgePtr. Both pointers are advanced with each operation.

HFstFree is a pointer to the first free master pointer in the zone. Instead of just allocating space for one master pointer each time a relocatable block is created, the Memory Manager "preallocates" several master pointers at a time, themselves forming a nonrelocatable block. The moreMast field of the zone record tells the Memory Manager how many master pointers at a time to preallocate for this zone. Master pointers for the system heap zone are allocated 32 at a time; for the application zone, 64 at a time. For other heap zones, you specify the value of moreMast when you create the zone.

Assembly-language note: The default number of master pointers in the system and application heap zones is determined by the global constant dfltMasters. The number in the system heap zone is equal to dfltMasters, and the number in the application heap zone is equal to twice dfltMasters. The global constant maxMasters specifies the maximum number of master pointers in a heap zone.

All master pointers that are allocated but not currently in use are linked together into a list beginning in the hFstFree field. When you allocate a new relocatable block, the Memory Manager removes the first available master pointer from this list, sets it to point to the new block, and returns its address to you as a handle to the block. (If the list is empty, it allocates a fresh block of moreMast master pointers.) When you release a relocatable block, its master pointer isn't released, but is linked onto the beginning of the list to be reused. Thus the amount of space devoted to master pointers can increase, but can never decrease until the zone is reinitialized.

The zcbFree field always contains the number of free bytes remaining in the zone. As blocks are allocated and released, the Memory Manager adjusts zcbFree accordingly. This number represents an upper limit on the size of block you can allocate from this heap zone.

(warning)

> It may not actually be possible to allocate a block as big as zcbFree bytes. Because nonrelocatable and locked blocks can't be moved, it isn't always possible to

collect all the free space into a single block by compaction.

The gzProc field is a pointer to the zone's grow zone function, or NIL if there is none. You supply this pointer when you create a new heap zone and can change it at any time later. The system and application heap zones initially have no grow zone function.

CntRel, maxRel, cntNRel, maxNRel, cntEmpty, cntHandles, and minCBFree are not used by the ROM-based version of the Memory Manager. *** These fields are reserved for eventual use by a special RAM-based version that will gather statistics on a program's memory usage within each heap zone. CntRel and cntNRel will count the number of relocatable and nonrelocatable blocks currently allocated within the zone. MaxRel and maxNRel will record the "historical maximum" values attained by cntRel and cntNRel since the program was started. CntEmpty will count the current number of empty master pointers, and cntHandles the total number of master pointers currently allocated. MinCBFree will record the historical minimum number of free bytes in the zone. ***

PurgeProc is a pointer to the zone's purge warning procedure, or NIL if there is none. The Memory Manager will call this procedure before it purges a block from the zone. If you want to install your own purge warning procedure, you have to be very careful not to interfere with the one the Resource Manager may have installed; for further details, see the Resource Manager manual and "Grow Zone Operations" in the "Memory Manager Routines" section below.

The last field of a zone record, heapData, is a dummy field marking the beginning of the zone's usable memory space. HeapData nominally contains an integer, but this integer has no significance in itself--it's just the first two bytes in the block header of the first block in the zone. The purpose of the heapData field is to give you a way of locating the effective beginning of the zone. For example, if myZone is a zone pointer, then

```
@(myZone^.heapData)
```

is a pointer to the first usable byte in the zone, just as

```
myZone^.bkLim
```

is a limit pointer to the byte following the last usable byte in the zone.

Structure of Blocks

Every block in a heap zone, whether allocated or free, has a block header that the Memory Manager uses to find its way around in the zone. Block headers are completely transparent to your program. All pointers and handles to allocated blocks point to the beginning of the block's contents, following the end of the header. Similarly, all block sizes seen by your program refer to the block's logical size (the number of bytes in its contents) rather than its physical size (the number of bytes it actually occupies in memory, including the header and any unused bytes at the end of the block).

Since your program shouldn't normally have to deal with block headers directly, there's no Pascal record type defining their structure. A block header consists of eight bytes, as shown in Figure 12.

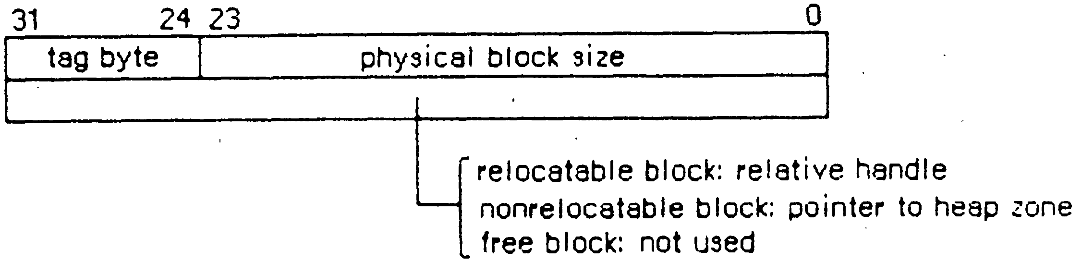

31
24
23
0
tag byte
physical block size
relocatable block: relative handle
nonrelocatable block: pointer to heap zone
free block: not used

Figure 12. Block Header

The first byte of the block header is the tag byte, discussed below. The next three bytes contain the block's physical size in bytes. Adding this number to the block's address gives the address of the next block in the zone.

Assembly-language note: You can use the global constants tagMask and bcMask to determine the value of the tag byte and the block's physical size, respectively.

The contents of the second long word (four bytes) in the block header depend on the type of block. For relocatable blocks, it contains the block's relative hardle: a pointer to the block's master pointer, expressed as an offset relative to the start of the heap zone rather than as an absolute memory address. Adding the relative handle to the zone pointer produces a true handle for this block. For nonrelocatable blocks, the second long word of the header is just a pointer to the block's zone. For free blocks, these four bytes are unused.

The structure of a tag byte is shown in Figure 13.

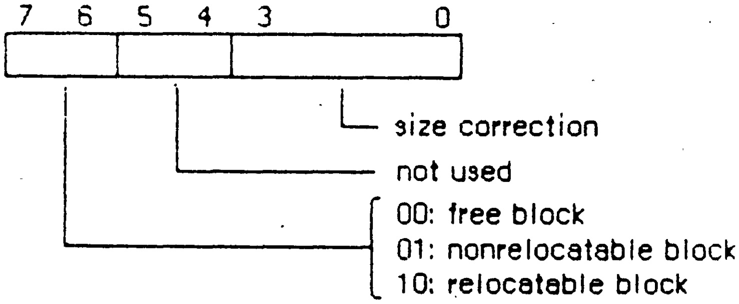


Figure 13. Tag Byte

Assembly-language note: You can use the global constants tyBkFree, tyBkNRel, and tyBkRel to test whether the value of the tag byte indicates a free, nonrelocatable, or relocatable block, respectively. Alternatively, you can use the global constants freeTag, nRelTag, and relTag as masks to determine the value of the tag byte.

The "size correction" in the tag byte of a block header is the number of unused bytes at the end of the block, beyond the end of the block's contents. It's equal to the difference between the block's logical and physical sizes, excluding the eight bytes of overhead for the block header:

logicalSize = physicalSize - sizeCorrection - 8

Assembly-language note: You can use the global constant bcOffMask to determine the size correction of a block.

There are two reasons why a block may contain such unused bytes:

- The Memory Manager allocates space only in even numbers of bytes. If the block's logical size is odd, an extra, unused byte is added at the end to keep the physical size even.

- The minimum number of bytes in a block is 12. This minimum applies to all blocks, free as well as allocated. If allocating the required number of bytes from a free block would leave a fragment of fewer than 12 free bytes, the leftover bytes are included unused at the end of the newly allocated block instead of being returned to free storage.

Structure of Master Pointers

The master pointer to a relocatable block has the structure shown in Figure 14. The low-order three bytes of the long word contain the address of the block's contents. The high-order byte contains some flag bits that specify the block's current status. Bit 7 of this byte is the lock bit (1 if the block is locked, Ø if it's unlocked); bit 6 is the purge bit (1 if the block is purgeable, Ø if it's unpurgeable). Bit 5 is used by the Resource Manager to identify blocks containing resource information; such blocks are marked by a 1 in this bit.

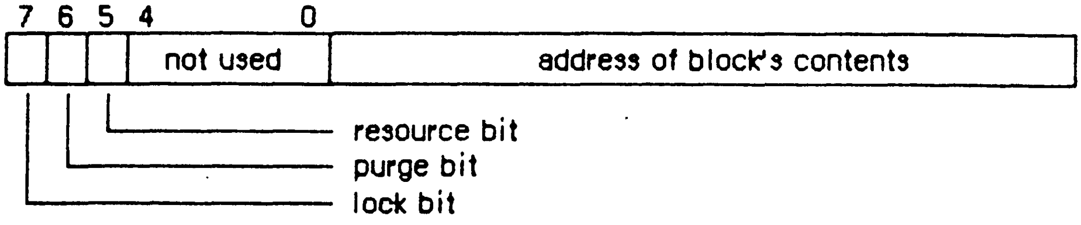


Figure 14. Structure of a Master Pointer

(warning)
Note that the flag bits in the high-order byte have numerical significance in any operation performed on a master pointer. For example, the lock bit is also the sign bit.

Assembly-language note: You can use the mask in the global variable Lo3Bytes to determine the value of the low-order three bytes of a master pointer. To determine the value of bits 5, 6, and 7, you can use the global constants resource, purge, and lock, respectively.

USING THE MEMORY MANAGER

This section discusses how the Memory 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.

There's ordinarily no need to initialize the Memory Manager before using it. The system heap zone is automatically initialized each time the system is started up, and the application heap zone each time an application program is started up. In the unlikely event that you need to reinitialize the application zone while your program is running, you can use InitApplZone.

When your application starts up it should allocate the memory it requires in the most space-efficient manner possible. The main segment of your program should call the MaxApplZone procedure, which expands the application heap zone to its limit. Then call the procedure MoreMasters to allocate as many blocks of master pointers as your application and any desk accessories will need. Next initialize QuickDraw and the Window Manager (if you're going to use it). These last two steps ensure that most of the nonrelocatable blocks you'll need are packed together at the bottom of the heap.

To allocate a new relocatable block, use NewHandle; for a nonrelocatable block, use NewPtr. These functions return a handle or a pointer, as the case may be, to the newly allocated block. To release a block when you're finished with it, use DisposHandle or DisposPtr.

(warning)

> Don't use the Pascal standard procedures NEW and DISPOSE, because they don't use the Memory Manager.
> *** Eventually these routines will be changed to work through the Memory Manager. ***

You can also change the size of an already allocated block with SetHandleSize or SetPtrSize, and find out its current size with GetHandleSize or GetPtrSize. Use HLock and HUnlock to lock and unlock relocatable blocks.

(note)

> In general, you should use relocatable blocks whenever possible, to avoid unnecessary fragmentation of free space. Use nonrelocatable blocks only for things like I/O buffers, queues, and other objects that must have a fixed location in memory.

(note)

> If you must lock a relocatable block, unlock it at the earliest possible opportunity. Before allocating a block that you know will be locked for long periods of time, call ResrvMem to make room for the block as near as possible to the beginning of the zone.

In some situations it may be desirable to determine the handle that points to a given master pointer. To do this you can call the RecoverHandle function. For example, a relocatable block of code might want to find out the handle that refers to it, so it can lock itself down in the heap.

Ordinarily, you shouldn't have to worry about compacting the heap or purging blocks from it; the Memory Manager automatically takes care of these chores for you. You can control which blocks are purgeable with HPurge and HNoPurge. If for some reason you want to compact or purge the heap explicitly, you can do so with CompactMem or PurgeMem. To explicitly purge a specific block, use EmptyHandle.

(warning)

> Before attempting to access any purgeable block, you must check its handle to make sure the block is still allocated. If the handle is empty (that is, if h^ = NIL, where h is the handle), then the block has been purged; before accessing it, you have to reallocate it by calling ReallocHandle, and then recreate its contents. (If it's a resource block, just call the Resource Manager procedure LoadResource; it checks the handle and reads

the resource into memory if it's not already in memory.)

You can find out how much free space is left in a heap zone by calling FreeMem (to get the total number of free bytes) or MaxMem (to get the size of the largest single free block and the maximum amount by which the zone can grow). Beware: MaxMem compacts the entire zone and purges all purgeable blocks. To limit the growth of the application zone, use SetApplLimit; to install a grow zone function to help the Memory Manager allocate space in a zone, use SetGrowZone.

You can create additional heap zones for your program's own use, either within the original application zone or in the stack, with InitZone. If you do maintain more than one heap zone, you can find out which zone is current at any given time with GetZone and switch from one to another with SetZone. Almost all Memory Manager operations implicitly apply to the current heap zone. To refer to the system heap zone or the (original) application heap zone, use the Memory Manager function SystemZone or ApplicZone. To find out which zone a particular block resides in, use HandleZone (if the block is relocatable) or PtrZone (if it's nonrelocatable).

(note)

> Most applications will just use the original application heap zone and never have to worry about which zone is current.

After calling any Memory Manager routine, you can determine whether it was successfully completed or failed, by calling MemError.

Assembly-language note: Code that will be executed via an interrupt can't use the Memory Manager, because an interrupt can occur unpredictably at any time; in particular, it can occur while the Memory Manager is in the middle of an operation, when the heap is inconsistent.

MEMORY MANAGER ROUTINES

This section describes all the Memory Manager procedures and functions. Each routine is presented first in its Pascal form. For most routines, this is followed by a box containing information needed to use the routine from assembly language; Pascal programmers can just skip this box.

In addition to their normal results, many Memory Manager routines yield a result code that you can examine by calling the MemError function. The description of each routine includes a list of all result codes it can yield.

Assembly-language note: When called from assembly language, not all Memory Manager routines return a result code. Those that do always leave it as a word-length quantity in the low-order word of register DØ on return from the trap. However, some routines leave something else there instead: see the descriptions of individual routines for details. Just before returning, the trap dispatcher tests the lower word of DØ with a TST.W instruction, so that on return from the trap the condition codes reflect the status of the result code, if any.

The stack-based interface routines called from Pascal always produce a result code. If the underlying trap doesn't return one, the interface routine "manufactures" a result code of noErr and stores it where it can later be accessed with MemError.

Assembly-language note: You can specify that some Memory Manager routines apply to the system heap zone instead of the current zone by setting bit 1Ø of the routine trap word. You do this by supplying the word SYS (uppercase) as the second argument to the routine macro:

```
        _FreeMem ,SYS
```

If you want a block of memory to be cleared to zeros when it's allocated by a NewPtr or NewHandle call, set bit 9 of the routine trap word. You can do this by supplying the word CLEAR (uppercase) as the second argument to the routine macro:

```
        _NewHandle ,CLEAR
```

You can combine SYS and CLEAR in the same macro call, but SYS must come first:

```
        _NewHandle ,SYS,CLEAR
```

The description of each routine lists whether SYS or CLEAR are applicable.

Initialization and Allocation

PROCEDURE InitApplZone;

Trap macro	_InitApplZone
On exit	DØ: result code (integer)

InitApplZone initializes the application heap zone and makes it the current zone. The contents of any previous application zone are lost; all previously existing blocks in that zone are discarded. InitApplZone is called by the Segment Loader when starting up an application; you shouldn't normally need to call it.

(warning)

> Reinitializing the application zone from within a running program is tricky, since the program's code itself resides in the application zone. To do it safely, the code containing the InitApplZone call cannot be in the application zone.

The application zone has an initial size of 6K bytes, and can be expanded as needed in 1K increments. Space is initially allocated for 64 master pointers; should more be needed later, they will be added 64 at a time. The zone's grow zone function is set to NIL.

Result codes	noErr	No error

PROCEDURE SetApplBase (startPtr: Ptr);

Trap macro	_SetApplBase
On entry	AØ: startPtr (pointer)
On exit	DØ: result code (integer)

SetApplBase changes the starting address of the application heap zone to the address designated by startPtr, and then calls InitApplZone. SetApplBase is normally called only by the system itself; you should never need to call this procedure.

Since the application heap zone begins immediately following the end of the system zone, changing its starting address has the effect of changing the size of the system zone. The system zone can be made larger, but never smaller; if startPtr points to an address lower than the current end of the system zone, it's ignored and the application zone's starting address is left unchanged.

(warning)
Like InitApplZone, SetApplBase is a tricky operation, because the code of the program itself resides in the application heap zone. To do it safely, the code containing the SetApplBase call cannot be in the application zone.

Result codes noErr No error

```
PROCEDURE InitZone (pGrowZone: ProcPtr; cMoreMasters: INTEGER;
         limitPtr, startPtr: Ptr);
```

Trap macro _InitZone

On entry A∅: pointer to parameter block

Parameter block

∅	startPtr	pointer
4	limitPtr	pointer
8	cMoreMasters	integer
1∅	pGrowZone	pointer

On exit D∅: result code (integer)

InitZone creates a new heap zone, initializes its header and trailer, and makes it the current zone. The startPtr parameter is a pointer to the first byte of the new zone; limitPtr points to the first byte of the zone trailer. The new zone will occupy memory addresses from ORD(startPtr) to ORD(limitPtr)+11.

CMoreMasters tells how many master pointers should be allocated at a time for the new zone. This number of master pointers are created initially; should more be needed later, they will be added in increments of this same number. For the system heap zone, this number is initially 32; for the application heap zone, it's 64.

The pGrowZone parameter is a pointer to the grow zone function for the new zone, if any. If you're not defining a grow zone function for this zone, pass NIL.

The new zone includes a 52-byte header, so its actual usable space runs from ORD(startPtr)+52 through ORD(limitPtr)-1. In addition, each master pointer occupies four bytes within this usable area. Thus the total available space in the zone, in bytes, is initially

```
        ORD(limitPtr) - ORD(startPtr) - 52 - 4*cMoreMasters
```

This number must not be less than Ø. Note that the amount of available space in the zone will decrease as more master pointers are allocated.

Result codes	noErr	No error

PROCEDURE SetApplLimit (zoneLimit: Ptr);

Trap macro	_SetApplLimit
On entry	AØ: zoneLimit (pointer)
On exit	DØ: result code (integer)

SetApplLimit sets the application heap limit, beyond which the application heap zone can't be expanded. The actual expansion isn't under your program's control, but is done automatically by the Memory Manager when necessary to satisfy allocation requests. Only the original application zone can be expanded.

ZoneLimit is a limit pointer to a byte in memory beyond which the zone will not be allowed to grow. The zone can grow to include the byte **preceding** zoneLimit in memory, but no farther. If the zone already extends beyond the specified limit it won't be cut back, but it will be prevented from growing any more.

(warning)

> Notice that zoneLimit is **not** a byte count. To limit the application zone to a particular size (say 8K bytes), you have to write something like
>
> ```
> SetApplLimit(Ptr(ApplicZone) + 8192)
> ```
>
> The Memory Manager function ApplicZone is explained below.

Assembly-language note: The global variable ApplLimit contains the application heap limit.

Result codes	noErr	No error

PROCEDURE MaxApplZone; [No trap macro]

MaxApplZone expands the application heap zone to the application heap limit without purging any blocks currently in the zone. If the zone already extends to the limit, it won't be changed.

Result codes	noErr	No error

PROCEDURE MoreMasters;

Trap macro _MoreMasters

MoreMasters allocates another block of master pointers in the current heap zone. This procedure is usually called very early in an application.

Result codes	noErr	No error
	memFullErr	Not enough room in zone

Heap Zone Access

FUNCTION GetZone : THz;

Trap macro _GetZone

On exit	AØ: function result (pointer) DØ: result code (integer)

GetZone returns a pointer to the current heap zone.

Assembly-language note: The global variable TheZone contains a pointer to the current heap zone.

Result codes   noErr   No error

PROCEDURE SetZone (hz: THz);

Trap macro	_SetZone	
On entry	A0:	hz (pointer)
On exit	D0:	result code (integer)

SetZone sets the current heap zone to the zone pointed to by hz.

Assembly-language note: You can set the current heap zone by storing a pointer to it in the global variable TheZone.

Result codes   noErr   No error

FUNCTION SystemZone : THz; [No trap macro]

SystemZone returns a pointer to the system heap zone.

Assembly-language note: The global variable SysZone contains a pointer to the system heap zone.

Result codes   noErr   No error

FUNCTION ApplicZone : THz; [No trap macro]

ApplicZone returns a pointer to the original application heap zone.

<u>Assembly-language note</u>: The global variable ApplZone contains a pointer to the original application heap zone.

<u>Result codes</u>	noErr	No error

Allocating and Releasing Relocatable Blocks

FUNCTION NewHandle (logicalSize: Size) : Handle;

<u>Trap macro</u>	_NewHandle	
	_NewHandle ,SYS	(applies to system heap)
	_NewHandle ,CLEAR	(clears allocated block)
	_NewHandle ,SYS,CLEAR	(applies to system heap and clears allocated block)
<u>On entry</u>	DØ: logicalSize (long integer)	
<u>On exit</u>	AØ: function result (handle) DØ: result code (integer)	

NewHandle attempts to allocate a new relocatable block of logicalSize bytes from the current heap zone and then return a handle to it. The new block will be unlocked and unpurgeable. The new block will be unlocked and unpurgeable. If logicalSize bytes can't be allocated, NewHandle returns NIL.

NewHandle will pursue all available avenues to create a free block of the requested size, including compacting the heap zone, increasing its size, purging blocks from it, and calling its grow zone function, if any.

<u>Result codes</u>	noErr	No error
	memFullErr	Not enough room in zone

PROCEDURE DisposHandle (h: Handle);

Trap macro	_DisposHandle	
On entry	AØ:	h (handle)
On exit	AØ: DØ:	Ø result code (integer)

DisposHandle releases the memory occupied by the relocatable block whose handle is h.

(warning)

> After a call to DisposHandle, all handles to the released block become invalid and should not be used again.

Result codes	noErr	No error
	memWZErr	Attempt to operate on a free block

FUNCTION GetHandleSize (h: Handle) : Size;

Trap macro	_GetHandleSize	
On entry	AØ:	h (handle)

On exit	DØ:	if >= Ø, function result (long integer)
		if < Ø, result code (integer)

GetHandleSize returns the logical size, in bytes, of the relocatable block whose handle is h. In case of an error, GetHandleSize returns Ø.

Assembly-language note: Recall that the trap dispatcher sets the condition codes before returning from a trap by testing the low-order word of register DØ with a TST.W instruction. Since the block size returned in DØ by _GetHandleSize is a full 32-bit long word, the word-length test sets the condition codes incorrectly in this case. To branch on the contents of DØ, use your own TST.L instruction on return from the trap to test the full 32 bits of the register.

Result codes	noErr	No error [Pascal only]
	nilHandleErr	NIL master pointer
	memWZErr	Attempt to operate on a free block

PROCEDURE SetHandleSize (h: Handle; newSize: Size);

Trap macro	_SetHandleSize
On entry	AØ: h (handle) DØ: newSize (long integer)
On exit	DØ: result code (integer)

SetHandleSize changes the logical size of the relocatable block whose handle is h to newSize bytes.

(note)

> Don't attempt to increase the size of a locked block, because its unlikely the Memory Manager will be able to do so.

Result codes	noErr	No error
	memFullErr	Not enough room to grow
	nilHandleErr	NIL master pointer
	memWZErr	Attempt to operate on a free block

FUNCTION HandleZone (h: Handle) : THz;

Trap macro	_HandleZone
On entry	AØ: h (handle)
On exit	AØ: function result (pointer) DØ: result code (integer)

HandleZone returns a pointer to the heap zone containing the relocatable block whose handle is h.

(warning)

> If handle h is empty (points to a NIL master pointer), HandleZone returns a pointer to the current heap zone. In case of an error, the result returned by HandleZone is meaningless and should be ignored.

Result codes	noErr	No error
	memWZErr	Attempt to operate on a free block

FUNCTION RecoverHandle (p: Ptr) : Handle;

Trap macro	_RecoverHandle	
	_RecoverHandle ,SYS	(applies to system heap)
On entry	AØ: p (pointer)	
On exit	AØ: function result (handle)	
	DØ: unchanged	

RecoverHandle returns a handle to the relocatable block pointed to by p.

Assembly-language note: The trap _RecoverHandle doesn't return a result code in register DØ; the previous contents of DØ are preserved unchanged.

Result codes	noErr	No error	[Pascal only]

PROCEDURE ReallocHandle (h: Handle; logicalSize: Size);

Trap macro	_ReallocHandle
On entry	AØ: h (handle)
	DØ: logicalSize (long integer)
On exit	AØ: original h or Ø
	DØ: result code (integer)

ReallocHandle allocates a new relocatable block with a logical size of logicalSize bytes. It then updates handle h by setting its master pointer to point to the new block. The main use of this procedure is to reallocate space for a block that has been purged. Normally h is an empty handle, but it need not be: if it points to an existing block, that block is released before the new block is created.

In case of an error, no new block is allocated and handle h is left unchanged.

Assembly-language note: On return from _ReallocHandle, register AØ contains the original handle h, or Ø if no room could be found for the requested block.

Result codes		
	noErr	No error
	memFullErr	Not enough room in zone
	memWZErr	Attempt to operate on a free block
	memPurErr	Block is locked

Allocating and Releasing Nonrelocatable Blocks

FUNCTION NewPtr (logicalSize: Size) : Ptr;

Trap macro	_NewPtr	
	_NewPtr ,SYS	(applies to system heap)
	_NewPtr ,CLEAR	(clears allocated block)
	_NewPtr ,SYS,CLEAR	(applies to system heap and clears allocated block)
On entry	DØ: logicalSize (long integer)	
On exit	AØ: function result (pointer) DØ: result code (integer)	

NewPtr attempts to allocate a new nonrelocatable block of logicalSize bytes from the current heap zone and then return a pointer to it. If logicalSize bytes can't be allocated, NewPtr returns NIL.

NewPtr will pursue all available avenues to create a free block of the requested size, including compacting the heap zone, increasing its size, purging blocks from it, and calling its grow zone function, if any.

Result codes		
	noErr	No error
	memFullErr	Not enough room in zone

PROCEDURE DisposPtr (p: Ptr);

Trap macro	_DisposPtr
On entry	AØ: p (pointer)
On exit	AØ: Ø DØ: result code (integer)

DisposPtr releases the memory occupied by the nonrelocatable block pointed to by p.

(warning)
After a call to DisposPtr, all pointers to the released block become invalid and should not be used again.

Result codes		
	noErr	No error
	memWZErr	Attempt to operate on a free block

FUNCTION GetPtrSize (p: Ptr) : Size;

Trap macro	_GetPtrSize
On entry	AØ: p (pointer)
On exit	DØ: if >= Ø, function result (long integer) if < Ø, result code (integer)

GetPtrSize returns the logical size, in bytes, of the nonrelocatable block pointed to by p. In case of an error, GetPtrSize returns Ø.

Assembly-language note: Recall that the trap dispatcher sets the condition codes before returning from a trap by testing the low-order half of register DØ with a TST.W instruction. Since the block size returned in DØ by _GetPtrSize is a full 32-bit long word, the word-length test sets the condition codes incorrectly in this case. To branch on the contents of DØ, use your own TST.L instruction on return from the trap to test the full 32 bits of the register.

Result codes	noErr	No error [Pascal only]
	memWZErr	Attempt to operate on a free block

PROCEDURE SetPtrSize (p: Ptr; newSize: Size);

Trap macro	_SetPtrSize	
On entry	A0:	p (pointer)
	D0:	newSize (long integer)
On exit	D0:	result code (integer)

SetPtrSize changes the logical size of the nonrelocatable block pointed to by p to newSize bytes.

(note)

> Don't attempt to increase the size of a locked block, because it's unlikely the Memory Manager will be able to do so.

Result codes	noErr	No error
	memFullErr	Not enough room to grow
	memWZErr	Attempt to operate on a free block

FUNCTION PtrZone (p: Ptr) : THz;

Trap macro	_PtrZone	
On entry	A0:	p (pointer)
On exit	A0:	function result (pointer)
	D0:	result code (integer)

PtrZone returns a pointer to the heap zone containing the nonrelocatable block pointed to by p. In case of an error, the result returned by PtrZone is meaningless and should be ignored.

Result codes	noErr	No error
	memWZErr	Attempt to operate on a free block

Freeing Space in the Heap

FUNCTION FreeMem : LONGINT;

Trap macro	_FreeMem _FreeMem ,SYS	 (applies to system heap)
On exit	DØ: function result (long integer)	

FreeMem returns the total amount of free space in the current heap zone, in bytes. Notice that it usually isn't possible to allocate a block of this size, because of fragmentation due to nonrelocatable or locked blocks.

Result codes	noErr	No error

FUNCTION MaxMem (VAR grow: Size) : Size;

Trap macro	_MaxMem _MaxMem ,SYS	 (applies to system heap)
On exit	DØ: function result (long integer) AØ: grow (long integer)	

MaxMem compacts the current heap zone and purges all purgeable blocks from the zone. It returns as its result the size in bytes of the largest contiguous free block in the zone after the compaction. If the current zone is the original application heap zone, the variable parameter grow is set to the maximum number of bytes by which the zone can grow. For any other heap zone, grow is set to Ø. MaxMem doesn't actually expand the zone or call its grow zone function.

Result codes	noErr	No error

FUNCTION CompactMem (cbNeeded: Size) : Size;

Trap macro	_CompactMem _CompactMem ,SYS	 (applies to system heap)
On entry	DØ: cbNeeded (long integer)	
On exit	DØ: function result (long integer)	

CompactMem compacts the current heap zone by moving relocatable blocks forward and collecting free space together until a contiguous block of at least cbNeeded free bytes is found or the entire zone is compacted; it doesn't purge any purgeable blocks. CompactMem returns the size in bytes of the largest contiguous free block remaining. Note that it doesn't actually allocate the block.

(note)

> To force a compaction of the entire heap zone, pass maxSize for cbNeeded.

Result codes	noErr	No error

FUNCTION ResrvMem (cbNeeded: Size);

Trap macro	_ResrvMem _ResrvMem ,SYS	 (applies to system heap)
On entry	DØ: cbNeeded (long integer)	
On exit	DØ: result code (integer)	

ResrvMem creates free space for a block of cbNeeded contiguous bytes at the lowest possible position in the current heap zone. It will try every available means to place the block as close as possible to the beginning of the zone, including moving other blocks upward, expanding the zone, or purging blocks from it. Notice that ResrvMem doesn't actually allocate the block.

(note)

> When you allocate a relocatable block that you know will be locked for long periods of time, call ResrvMem first. This reserves space for the block near the beginning of the heap zone, where it will interfere with compaction as little as possible. It isn't necessary to call ResrvMem

for a nonrelocatable block; NewPtr calls it automatically.

Result codes		
	noErr	No error
	memFullErr	Not enough room in zone

PROCEDURE PurgeMem (cbNeeded: Size);

Trap macro	_PurgeMem _PurgeMem ,SYS	 (applies to system heap)
On entry	DØ: cbNeeded (long integer)	
On exit	DØ: result code (integer)	

PurgeMem sequentially purges blocks from the current heap zone until a contiguous block of at least cbNeeded free bytes is created or the entire zone is purged; it doesn't compact the heap zone. Only relocatable, unlocked, purgeable blocks can be purged. Notice that PurgeMem doesn't actually allocate the block.

(note)

To force a purge of the entire heap zone, pass maxSize for cbNeeded.

Result codes		
	noErr	No error
	memFullErr	Not enough room in zone

PROCEDURE EmptyHandle (h: Handle);

Trap macro	_EmptyHandle
On entry	AØ: h (handle)
On exit	AØ: h (handle) DØ: result code (integer)

EmptyHandle purges the relocatable block whose handle is h from its heap zone and sets its master pointer to NIL (making it an empty handle). If h is already empty, EmptyHandle does nothing.

(note)

Since the space occupied by the block's master pointer itself remains allocated, all handles pointing to it remain valid but empty. When you later reallocate space for the block with ReallocHandle, the master pointer will be updated, causing all existing handles to point correctly to the new block.

The block whose handle is h must be unlocked, but need not be purgeable.

Result codes		
	noErr	No error
	memWZErr	Attempt to operate on a free block
	memPurErr	Block is locked

Properties of Relocatable Blocks

PROCEDURE HLock (h: Handle);

Trap macro	_HLock	
On entry	AØ:	h (handle)
On exit	DØ:	result code (integer)

HLock locks a relocatable block, preventing it from being moved within its heap zone. If the block is already locked, HLock does nothing.

Assembly-language note: Changing the value of the block's master pointer's lock bit with a BSET instruction is faster than HLock. However, HLock may eventually perform additional tasks.

Result codes		
	noErr	No error
	nilHandleErr	NIL master pointer
	memWZErr	Attempt to operate on a free block

PROCEDURE HUnlock (h: Handle);

Trap macro	_HUnlock
On entry	A0: h (handle)
On exit	D0: result code (integer)

HUnlock unlocks a relocatable block, allowing it to be moved within its heap zone. If the block is already unlocked, HUnlock does nothing.

Assembly-language note: Changing the value of the block's master pointer's lock bit with a BCLR instruction is faster than HUnlock. However, HUnlock may eventually perform additional tasks.

Result codes	noErr	No error
	nilHandleErr	NIL master pointer
	memWZErr	Attempt to operate on a free block

PROCEDURE HPurge (h: Handle);

Trap macro	_HPurge
On entry	A0: h (handle)
On exit	D0: result code (integer)

HPurge marks a relocatable block as purgeable. If the block is already purgeable, HPurge does nothing.

Result codes	noErr	No error
	nilHandleErr	NIL master pointer
	memWZErr	Attempt to operate on a free block

PROCEDURE HNoPurge (h: Handle);

Trap macro	_HNoPurge
On entry	AØ: h (handle)
On exit	DØ: result code (integer)

HNoPurge marks a relocatable block as unpurgeable. If the block is already unpurgeable, HNoPurge does nothing.

Result codes	noErr	No error
	nilHandleErr	NIL master pointer
	memWZErr	Attempt to operate on a free block

Grow Zone Operations

PROCEDURE SetGrowZone (growZone: ProcPtr);

Trap macro	_SetGrowZone
On entry	AØ: growZone (pointer)
On exit	DØ: result code (integer)

SetGrowZone sets the current heap zone's grow zone function as designated by the growZone parameter. A NIL parameter value removes any grow zone function the zone may previously have had.

(note)

> If your program presses the limits of the available heap space, it's a good idea to have a grow zone function of some sort. At the very least, the grow zone function should detect when the Memory Manager is about to run out of space at a critical time (see GZCritical, below) and take some graceful action--such as displaying an alert box with the message "Out of memory"--instead of just failing unpredictably.

The Memory Manager calls the grow zone function as a last resort when trying to allocate space, if it has failed to create a block of the needed size after compacting the zone, increasing its size (in the case of the original application zone), or purging blocks from it. Memory

Manager routines that may cause the grow zone function to be called are NewHandle, NewPtr, SetHandleSize, SetPtrSize, ReallocHandle, and ResrvMem.

The grow zone function should be of the form

```
FUNCTION MyGrowZone (cbNeeded: Size) : Size;
```

The cbNeeded parameter gives the physical size of the needed block in bytes, **including the block header.** The grow zone function should attempt to create a free block of at least this size. It should return a nonzero number if it's able to allocate some memory, or Ø if it's not able to allocate any.

If the grow zone function returns Ø, the Memory Manager will give up trying to allocate the needed block and will signal failure with the result code memFullFrr. Otherwise it will compact the heap zone and try again to allocate the block. If still unsuccessful, it will continue to call the grow zone function repeatedly, compacting the zone again after each call, until it either succeeds in allocating the needed block or receives a zero result and gives up.

The usual way for the grow zone function to free more space is to call EmptyHandle to purge blocks that were previously marked unpurgeable. Another possibility is to unlock blocks that were previously locked.

(note)

> Although just unlocking blocks doesn't actually free any additional space in the zone, the grow zone function should still return a nonzero result in this case. This signals the Memory Manager to compact the heap and try again to allocate the needed block.

(warning)

> Depending on the circumstances in which the grow zone function is called, there may be particular blocks within the heap zone that must not be purged or released. For instance, if your program is attempting to increase the size of a relocatable block with SetHandleSize, it would be disastrous to release the block being expanded. To deal with such cases safely, it's essential to understand the use of the functions GZCritical and GZSaveHnd (see below).

(warning)

> Whenever you call the Resource Manager with SetResPurge(TRUE), it installs its own grow zone function into the application heap zone. The Resource Manager's grow zone function automatically writes to the disk all changed resources before they're purged. If you install your own grow zone function into the application heap zone, you shouldn't call SetResPurge(TRUE).

Result codes   noErr   No error

FUNCTION GZCritical : BOOLEAN; [No trap macro]

GZCritical returns TRUE if the Memory Manager critically needs space--for example, to create a new relocatable or nonrelocatable block or to reallocate a handle. It returns FALSE in less critical cases, such as ResrvMem trying to reserve space as low as possible in the heap zone or SetHandleSize trying to increase the size of a relocatable block. GZCritical doesn't affect the value returned by MemError.

(warning)
If you're writing a grow zone function in Pascal, you should always call GZCritical and proceed only if the result is TRUE. All the information you need to handle the critical cases safely is the value of GZSaveHnd (see below). The noncritical cases require additional information that isn't available from Pascal, so your grow zone function should just return Ø and not attempt to free any space.

Assembly-language note: To find out whether a given grow zone call is critical, you can use the following:

```
          MOVE.L  GZMoveHnd,DØ
          BEQ.S   Critical
          CMP.L   GZRootHnd,DØ
          BEQ.S   Critical

          CLR.L   4(SP)          ;if noncritical, just return Ø
          RTS

Critical  . . .                  ;handle critical case
```

To handle the critical cases safely (and the noncritical ones if you choose to do more than just return Ø), see the note below under GZSaveHnd.

FUNCTION GZSaveHnd : Handle; [No trap macro]

GZSaveHnd returns a handle to a relocatable block that mustn't be purged or released by the grow zone function, or NIL if there is no such block. For example, during a SetHandleSize call, the handle being changed mustn't be purged. The grow zone function will be safe if it avoids purging or releasing this block, **provided that the grow zone call was critical.** To handle noncritical cases safely, further information is needed that isn't available from Pascal. GZSaveHnd doesn't affect the value returned by MemError.

Assembly-language note: You can find the same handle in the global variable GZRootHnd. The "further information" that isn't available from Pascal is the contents of two other global variables, GZRootPtr and GZMoveHnd, which may be nonzero in noncritical cases. If GZRootPtr is nonzero, it's a pointer to a nonrelocatable block that must not be released; GZMoveHnd is a handle to a relocatable block that must not be released but may be purged.

Miscellaneous Routines

PROCEDURE BlockMove (sourcePtr,destPtr: Ptr; byteCount: Size);

Trap macro	_BlockMove	
On entry	AØ:	sourcePtr (pointer)
	A1:	destPtr (pointer)
	DØ:	byteCount (long integer)
On exit	DØ:	result code (integer)

BlockMove moves a block of byteCount consecutive bytes from the address designated by sourcePtr to that designated by destPtr. No pointers are updated.

Result codes	noErr	No error

FUNCTION TopMem : Ptr; [No trap macro]

TopMem returns a pointer to the address following the last byte of RAM.

Assembly-language note: To get a pointer to the end of RAM from assembly language, use the global variable MemTop.

Result codes	noErr	No error

FUNCTION MemError : OSErr; [No trap macro]

MemError returns the result code produced by the last Memory Manager routine called. (OSErr is an Operating System Utility data type declared as INTEGER.)

Assembly-language note: To get a routine's result code from assembly language, look in register DØ on return from the routine (except for certain routines as noted).

SPECIAL TECHNIQUES

This section describes some special or unusual techniques that you may find useful.

Subdividing the Application Heap Zone

In some applications, you may want to subdivide the original application heap zone into two or more independent zones to be used for different purposes. In doing this, it's important not to destroy any existing blocks in the original zone (such as those containing the code of your program). The recommended procedure is to allocate space for the subzones as nonrelocatable blocks within the original zone, then use InitZone to initialize them as independent zones. For example, to divide the available space in the application zone in half, you might write something like the following:

```
CONST minSize = 576;  {52 + 12 + 32*(12 + 4): zone header, }
                      { zone trailer, and 32 minimum-size }
                      { blocks with master pointers}
VAR myZone1, myZone2: THz;
    start, limit: Ptr;
    availSpace, zoneSize: Size;
    . . . ;
BEGIN
. . . ;
availSpace := CompactMem(maxSize);          {size of largest free block}
zoneSize := 2 * (availSpace DIV 4) - 8);  {force new zone size to }
                                            { an even number of bytes}
IF zoneSize < minSize                       {need 8 bytes for block }
                                            { header}
  THEN . . .                                {error--not enough room}
  ELSE
    BEGIN
    start := NewPtr(zoneSize);              {allocate nonrelocatable block}
    limit := POINTER(ORD(start) + zoneSize);
    InitZone(NIL, 32, limit, start);
    myZone1 := THz(start);                  {convert Ptr to THz}

    start := NewPtr(zoneSize);              {allocate nonrelocatable block}
    limit := POINTER(ORD(start) + zoneSize);
    InitZone(NIL, 32, limit, start);
    myZone2 := THz(start)                   {convert Ptr to THz}
    END;
. . .
END
```

Assembly-language note: The equivalent assembly code might be as follows:

```
minSize   .EQU     52+12+<32*<12+4>> ;zone header, zone trailer,
                                     ; and 32 minimum-size blocks
                                     ; with master pointers
          . . .
          MOVE.L   #maxSize,DØ      ;compact entire zone
          _CompactMem               ;DØ has size of largest free block

          ASR.L    #2,DØ            ;force new zone size to an
          ASL.L    #1,DØ            ; even number of bytes
          SUBQ.L   #8,DØ            ;adjust for block header
          CMP.L    #minSize,DØ      ;need 8 bytes for block header
          BLO      NoRoom           ;error if < minimum size

          MOVE.L   DØ,D1            ;save zone size
          _NewPtr                   ;allocate nonrelocatable block
          MOVE.L   AØ,myZone1       ;store zone pointer

          CLR.L    -(SP)            ;NIL grow zone function
          MOVE.W   #32,-(SP)        ;allocate 32 master pointers
          MOVE.L   AØ,-(SP)         ;AØ has zone pointer
          ADD.L    D1,(SP)          ;convert to limit pointer
          MOVE.L   AØ,-(SP)         ;push as start pointer

          MOVE.L   SP,AØ            ;point to argument block
          _InitZone                 ;create zone 1

          MOVE.L   D1,DØ            ;get back zone size
          _NewPtr                   ;allocate nonrelocatable block
          MOVE.L   AØ,myZone2       ;store zone pointer

          MOVE.L   AØ,4(SP)         ;move zone pointer to stack
          ADD.L    D1,4(SP)         ;convert to limit pointer
          MOVE.L   AØ,(SP)          ;move to stack as start pointer

          MOVE.L   SP,AØ            ;point to argument block
          _InitZone                 ;create zone 2
          ADD.W    #14,SP           ;pop arguments off stack
          . . .
```

Creating a Heap Zone on the Stack

Another place you can get the space for a new heap zone is from the stack. For example:

```
CONST zoneSize = 2048;
VAR zoneArea: PACKED ARRAY [1..zoneSize] OF SignedByte;
    stackZone: THz;
    limit: Ptr;
    . . . ;
BEGIN
. . . ;
stackZone := @zoneArea;
limit := POINTER(ORD(stackZone) + zoneSize);
InitZone(NIL, 16, limit, @zoneArea);
. . .
END
```

The heap zone created by this method will be usable up until the time that this routine is completed (because its variables will be released).

Assembly-language note: Here's how you might do the same thing in assembly language:

```
zoneSize  .EQU    2048
          . . .
          MOVE.L  SP,A2           ;save stack pointer for limit
          SUB.W   #zoneSize,SP    ;make room on stack
          MOVE.L  SP,A1           ;save stack pointer for start
          MOVE.L  A1,stackZone    ;store as zone pointer

          CLR.L   -(SP)           ;NIL grow zone function
          MOVE.W  #16,-(SP)       ;allocate 16 master pointers
          MOVE.L  A2,-(SP)        ;push limit pointer
          MOVE.L  A1,-(SP)        ;push start pointer

          MOVE.L  SP,A0           ;point to argument block
          _InitZone               ;create new zone
          ADD.W   #14,SP          ;pop arguments off stack
          . . .
```

Pointer and Handle Conversion

To save time in critical situations in assembly language, here's a quick way to convert a dereferenced pointer to a relocatable block back into a handle without paying the overhead of a _RecoverHandle trap. Recall that the relative handle stored in the block's header is the offset of the block's master pointer relative to the start of its heap zone. So to convert a copy of the master pointer back into the original handle, find the relative handle and add it to the address of the zone. For example, if register A2 contains the master pointer of a block in the current heap zone, the following code will reconstruct the block's handle in A3:

```
MOVE.L  -4(A2),A3       ;relative handle is 4 bytes back
                        ; from start of contents
ADD.L   TheZone,A3      ;use as offset from start of zone
```

(note)

This example works only when the handle belongs to the zone pointed to by TheZone.

Conversely, given a true (absolute) handle to a relocatable block, you can find the zone the block belongs to by subtracting the relative handle from the absolute handle. If the absolute handle is in register A2, the following instructions will convert it into a pointer to the block's heap zone:

```
MOVE.L  (A2),A3         ;get pointer to block
SUB.L   -4(A3),A2       ;subtract relative handle
                        ; to get zone pointer
```

For nonrelocatable blocks, the header contains a pointer directly back to the zone:

```
MOVE.L  -4(A2),A2       ;get zone pointer directly
```

SUMMARY OF THE MEMORY MANAGER

Constants

```
CONST maxSize       = $8ØØØØØ; {maximum block size}

      { Result codes }

      memFullErr    = -1Ø8;    {not enough room in zone}
      memPurErr     = -112;    {attempt to purge a locked block}
      memWZErr      = -111;    {attempt to operate on a free block}
      nilHandleErr  = -1Ø9;    {NIL master pointer}
      noErr         = Ø;       {no error}
```

Data Types

```
TYPE SignedByte = -128..127;
     Byte       = Ø..255;
     Ptr        = ^SignedByte;
     Handle     = ^Ptr;

     Str255       = String[255];
     StringPtr    = ^Str255;
     StringHandle = ^StringPtr;

     ProcPtr = Ptr;

     Fixed = LONGINT;

     Size = LONGINT;
     THz  = ^Zone;
     Zone = RECORD
              bkLim:      Ptr;     {limit pointer}
              purgePtr:   Ptr;     {used internally}
              hFstFree:   Ptr;     {first free master pointer}
              zcbFree:    LONGINT; {number of free bytes}
              gzProc:     ProcPtr; {grow zone function}
              moreMast:   INTEGER; {master pointers to allocate}
              flags:      INTEGER; {used internally}
              cntRel:     INTEGER; {relocatable blocks}
              maxRel:     INTEGER; {maximum cntRel value}
              cntNRel:    INTEGER; {nonrelocatable blocks}
              maxNRel:    INTEGER; {maximum maxRel value}
              cntEmpty:   INTEGER; {empty master pointers}
              cntHandles: INTEGER; {total master pointers}
              minCBFree:  LONGINT; {minimum zcbFree value}
              purgeProc:  ProcPtr; {purge warning procedure}
              sparePtr:   Ptr;     {used internally}
              allocPtr:   Ptr;     {used internally}
              heapData:   INTEGER  {first usable byte in zone}
            END;
```

Routines

Initialization and Allocation

```
PROCEDURE InitApplZone;
PROCEDURE SetApplBase   (startPtr: Ptr);
PROCEDURE InitZone      (pGrowZone: ProcPtr; cMoreMasters: INTEGER;
                         limitPtr,startPtr: Ptr);
PROCEDURE SetApplLimit  (zoneLimit: Ptr);
PROCEDURE MaxApplZone;  [No trap macro]
PROCEDURE MoreMasters;
```

Heap Zone Access

```
FUNCTION  GetZone :    THz;
PROCEDURE SetZone     (hz: THz);
FUNCTION  SystemZone : THz;  [No trap macro]
FUNCTION  ApplicZone : THz;  [No trap macro]
```

Allocating and Releasing Relocatable Blocks

```
FUNCTION  NewHandle     (logicalSize: Size) : Handle;
PROCEDURE DisposHandle  (h: Handle);
FUNCTION  GetHandleSize (h: Handle) : Size;
PROCEDURE SetHandleSize (h: Handle; newSize: Size);
FUNCTION  HandleZone    (h: Handle) : THz;
FUNCTION  RecoverHandle (p: Ptr) : Handle;
PROCEDURE ReallocHandle (h: Handle; logicalSize: Size);
```

Allocating and Releasing Nonrelocatable Blocks

```
FUNCTION  NewPtr     (logicalSize: Size) : Ptr;
PROCEDURE DisposPtr  (p: Ptr);
FUNCTION  GetPtrSize (p: Ptr) : Size;
PROCEDURE SetPtrSize (p: Ptr; newSize: Size);
FUNCTION  PtrZone    (p: Ptr) : THz;
```

Freeing Space in the Heap

```
FUNCTION  FreeMem :    LONGINT;
FUNCTION  MaxMem      (VAR grow: Size) : Size;
FUNCTION  CompactMem  (cbNeeded: Size) : Size;
PROCEDURE ResrvMem    (cbNeeded: Size);
PROCEDURE PurgeMem    (cbNeeded: Size);
PROCEDURE EmptyHandle (h: Handle);
```

Properties of Relocatable Blocks

```
PROCEDURE HLock    (h: Handle);
PROCEDURE HUnlock  (h: Handle);
PROCEDURE HPurge   (h: Handle);
PROCEDURE HNoPurge (h: Handle);
```

Grow Zone Operations

```
PROCEDURE SetGrowZone (growZone: ProcPtr);
FUNCTION  GZCritical : BOOLEAN;  [No trap macro]
FUNCTION  GZSaveHnd :  Handle;  [No trap macro]
```

Miscellaneous Routines

```
PROCEDURE BlockMove (sourcePtr,destPtr: Ptr; byteCount: Size);
FUNCTION  TopMem :    Ptr;  [No trap macro]
FUNCTION  MemError : OSErr;  [No trap macro]
```

Grow Zone Function

```
FUNCTION  MyGrowZone (cbNeeded: Size) : Size;
```

Assembly-Language Information

Constants

```
; Master pointer counts

dfltMasters    .EQU    32         ;default master-pointer count
maxMasters     .EQU    $1000      ;maximum master-pointer count (4K)

; Heap zone parameters

sysZoneSize    .EQU    $4000      ;default size of system heap zone (16K)
heapStart      .EQU    $0B00      ;start address of system heap zone
appZoneSize    .EQU    $1800      ;initial size of application zone (6K)
minZone        .EQU    52+12+<32*<12+4>>
                                  ;minimum size of application zone

; Stack parameters

dfltStackSize .EQU     $2000      ;initial space allotment for stack (8K)
mnStackSize    .EQU    $400       ;minimum space allotment for stack (1K)

; Values for tag byte of a block header

tybkFree       .EQU    0          ;free block
tybkNRel       .EQU    1          ;nonrelocatable block
tybkRel        .EQU    2          ;relocatable block

; Masks for the fields of a block header

tagMask        .EQU    $C0000000 ;tag field of block header
bcOffMask      .EQU    $0F000000 ;size correction
bcMask         .EQU    $00FFFFFF ;physical block size
freeTag        .EQU    0          ;tag for free block
nRelTag        .EQU    $40000000 ;tag for nonrelocatable block
relTag         .EQU    $80000000 ;tag for relocatable block

; Masks for the fields of a master pointer

ptrMask        .EQU    $00FFFFFF ;address part of master pointer
handleMask     .EQU    $00FFFFFF ;address part of master pointer

; Flags for the high-order byte of a master pointer

lock           .EQU    7          ;lock bit
purge          .EQU    6          ;purge bit
resource       .EQU    5          ;resource bit
```

```
; Result codes

memFullErr      .EQU   -1Ø8       ;not enough room in zone
memPurErr       .EQU   -112       ;attempt to purge a locked block
memWZErr        .EQU   -111       ;attempt to operate on a free block
nilHandleErr    .EQU   -1Ø9       ;NIL master pointer
noErr           .EQU      Ø       ;no error
```

Zone Record Data Structure

bkLim	Limit pointer
hFstFree	First free master pointer
zcbFree	Number of free bytes
gzProc	Grow zone function
mAllocCnt	Master pointers to allocate
cntRel	Relocatable blocks
maxRel	Maximum cntRel value
cntNRel	Nonrelocatable blocks
maxNRel	Maximum maxRel value
cntEmpty	Empty master pointers
cntHandles	Total master pointers
minCBFree	Minimum zcbFree value
purgeProc	Purge warning procedure
heapData	First usable byte in zone

Block Header Data Structure

tagBC	Tag, size correction, and physical byte count
handle	Relocatable block: relative handle Nonrelocatable block: zone pointer
blkData	First byte of block contents

Parameter Block Structure for InitZone

startPtr	Pointer to first byte in zone
limitPtr	Pointer to first byte in zone trailer
cMoreMasters	Number of master pointers for zone
pGrowZone	Pointer to grow zone function

Routines

Name	On entry	On exit
InitApplZone		DØ: result code (int)
SetApplBase	AØ: startPtr (ptr)	DØ: result code (int)
InitZone	AØ: ptr to parameter block Ø startPtr (ptr) 4 limitPtr (ptr) 8 cMoreMasters (int) 1Ø pGrowZone (ptr)	DØ: result code (int)
SetApplLimit	AØ: zoneLimit (ptr)	DØ: result code (int)

MoreMasters		
GetZone	AØ: function result (ptr)	DØ: result code (int)
SetZone	AØ: hz (ptr)	DØ: result code (int)
NewHandle	DØ: logicalSize (longint)	AØ: function result (handle) DØ: result code (int)
DisposHandle	AØ: h (handle)	AØ: Ø DØ: result code (int)
GetHandleSize	AØ: h (handle)	DØ: if >=Ø, function result (longint) if <Ø, result code (int)
SetHandleSize	AØ: h (handle) DØ: newSize (longint)	DØ: result code (int)
HandleZone	AØ: h (handle)	AØ: function result (ptr) DØ: result code (int)
RecoverHandle	AØ: p (ptr)	AØ: function result (handle) DØ: unchanged
ReallocHandle	AØ: h (handle) DØ: logicalSize (longint)	AØ: original h or Ø DØ: result code (int)
NewPtr	DØ: logicalSize (longint)	AØ: function result (ptr) DØ: result code (int)
DisposPtr	AØ: p (ptr)	AØ: Ø DØ: result code (int)
GetPtrSize	AØ: p (ptr)	DØ: if >=Ø, function result (longint) if <Ø, result code (int)
SetPtrSize	AØ: p (ptr) DØ: newSize (longint)	DØ: result code (int)
PtrZone	AØ: p (ptr)	AØ: function result (ptr) DØ: result code (int)
FreeMem		DØ: function result (longint)
MaxMem		DØ: function result (longint) AØ: grow (longint)
CompactMem	DØ: cbNeeded (longint)	DØ: function result (longint)
ResrvMem	DØ: cbNeeded (longint)	DØ: result code (int)
PurgeMem	DØ: cbNeeded (longint)	DØ: result code (int)
EmptyHandle	AØ: h (handle)	AØ: h (handle) DØ: result code (int)
HLock	AØ: h (handle)	DØ: result code (int)
HUnlock	AØ: h (handle)	DØ: result code (int)
HPurge	AØ: h (handle)	DØ: result code (int)
HNoPurge	AØ: h (handle)	DØ: result code (int)
SetGrowZone	AØ: growZone (ptr)	DØ: result code (int)
BlockMove	AØ: sourcePtr (ptr) A1: destPtr (ptr) DØ: byteCount (longint)	DØ: result code (int)

Variables

Name	Size	Contents
BufPtr	4 bytes	Pointer to end of application parameters
MinStack	4 bytes	Minimum space allotment for stack
DefltStack	4 bytes	Default space allotment for stack
HeapEnd	4 bytes	Current limit address of application heap zone
ApplLimit	4 bytes	Application heap limit
SysZone	4 bytes	Pointer to system heap zone
ApplZone	4 bytes	Pointer to application heap zone
TheZone	4 bytes	Pointer to current heap zone
SysZone	4 bytes	Pointer to start of system heap
ApplZone	4 bytes	Pointer to start of application heap
HeapEnd	4 bytes	Pointer to end of application heap
CurStackBase	4 bytes	Pointer to base (end) of stack; start of application globals
ScrnBase	4 bytes	Pointer to start of main screen buffer
SoundLow	4 bytes	Start of main sound buffer
CurrentA5	4 bytes	Current value of A5
MemTop	4 bytes	Pointer to end of RAM
GZHandle	4 bytes	Used by GZSaveHnd and GZCritical
GZRootPtr	4 bytes	Used by GZSaveHnd and GZCritical
GZMoveHnd	4 bytes	Used by GZSaveHnd and GZCritical

GLOSSARY

allocate: To reserve an area of memory for use.

application heap zone: The heap zone initially provided by the Memory Manager for use by the application program; initially equivalent to the application heap, but may be subdivided into two or more independent heap zones.

application heap limit: The boundary between the space available for the application heap and the space available for the stack.

application space: Memory between the system heap and screen and sound buffers, available for dynamic allocation by applications.

block: An area of contiguous memory within a heap zone.

block contents: The area of a block available for use.

block header: The internal "housekeeping" information maintained by the Memory Manager at the beginning of each block in a heap zone.

compaction: The process of moving allocated blocks within a heap zone in order to collect the free space into a single block.

current heap zone: The heap zone currently under attention, to which most Memory Manager operations implicitly apply.

dereference: To refer to a block by its master pointer instead of its handle.

empty handle: A handle that points to a NIL master pointer, signifying that the underlying relocatable block has been purged.

free block: A block containing space available for allocation.

grow zone function: A function supplied by the application program to help the Memory Manager create free space within a heap zone.

handle: A pointer to a master pointer, which designates a relocatable block by double indirection.

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

heap zone: An area of memory initialized by the Memory Manager for heap allocation.

limit pointer: A pointer to the first byte of a zone trailer (that is, to the byte following the last byte of usable space in a heap zone).

lock: To temporarily prevent a relocatable block from being moved during heap compaction.

lock bit: A bit in the master pointer to a relocatable block that indicates whether the block is currently locked.

logical size: The number of bytes in a block's contents.

master pointer: A single pointer to a relocatable block, maintained by the Memory Manager and updated whenever the block is moved, purged, or reallocated. All handles to a relocatable block refer to it by double indirection through the master pointer.

nonrelocatable block: A block whose location in its heap zone is fixed and can't be moved during heap compaction.

physical size: The actual number of bytes a block occupies within its heap zone.

purge: To remove a relocatable block from its heap zone, leaving its master pointer allocated but set to NIL.

purge bit: A bit in the master pointer to a relocatable block that indicates whether the block is currently purgeable.

purge warning procedure: A procedure associated with a particular heap zone that is called whenever a block is purged from that zone.

purgeable block: A relocatable block that can be purged from its heap zone.

reallocate: To allocate new space in a heap zone for a purged block, updating its master pointer to point to its new location.

relative handle: A handle to a relocatable block expressed as the offset of its master pointer within the heap zone, rather than as the absolute memory address of the master pointer.

release: To free an allocated block of memory.

relocatable block: A block that can be moved within its heap zone during compaction.

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

system heap zone: The heap zone provided by the Memory Manager for use by the Macintosh system software; equivalent to the system heap.

unlock: To allow a relocatable block to be moved during heap compaction.

unpurgeable block: A relocatable block that can't be purged from its heap zone.

zone header: The internal "housekeeping" information maintained by the Memory Manager at the beginning of each heap zone.

zone pointer: A pointer to a zone record.

zone record: A data structure representing a heap zone.

zone trailer: A minimum-size free block marking the end of a heap zone.

The Segment Loader: A Programmer's Guide /SEGLOAD/SEG

See Also: Inside Macintosh: A Road Map
Macintosh Memory Management: An Introduction
Programming Macintosh Applications in Assembly Language
The Resource Manager: A Programmer's Guide
The Memory Manager: A Programmer's Guide
The Scrap Manager: A Programmer's Guide
The File Manager: A Programmer's Guide

Modification History:			
	First Draft (ROM 4)	Caroline Rose	6/24/83
	Second Draft (ROM 7)	Caroline Rose & Bradley Hacker	8/24/84

ABSTRACT

The Segment Loader is the part of the Macintosh Operating System that lets you divide your application into several parts and have only some of them in memory at a time. When an application starts up, the Segment Loader also provides it with a list of which files to open or print. This manual describes the Segment Loader.

Summary of significant changes and additions since the first draft:

- The discussion of application parameters has been revised (page 4). Details about the Finder information passed at startup have been moved here from the manual The Structure of a Macintosh Application.

- There's a new way for Pascal programmers to access the Finder information: it involves using the new routines CountAppFiles, GetAppFiles, and ClrAppFiles (page 8).

TABLE OF CONTENTS

ABOUT THIS MANUAL

The Segment Loader is the part of the Macintosh Operating System that lets you divide your application into several parts and have only some of them in memory at a time. When an application starts up, the Segment Loader also provides it with a list of which files to open or print. This manual describes the Segment Loader. *** Eventually it will become part of the comprehensive Inside Macintosh manual. ***

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal and the information in the following manuals:

- Inside Macintosh: A Road Map
- Macintosh Memory Management: An Introduction
- Programming Macintosh Applications in Assembly Language, if you're using assembly language

You should also be familiar with:

- the basic concepts behind the Resource Manager
- the Memory Manager

ABOUT THE SEGMENT LOADER

The Segment Loader allows you to divide the code of your application into several parts or segments. The Finder starts up an application by calling a Segment Loader routine that loads in the main segment (the one containing the main program). Other segments are loaded in automatically when they're needed. Your application can call the Segment Loader to have these other segments removed from memory when they're no longer needed.

The Segment Loader enables you to have programs larger than 32K bytes, the maximum size of a single segment. Also, any code that isn't executed often (such as code for printing) needn't occupy memory when it isn't being used, but can instead be in a separate segment that's "swapped in" when needed.

This mechanism may remind you of the resources of an application, which the Resource Manager of the User Interface Toolbox reads into memory when necessary. An application's segments are in fact themselves stored as resources; their resource type is 'CODE'. A "loaded" segment has been read into memory by the Resource Manager and locked (so that it's neither relocatable nor purgeable). When a segment is unloaded, it's made relocatable and purgeable. You can create these resources from your application code and store them in resource files with the aid of the Resource Editor *** (eventually; for now, the Resource Compiler) ***.

Every segment has a name. If you do nothing about dividing your program into segments, it will consist of a single segment whose name is blank (eight spaces). Dividing your program into segments means specifying in your source file the beginning of each segment by name. The names are for your use only; they're not kept around after linking.

(warning)

> If you do specify segment names, note that normally the main segment should have a blank name. The reason for this is that the intrinsic Pascal routines must be in the same segment as your main program, and the Linker puts those routines in the blank-named segment (so that the right thing will happen if you don't specify any segment names at all).

APPLICATION PARAMETERS

When an application is started up, certain parameters are stored in 32 bytes of memory just above the application's globals, as shown in Figure 1; these are called the application parameters. The Segment Loader adjusts the size of the application globals according to application's needs and sets register A5 to point to the first of the application parameters.

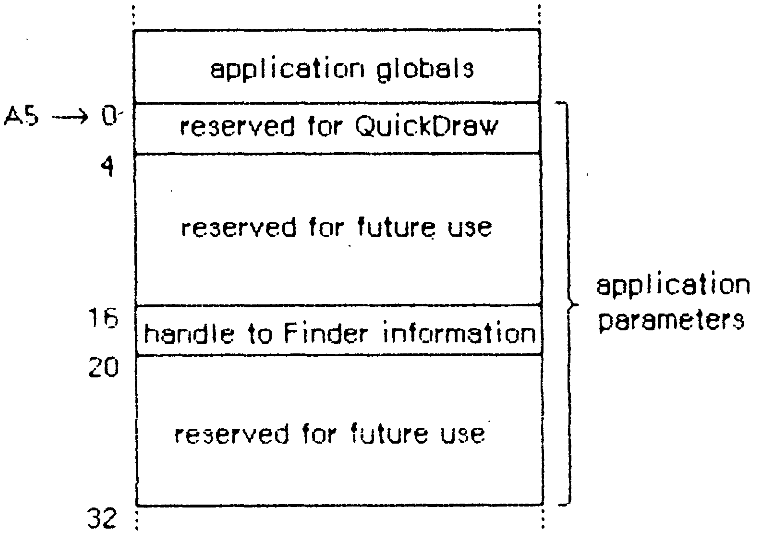


Figure 1. Application Parameters

The majority of the application parameters are reserved for future use or for use by QuickDraw, but there's a handle to the Finder information that all applications will need to access. When the Finder starts up your application it passes along a list of documents selected by the user to be printed or opened, if any. This information is called the Finder information; its structure is shown in Figure 2.

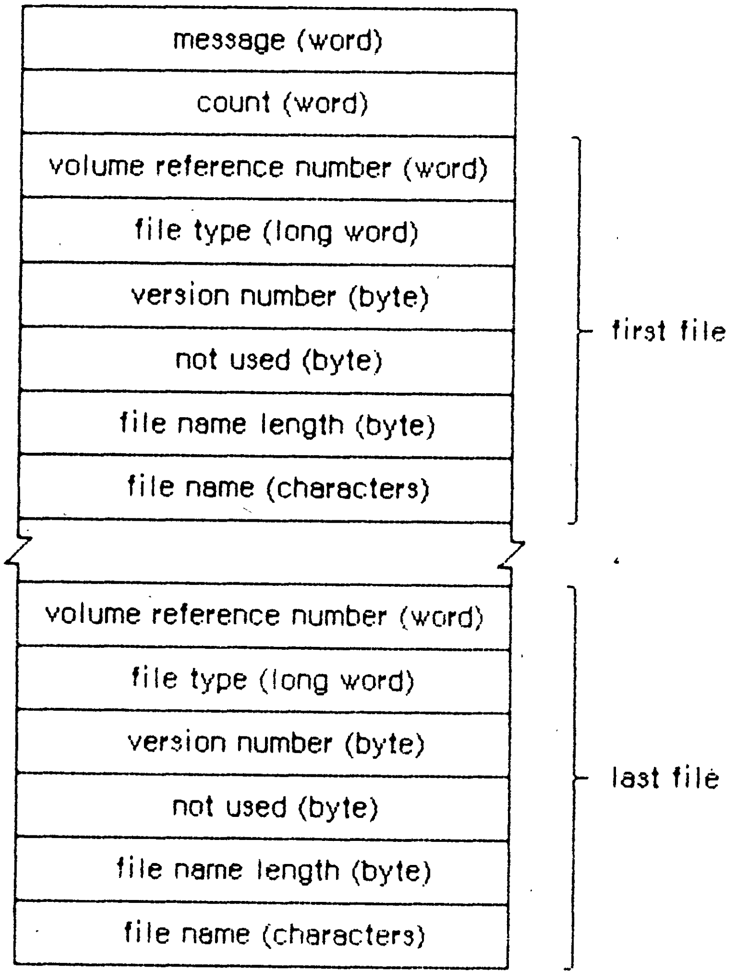


Figure 2. Finder Information

It's up to your application to access the Finder information and open or print the files selected by the user.

The message in the first word of the Finder information indicates whether the documents are to be opened (Ø) or printed (1), and the count following it indicates the number of documents (Ø if none). The rest of the Finder information specifies each of the selected documents by volume reference number, file type, version number, and file name; these terms are explained in the File Manager manual. File names are padded to an even number of bytes if necessary.

Your application should start up with an empty untitled document on the desktop if there are no documents listed in the Finder information. If one or more documents are to be opened, your application should go through each document one at a time, and determine whether it can be opened. If it can be opened, you should do so, and then check the next document in the list (unless you've opened your maximum number of documents, in which case you should ignore the rest). If your application doesn't recognize a document's file type (which can happen if the user selected your application along with another application's document), you may want to open the document anyway and check its internal structure to see if it's a compatible type. Display an alert box including the name of each document that can't be opened.

If one or more documents are to be printed, your application should go through each document in the list and determine whether it can be printed. If the document can be printed, your application should do so--preferably without doing its entire startup sequence. For example, it may not be necessary to show the menu bar or a document window, and reading the desk scrap into memory is definitely not required. If the document can't be printed, ignore it.

*** The above information will be moved out of the next draft of The Structure of a Macintosh Application. ***

USING THE SEGMENT LOADER

This section introduces you to the Segment Loader routines and how they fit into the flow of an application program. The routines themselves are described in detail in the next section.

When your application is first started up, you should determine whether any documents were selected to be printed or opened by it. First call CountAppFiles, which returns the number of selected documents and indicates whether they are to be printed or opened. If the number of selected documents is Ø, open an untitled document in the normal manner. Otherwise, call GetAppFiles once for each selected document. GetAppFiles returns information about each document, including its file type. Based on the file type, your application can decide how to treat the document, as described in the preceding section. For each document that your application opens or prints, call ClrAppFiles, which indicates to the Finder that you've processed it.

> Assembly-language note: Instead of using CountAppFiles, GetAppFiles, and ClrAppFiles, assembly-language programmers can access the Finder information via the global variable AppParmHandle, which contains a handle to the Finder information. Parse the Finder information as shown in Figure 2 above. For each document that your application opens or prints, set the file type in the Finer information to Ø.

To unload a segment when it's no longer needed, call UnloadSeg. If you don't want to keep track of when each particular segment should be unloaded, you can call UnloadSeg for every segment in your application at the end of your main event loop. This isn't harmful, since the segments aren't purged unless necessary.

(warning)

> A segment should never unload the segment that called it, because the return addresses on the stack would refer to code no longer in memory.

Another procedure, GetAppParms, lets you get information about your application such as its name and the reference number for its resource file.

> Assembly-language note: Assembly-language programmers can get the application name and reference number from the global variables CurApName and CurApRefNum.

The Segment Loader also provides the ExitToShell procedure--a way for an application to quit and return the user to the Finder.

Finally, there are three routines that can only be called from assembly language: Chain, Launch, and LoadSeg. Chain starts up another application without disturbing the application heap. Thus the current application can let another application take over while still keeping its data around in the heap. Launch is called by the Finder to start up an application; it's like Chain but doesn't retain the application heap. LoadSeg is called indirectly (via the jump table, as described later) to load segments when necessary--that is, whenever a routine in an unloaded segment is invoked. Most applications will never use Launch or LoadSeg.

SEGMENT LOADER ROUTINES

This section is split into two parts: the first describes the Segment Loader routines available to Pascal programmers and the second describes the routines available to assembly-language programmers.

> Assembly-language note: All the routines that have trap macros are described in detail in the second part of this section. There are no trap macros for the routines CountAppFiles, GetAppFiles, and ClearAppFiles.

PROCEDURE UnloadSeg (routineAddr: Ptr);

UnloadSeg unloads a segment, making it relocatable and purgeable; routineAddr is the address of any externally referenced routine in the segment. The segment won't actually be purged until the memory it occupies is needed. If the segment is purged, the Segment Loader will reload it the next time one of the routines in it is called.

PROCEDURE CountAppFiles (VAR message: INTEGER; VAR count: INTEGER);

CountAppFiles deciphers the Finder information passed to your application, and returns information about the documents that were selected when your application was started up. It returns the number of selected documents in the count parameter, and a number in the message parameter that indicates whether the documents are to opened or printed:

```
CONST appOpen  = Ø;  {open the document(s)}
      appPrint = 1;  {print the document(s)}
```

PROCEDURE GetAppFiles (index: INTEGER; VAR theFile: AppFile);

GetAppFiles returns information about a document that was selected when your application was started up (as listed in the Finder information). The index parameter indicates the file for which information should be returned; it must be between 1 and the number returned by CountAppFiles, inclusive. The information is returned in the following data structure:

```
TYPE AppFile = RECORD
                 vRefNum: INTEGER; {volume reference number}
                 fType:   OSType;  {file type}
                 versNum: INTEGER; {version number}
                 fName:   Str255   {file name}
               END;
```

Volume reference number, file type, version number, and file name are discussed in the File Manager manual.

PROCEDURE ClrAppFiles (index: INTEGER);

ClrAppFiles changes the Finder information passed to your application about the specified file such that the Finder knows you've processed the file. The index parameter must be between 1 and the number returned by CountAppFiles, inclusive. You should call ClrAppFiles for every document your application opens or prints, so that the information returned by CountAppFiles and GetAppFiles is always correct. (ClrAppFiles sets the file type in the Finder information to Ø.)

```
PROCEDURE GetAppParms (VAR apName: STRING[31]; VAR apRefNum: INTEGER;
            VAR apParam: Handle);
```

GetAppParms returns information about the current application. It returns the application name in apName and the reference number for the application's resource file in apRefNum. A handle to the Finder information is returned in apParam, but the Finder information is more easily accessed with the GetAppFiles call.

```
PROCEDURE ExitToShell;
```

ExitToShell provides an exit from an application by starting up the Finder (after releasing the entire application heap).

Assembly-Language Routines

Notice that unlike most Operating System routines, a few of the Segment Loader routines are stack-based rather than register-based.

UnloadSeg procedure

```
Trap macro    _UnloadSeg

On entry      4(SP):  routine address
```

UnloadSeg unloads the segment containing the externally referenced routine at the specified address, making it relocatable and purgeable. The segment won't actually be purged until the memory it occupies is needed. If the segment is purged, the Segment Loader will reload it the next time one of the routines in it is called.

Chain procedure

```
Trap macro    _Chain

On entry       (AØ):  pointer to the application's file name
              4(AØ):  integer specifying the configuration
                      of the sound and screen buffers
```

Chain starts up an application without doing anything to the application heap, so the current application can let another application take over while still keeping its data around in the heap.

Chain also configures memory for the sound and screen buffers. The value you pass in 4(AØ) determines which sound and screen buffers are allocated:

- If you pass Ø in 4(AØ), you get the main sound and screen buffers; in this case, you have the largest amount of memory available to your application.
- Any positive value in 4(AØ) causes the alternate sound buffer and main screen buffer to be allocated.
- Any negative value in 4(AØ) causes the alternate sound buffer and alternate screen buffer to be allocated.

The memory map in the Memory Manager manual shows the locations of the screen and sound buffers. *** (The memory map will be in the next draft of the Memory Manager manual.) ***

(note)
You can get the most recent value passed in 4(AØ) to the Chain procedure from the global variable CurPageOption.

Chain closes the resource file for any previous application and opens the resource file for the application being started.

Launch procedure

Trap macro _Launch

On entry (AØ): pointer to the application's file name
4(AØ): integer specifying the configuration of the sound and screen buffers

Launch is called by the Finder to start up an application and will rarely need to be called by an application itself. It's the same as the Chain routine (described above) except that it frees the storage occupied by the application heap and restores the heap to its original size.

(note)
Launch preserves a special handle in the application heap which is used for preserving the desk scrap between applications; see the Scrap Manager manual for further information.

LoadSeg procedure

Trap macro _LoadSeg

On entry 24(SP): segment number (integer)

LoadSeg is called indirectly via the jump table (as described in the following section) when the application calls a routine in an unloaded segment. It loads the segment having the given segment number, which was assigned by the Linker. If the segment isn't in memory, LoadSeg calls the Resource Manager to read it in. It changes the jump table entries for all the routines in the segment from the "unloaded" to the "loaded" state and then invokes the routine that was called. Normally you'll never need to call LoadSeg.

ExitToShell procedure

Trap macro _ExitToShell

ExitToShell provides an exit from an application by starting up the Finder with the Launch procedure.

THE JUMP TABLE

This section describes how the Segment Loader works internally, and is included here for advanced programmers; you don't have to know about this to be able to use the common Segment Loader routines.

The loading and unloading of segments is implemented through the application's jump table. The jump table contains one eight-byte entry for every externally referenced routine in every segment; all the entries for a particular segment are stored contiguously. The location of the jump table is shown in the Memory Manager manual *** (in the next draft of it, not the current draft) ***.

When the Linker encounters a call to a routine in another segment, it creates a jump table entry for the routine (see Figure 3). The jump table refers to segments by segment numbers assigned by the Linker. If the segment is loaded, the jump table entry contains code that jumps to the routine. If the segment isn't loaded, the entry contains code that loads the segment.

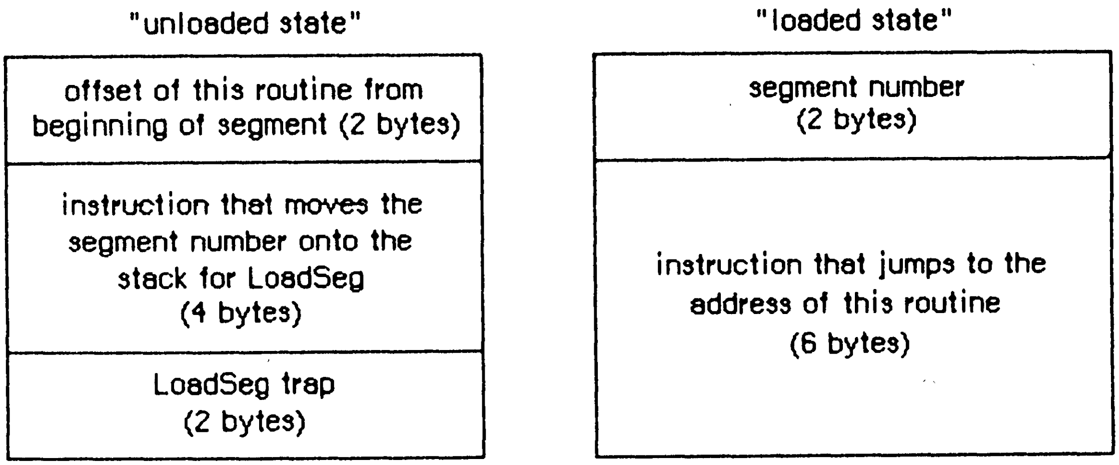

"unloaded state"
offset of this routine from beginning of segment (2 bytes)
instruction that moves the segment number onto the stack for LoadSeg (4 bytes)
LoadSeg trap (2 bytes)
"loaded state"
segment number (2 bytes)
instruction that jumps to the address of this routine (6 bytes)

Figure 3. Format of a Jump Table Entry

When a segment is unloaded, all its jump table entries are in the "unloaded" state. When a call to a routine in an unloaded segment is made, the code in the last six bytes of its jump table entry is executed. This code calls LoadSeg, which loads the segment into memory, transforms all of its jump table entries to the "loaded" state, and invokes the routine by executing the instruction in the last six bytes of the jump table entry. Subsequent calls to the routine also execute this instruction. If UnloadSeg is called to unload the segment, it restores the jump table entries to their "unloaded" state. Notice that whether the segment is loaded or unloaded, the last six bytes of the jump table entry are executed; the effect depends on the state of the entry at the time.

To be able to set all the jump table entries for a segment to a particular state, LoadSeg and UnloadSeg need to know exactly where in the jump table all the entries are located. They get this information from the segment header, four bytes at the beginning of the segment which contain the following:

Number of bytes	Contents
2 bytes	Offset of the first routine's entry from the beginning of the jump table
2 bytes	Number of entries for this segment

When an application is started up, its jump table is read in from segment Ø, a special segment created by the Linker for every executable file. Segment Ø contains the following:

Number of bytes	Contents
4 bytes	"Above A5" size; size in bytes from location pointed to by A5 to upper end of application space
4 bytes	"Below A5" size; size in bytes of application globals
4 bytes	Length of jump table in bytes
4 bytes	Offset to jump table from location pointed to by A5
n bytes	Jump table

For most applications, the offset to the jump table from the location pointed to by A5 is 32, and the "above A5" size is 32 plus the length of the jump table.

Assembly-language note: The offset to the jump table from the location pointed to by A5 is stored in the global variable CurJTOffset.

SUMMARY OF THE SEGMEɴT LOADER

Constants

```
CONST { Message returned by CountAppFiles }

      appOpen  = Ø;  {open the document(s)}
      appPrint = 1;  {print the document(s)}
```

Data Types

```
TYPE AppFile = RECORD
                 vRefNum: INTEGER; {volume reference number}
                 fType:   OSType;  {file type}
                 versNum: INTEGER; {version number}
                 fName:   Str255   {file name}
               END;
```

Routines

```
PROCEDURE UnloadSeg     (routineAddr: Ptr);
PROCEDURE CountAppFiles (VAR message: INTEGER; VAR count: INTEGER);
PROCEDURE GetAppFiles   (index: INTEGER; VAR theFile: AppFile);
PROCEDURE ClrAppFiles   (index: INTEGER);
PROCEDURE GetAppParms   (VAR apName: STRING[31]; VAR apRefNum: INTEGER;
                         VAR apParam: Handle);
PROCEDURE ExitToShell;
```

Assembly-Language Information

Routines

Chain procedure

```
      On entry     (AØ):  pointer to the application's file name
                  4(AØ):  integer specifying the configuration
                          of the sound and screen buffers
```

Launch procedure

```
      On entry     (AØ):  pointer to the application's file name
                  4(AØ):  integer specifying the configuration
                          of the sound and screen buffers
```

LoadSeg procedure

<u>On entry</u> 24(SP): segment number (integer)

UnloadSeg procedure

<u>On entry</u> 4(SP): routine address

ExitToShell procedure

<u>Variables</u>

Name	Size	Contents
AppParmHandle	4 bytes	Handle to Finder information
CurApName	32 bytes	Name of current application
CurApRefNum	2 bytes	Reference number of current application's resource file
CurPageOption	2 bytes	Most recent value passed in 4(A∅) to Chain procedure
CurJTOffset	2 bytes	Offset to jump table from location pointed to by A5

GLOSSARY

application parameters: Parameters passed to an application when it's started up, in 32 bytes of memory just above the application globals; primarily, a handle to the Finder information.

Finder information: Information that the Finder provides to an application upon starting it up, telling it which files to open or print.

jump table: A table that contains one entry for every routine in an application and is the means by which the loading and unloading of segments is implemented.

main segment: The segment containing the main program.

segment: One of several parts into which the code of an application may be divided. Not all segments need to be in memory at the same time.

MACINTOSH USER EDUCATION

The Operating System Event Manager: A Programmer's Guide

/OSEMGR/EVENTS

See Also: Inside Macintosh: A Road Map
Macintosh User Interface Guidelines
Macintosh Memory Management: An Introduction
Programming Macintosh Applications in Assembly Language
The Toolbox Event Manager: A Programmer's Guide
The Operating System Utilities: A Programmer's Guide

Modification History: First Draft Caroline Rose & Brent Davis 11/19/84

ABSTRACT

This manual describes the Operating System Event Manager, the part of the Macintosh Operating System that reports low-level user actions such as mouse-button presses and keystrokes. Usually your application will find out about events by calling the Toolbox Event Manager, which calls the Operating System Event Manager for you, but in some situations you'll need to call the Operating System Event Manager directly.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Operating System Event Manager, the part of the Macintosh Operating System that reports low-level user actions such as mouse-button presses and keystrokes. *** Eventually it will become part of the comprehensive Inside Macintosh manual. *** Usually your application will find out about events by calling the Toolbox Event Manager, which calls the Operating System Event Manager for you, but in some situations you'll need to call the Operating System Event Manager directly.

(note)
All references to "the Event Manager" in this manual refer to the Operating System Event Manager.

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal and the information in the following manuals:

- Inside Macintosh: A Road Map
- Macintosh Memory Management: An Introduction
- Programming Macintosh Applications in Assembly Language, if you're using assembly language

You should also be familiar with the Toolbox Event Manager.

(note)
Constants and data types defined in the Operating System Event Manager are presented in detail in the Toolbox Event Manager manual, since they're necessary for using that part of the Toolbox. They're also listed in the summary of this manual.

ABOUT THE OPERATING SYSTEM EVENT MANAGER

The Event Manager is the part of the Operating System that detects low-level, hardware-related events: mouse, keyboard, disk-inserted, device driver, and network events. It stores information about these events in the event queue and provides routines that access the queue (analogous to GetNextEvent and EventAvail in the Toolbox Event Manager). It also allows your application to post its own events into the event queue. Like the Toolbox Event Manager, the Operating System Event Manager returns a null event if it has no other events to report.

The Toolbox Event Manager calls the Operating System Event Manager to retrieve events from the event queue; in addition, it reports activate and update events, which aren't kept in the queue. It's extremely unusual for an application not to have to know about activate and update events, so usually you'll call the Toolbox Event Manager to get events.

The Operating System Event Manager also lets you:

- remove events from the event queue
- set the system event mask, to control which types of events get posted into the queue

USING THE OPERATING SYSTEM EVENT MANAGER

If you're using application-defined events in your program, you'll need to call the Operating System Event Manager function PostEvent to post them into the event queue. This function is sometimes also useful for reposting events that you've removed from the event queue with GetNextEvent.

In some situations you may want to remove from the event queue some or all events of a certain type or types. You can do this with the procedure FlushEvents. A common use of FlushEvents is to get rid of any stray events left over from before your application was started up.

You'll probably never call the other Operating System Event Manager routines: GetOSEvent, which gets an event from the event queue, removing it from the queue in the process; OSEventAvail, for looking at an event without dequeueing it; and SetEventMask, which changes the setting of the system event mask.

OPERATING SYSTEM EVENT MANAGER ROUTINES

Posting and Removing Events

FUNCTION PostEvent (eventCode: INTEGER; eventMsg: LONGINT): OSErr;

Trap macro	_PostEvent	
On entry	AØ:	eventCode (word)
	DØ:	eventMsg (long word)
On exit	DØ:	result code (word)

PostEvent places in the event queue an event of the type designated by eventCode, with the event message specified by eventMsg and with the current time, mouse location, and state of the modifier keys and mouse button. It returns a result code (of type OSErr, defined as INTEGER in the Operating System Utilities) equal to one of the following predefined constants:

```
CONST noErr     = Ø;    {no error (event posted)}
      evtNotEnb = 1;    {event type not designated in system }
                        { event mask}
```

(warning)

> Be very careful when posting any events other than your own application-defined events into the queue; attempting to post an activate or update event, for example, will interfere with the internal operation of the Toolbox Event Manager, since such events aren't normally placed in the queue at all.
>
> If you use PostEvent to repost an event, remember that the event time, location, and state of the modifier keys and mouse button will all be changed from their values when the event was originally posted, possibly altering the meaning of the event.

PROCEDURE FlushEvents (eventMask,stopMask: INTEGER);

Trap macro	_FlushEvents		
On entry	DØ:	low-order word: high-order word:	eventMask stopMask
On exit	DØ:	Ø or event code (word)	

FlushEvents removes events from the event queue as specified by the given event masks. It removes all events of the type or types specified by eventMask, up to but not including the first event of any type specified by stopMask; if the event queue doesn't contain any events of the types specified by eventMask, it does nothing. To remove all events specified by eventMask, use a stopMask value of Ø.

At the beginning of your application, 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.

Assembly-language note: On exit from this routine, DØ contains Ø if all events were removed from the queue or, if not, an event code specifying the type of event that caused the removal process to stop.

Accessing Events

FUNCTION GetOSEvent (eventMask: INTEGER; VAR theEvent: EventRecord) :
         BOOLEAN;

Trap macro	_GetOSEvent
On entry	AØ: pointer to event record theEvent DØ: eventMask (word)
On exit	DØ: Ø if non-null event returned, or -1 if null event returned (byte)

GetOSEvent 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. The eventMask parameter specifies which event types are of interest. GetOSEvent will return the next available event of any type designated by the mask. If no event of any of the designated types is available, GetOSEvent returns a null event and a function result of FALSE; otherwise it returns TRUE.

FUNCTION OSEventAvail (eventMask: INTEGER; VAR theEvent: EventRecord) :
         BOOLEAN;

Trap macro	_OSEventAvail
On entry	AØ: pointer to event record theEvent DØ: eventMask (word)
On exit	DØ: Ø if non-null event returned, or -1 if null event returned (byte)

OSEventAvail works exactly the same as GetOSEvent (above) except that it doesn't remove the event from the event queue.

(note)

> An event returned by OSEventAvail will not be accessible later if in the meantime the queue becomes full and the event is discarded from it; since the events discarded are always the oldest ones in the queue, however, this will happen only in an unusually busy environment.

Setting the System Event Mask

PROCEDURE SetEventMask (theMask: INTEGER); [No trap macro]

SetEventMask sets the system event mask to the specified event mask. The Operating System Event Manager will post only those event types that correspond to bits set in the mask. (As usual, it will not post activate and update events, which are generated by the Window Manager and not stored in the event queue.) The system event mask is initially set to post all except key-up events.

(warning)

> Because desk accessories may rely on receiving certain types of events, your application shouldn't set the system event mask to prevent any additional types (besides key-up) from being posted. You should use SetEventMask only to enable key-up events in the unusual case that your application needs to respond to them.

> Assembly-language note: The system event mask is available to assembly-language programmers in the global variable SysEvtMask.

STRUCTURE OF THE EVENT QUEUE

The event queue is a standard Macintosh Operating System queue, as described in the Operating System Utilities manual. Most programmers will never need to access the event queue directly; some advanced programmers, though, may need to do so for special purposes.

Each entry in the event queue contains information about an event:

```
TYPE EvQEl = RECORD
                qLink:        QElemPtr;  {next queue entry}
                qType:        INTEGER;   {queue type}
                evQWhat:      INTEGER;   {event code}
                evQMessage:   LONGINT;   {event message}
                evQWhen:      LONGINT;   {ticks since startup}
                evQWhere:     Point;     {mouse location}
                evQModifiers: INTEGER    {modifier flags}
             END;
```

QLink points to the next entry in the queue, and qType indicates the queue type, which must be ORD(evType). The remaining five fields of the event queue element contain exactly the same information about the event as do the fields of the event record for that event; see the Toolbox Event Manager manual for a detailed description of the specific contents of these fields.

You can get a pointer to the event queue by calling the Operating System Event Manager function GetEvQHdr.

FUNCTION GetEvQHdr : QHdrPtr; [No trap macro]

GetEvQHdr returns a pointer to the event queue.

Assembly-language note: To access the contents of the event queue from assembly language, you can use offsets from the address of the global variable EventQueue.

DEFINING A NONSTANDARD KEYBOARD CONFIGURATION

*** This information is forthcoming ***

SUMMARY OF THE OPERATING SYSTEM EVENT MANAGER

Constants

```
CONST { Event codes }

      mouseDown    = 1;   {mouse-down}
      mouseUp      = 2;   {mouse-up}
      keyDown      = 3;   {key-down}
      keyUp        = 4;   {key-up}
      autoKey      = 5;   {auto-key}
      updateEvt    = 6;   {update; Toolbox only}
      diskEvt      = 7;   {disk-inserted}
      activateEvt  = 8;   {activate; Toolbox only}
      networkEvt   = 1Ø;  {network}
      driverEvt    = 11;  {device driver}
      app1Evt      = 12;  {application-defined}
      app2Evt      = 13;  {application-defined}
      app3Evt      = 14;  {application-defined}
      app4Evt      = 15;  {application-defined}

      { Masks for accessing keyboard event message }

      charCodeMask = $ØØØØØØFF;  {character code}
      keyCodeMask  = $ØØØØFFØØ;  {key code}

      { Masks for forming event mask }

      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}
      networkMask  = 1Ø24;    {network}
      driverMask   = 2Ø48;    {device driver}
      app1Mask     = 4Ø96;    {application-defined}
      app2Mask     = 8192;    {application-defined}
      app3Mask     = 16384;   {application-defined}
      app4Mask     = -32768;  {application-defined}
      everyEvent   = -1;      {all event types}
```

```
{ Modifier flags in event record }

activeFlag = 1;     {set if window being activated}
btnState   = 128;   {set if mouse button up}
cmdKey     = 256;   {set if Command key down}
shiftKey   = 512;   {set if Shift key down}
alphaLock  = 1Ø24;  {set if Caps Lock key down}
optionKey  = 2Ø48;  {set if Option key down}

{ Result codes returned by PostEvent }

noErr      =  Ø;     {no error (event posted)}
evtNotEnb  =  1;     {event type not designated in system event }
                     { mask}
```

Data Types

```
TYPE EventRecord = RECORD
                     what:       INTEGER;    {event code}
                     message:    LONGINT;    {event message}
                     when:       LONGINT;    {ticks since startup}
                     where:      Point;      {mouse location}
                     modifiers:  INTEGER     {modifier flags}
                   END;

     EvQEl = RECORD
               qLink:        QElemPtr;  {next queue entry}
               qType:        INTEGER;   {queue type}
               evQWhat:      INTEGER;   {event code}
               evQMessage:   LONGINT;   {event message}
               evQWhen:      LONGINT;   {ticks since startup}
               evQWhere:     Point;     {mouse location}
               evQModifiers: INTEGER    {modifier flags}
             END;
```

Routines

Posting and Removing Events

```
FUNCTION  PostEvent   (eventCode: INTEGER; eventMsg: LONGINT) : OSErr;
PROCEDURE FlushEvents (eventMask,stopMask: INTEGER);
```

Accessing Events

```
FUNCTION GetOSEvent   (eventMask: INTEGER; VAR theEvent: EventRecord) :
                       BOOLEAN;
FUNCTION OSEventAvail (eventMask: INTEGER; VAR theEvent: EventRecord) :
                       BOOLEAN;
```

Setting the System Event Mask

```
PROCEDURE SetEventMask (theMask: INTEGER);  [No trap macro]
```

Directly Accessing the Event Queue

```
FUNCTION GetEvQHdr : QHdrPtr;  [No trap macro]
```

Assembly-Language Information

Constants

```
; Event codes

nullEvt        .EQU  Ø   ;null
mButDwnEvt     .EQU  1   ;mouse-down
mButUpEvt      .EQU  2   ;mouse-up
keyDwnEvt      .EQU  3   ;key-down
keyUpEvt       .EQU  4   ;key-up
autoKeyEvt     .EQU  5   ;auto-key
updatEvt       .EQU  6   ;update; Toolbox only
diskInsertEvt  .EQU  7   ;disk-inserted
activateEvt    .EQU  8   ;activate; Toolbox only
networkEvt     .EQU  1Ø  ;network
ioDrvrEvt      .EQU  11  ;device driver
app1Evt        .EQU  12  ;application-defined
app2Evt        .EQU  13  ;application-defined
app3Evt        .EQU  14  ;application-defined
app4Evt        .EQU  15  ;application-defined

; Modifier flags in event record

activeFlag     .EQU  Ø   ;set if window being activated
btnState       .EQU  2   ;set if mouse button up
cmdKey         .EQU  3   ;set if Command key down
shiftKey       .EQU  4   ;set if Shift key down
alphaLock      .EQU  5   ;set if Caps Lock key down
optionKey      .EQU  6   ;set if Option key down

; Result codes returned by PostEvent

noErr          .EQU   Ø  ;no error (event posted)
evtNotEnb      .EQU   1  ;event type not designated in system event
                         ; mask
```

Event Record Data Structure

evtNum	Event code
evtMessage	Event message
evtTicks	Ticks since startup
evtMouse	Mouse location
evtMeta	State of modifier keys
evtMBut	State of mouse button
evtBlkSize	Length of above structure

Event Queue Element Data Structure

qLink	Pointer to next queue entry
qType	Queue type
evtQWhat	Event code
evtQMessage	Event message
evtQWhen	Ticks since startup
evtQWhere	Mouse location
evtQMeta	State of modifier keys
evtQMBut	State of mouse button
evtQBlkSize	Length of above structure

Routines

Name	On entry	On exit
PostEvent	AØ: eventCode (word) DØ: eventMsg (long)	DØ: result code (word) = Ø if no error, 1 if event type not designated in system event mask
FlushEvents	DØ: low-order word: eventMask high-order word: stopMask	DØ: Ø or event code (word)
GetOSEvent and OSEventAvail	AØ: ptr to event record theEvent DØ: eventMask (word)	DØ: Ø if non-null event, -1 if null event (byte)

Variables

Name	Size	Contents
SysEvtMask	2 bytes	System event mask
EventQueue	1Ø bytes	Event queue

GLOSSARY

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 repeatedly when the user presses and holds down a character key on the keyboard or keypad.

device driver event: An event generated by one of the Macintosh's device drivers.

disk-inserted event: An event generated when the user inserts a disk in a disk drive or takes any other action that requires a volume to be mounted.

event: A notification to an application of some occurrence that the application may want to respond to.

event code: An integer representing a particular type of event.

event mask: A parameter passed to a Toolbox or Operating System Event Manager routine to specify which types of event the routine should apply to.

event message: A field of an event record containing information specific to the particular type of event.

event queue: The Operating System Event Manager's list of pending events.

event record: The internal representation of an event, through which your program learns all pertinent information about that event.

key-down event: An event generated when the user presses a character key on the keyboard or keypad.

key-up event: An event generated when the user releases a character key on the keyboard or keypad.

keyboard event: An event generated when the user presses, releases, or holds down a character key on the keyboard or keypad; any key-down, key-up, or auto-key event.

modifier key: A key (Shift, Caps Lock, Option, or Command) that generates no keyboard events of its own, but changes the meaning of other keys or mouse actions.

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 AppleBus Manager.

null event: An event reported when there are 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 File Manager: A Programmer's Guide /OS/FS

See Also: The Macintosh User Interface Guidelines
The Memory Manager: A Programmer's Guide
Inside Macintosh: A Road Map
Macintosh Packages: A Programmer's Guide
The Structure of a Macintosh Application
Programming Macintosh Applications in Assembly Language

Modification History: First Draft (ROM 7) Bradley Hacker 5/21/84

ABSTRACT

This manual describes the File Manager, the part of the Macintosh Operating System that controls the exchange of information between a Macintosh application and files.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the File Manager, the part of the Macintosh Operating System that controls the exchange of information between a Macintosh application and files. *** Eventually it will become part of the comprehensive Inside Macintosh manual. *** The File Manager allows you to create and access any number of files containing whatever information you choose.

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal. You should also be familiar with the following:

- the basic concepts behind the Macintosh Operating System's Memory Manager

- devices and device drivers, as described in the Inside Macintosh Road Map

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 File Manager and what you can do with it. It then discusses some basic concepts behind the File Manager: what files and volumes are and how they're accessed.

A section on using the File Manager introduces its routines and tells how they fit into the flow of your application. This is followed by sections explaining the File Manager's simplest, "high-level" Pascal routines and then its more complex, "low-level" Pascal and assembly-language routines. Both sections give detailed descriptions of all the procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that won't interest all readers. The data structures that the File Manager uses to store information in memory and on disks are described, and special information is provided for programmers who want to write their own file system.

Finally, there's a summary of the File Manager, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE FILE MANAGER

The File Manager is the part of the Operating System that handles communication between an application and files on block devices such as disk drives. Files are a principal means by which data is stored and transmitted on the Macintosh. A file is a named, ordered sequence of bytes. The File Manager contains routines used to read and write to files.

Volumes

A volume is a piece of storage medium, such as a disk, formatted to contain files. A volume can be an entire disk or only part of a disk. Currently, the 3 1/2-inch Macintosh disks are one volume.

(note)

> Specialized memory devices other than disks can also contain volumes, but the information in this manual applies only to volumes on disks.

You identify a volume by its volume name, which consists of any sequence of 1 to 27 printing characters. Volume names must always be followed by a colon (:) to distinguish them from other names. You can use uppercase and lowercase letters when naming volumes, but the File Manager ignores case when comparing names (it doesn't ignore diacritical marks).

(note)

> The colon (:) after a volume name should only be used when calling File Manager routines; it should never be seen by the user.

A volume contains descriptive information about itself, including its name and a file directory listing information about files contained on the volume; it also contains files. The files are contained in allocation blocks, which are areas of volume space occupying multiples of 512 bytes.

A volume can be mounted or unmounted. A volume becomes mounted when it's in a disk drive and the File Manager reads descriptive information about the volume into memory. Once mounted, a volume may remain in a drive or be ejected. Only mounted volumes are known to the File Manager, and an application can access information on mounted volumes only. A volume becomes unmounted when the File Manager releases the memory used to store the descriptive information. Your application should unmount a volume when it's finished with the volume, or when it needs the memory occupied by the volume.

The File Manager assigns each mounted volume a volume reference number that you can use instead of its volume name to refer to it. Every mounted volume is also assigned a volume buffer, which is temporary storage space on the heap used when reading and writing information on the volume. The number of volumes that may be mounted at any time is limited only by the number of drives attached and available memory.

A mounted volume can be on-line or off-line. A mounted volume is on-line as long as the volume buffer and all the descriptive information read from the volume when it was mounted remain in memory (about 1K to 1.5K bytes); it becomes off-line when all but 94 bytes of descriptive information are released. You can access information on on-line volumes immediately, but off-line volumes must be placed on-line before their information can be accessed. An application should place a volume off-line whenever it needs most of the memory the volume occupies. When an application ejects a volume from a drive, the File Manager automatically places the volume off-line.

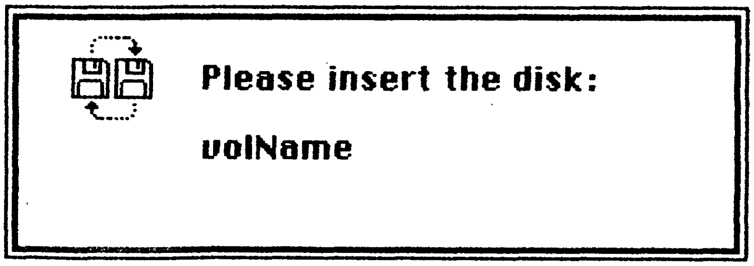
To prevent unauthorized writing to a volume, volumes can be locked. Locking a volume involves either setting a software flag on the volume or changing some part of the volume physically (for example, sliding a tab from one position to another on a disk). Locking a volume ensures that none of the data on the volume can be changed.

Accessing Volumes

You can access a mounted volume via its volume name or volume reference number. On-line volumes in disk drives can also be accessed via the drive number of the drive on which the volume is mounted (the internal drive is number 1, the external drive is number 2, and any additional drives connected via a serial port will have larger numbers). When accessing a mounted volume, you should always use the volume name or volume reference number, rather than a drive number, because the volume may have been ejected or placed off-line. Whenever possible, use the volume reference number (to avoid confusion between volumes with the same name).

One volume is always the default volume. Whenever you call a routine to access a volume but don't specify which volume, the default volume is accessed. Initially, the volume used to start up the system is the default volume, but an application can designate any mounted volume as the default volume.

Whenever the File Manager needs to access a mounted volume that's been ejected from its drive, the dialog box shown in Figure 1 is displayed, and the File Manager waits until the user inserts the volume named volName into a drive.



Please insert the disk:

volName

Figure 1. Disk-Switch Dialog

Files

A file is a finite sequence of numbered bytes. Any byte or group of bytes in the sequence can be accessed individually. A file is identified by its file name and version number. A file name consists of any sequence of 1 to 255 printing characters, excluding colons (:). You can use uppercase and lowercase letters when naming volumes, but the File Manager ignores case when comparing names (it doesn't ignore diacritical marks). The version number is any number from Ø to 255, and is used by the File Manager to distinguish between different files with the same name. A byte within a file is identified by its position within the ordered sequence.

(warning)

> Your application should constrain file names to fewer than 64 characters, because the Finder will generate an error if given a longer name. You should always assign files a version number of Ø, because the Resource Manager and Segment Loader won't operate on files with nonzero file numbers, the Finder ignores version numbers, and the Standard File Package clears version numbers.

There are two parts or forks to a file: the data fork and the resource fork. Normally the resource fork of an application file contains the resources used by the application such as menus, fonts, and icons, and also the application code itself. The data fork can contain anything an application wants to store there. Information stored in resource forks should always be accessed via the Resource Manager. Information in data forks can only be accessed via the File Manager. For simplicity, "file" will be used instead of "data fork" in this manual.

A file can contain anywhere from Ø to 16,777,216 bytes (16 megabytes). Each byte is numbered: the first byte is byte Ø. You can read bytes from and write bytes to a file either singly or in sequences of unlimited length. Each read or write operation can start anywhere in the file, regardless of where the last operation began or ended. Figure 2 shows the structure of a file.

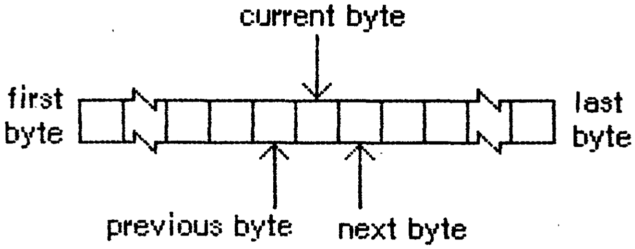


Figure 2. A File

A file's maximum size is defined by its physical end-of-file, which is 1 greater than the number of the last byte in its last allocation block (Figure 3). The physical end-of-file is equivalent to the maximum number of bytes the file can contain. A file's actual size is defined by its logical end-of-file, which is 1 greater than the number of the last byte in the file. The logical end-of-file is equivalent to the actual number of bytes in the file, since the first byte is byte number Ø. The physical end-of-file is always greater than the logical end-of-file. For example, an empty file (one with Ø bytes) in a 1K-byte allocation block has a logical end-of-file of Ø and a physical end-of-file of 1Ø24. A file with 5Ø bytes has a logical end-of-file of 5Ø and a physical end-of-file of 1Ø24.

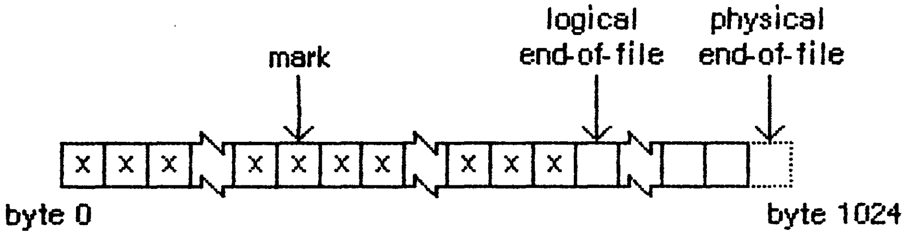
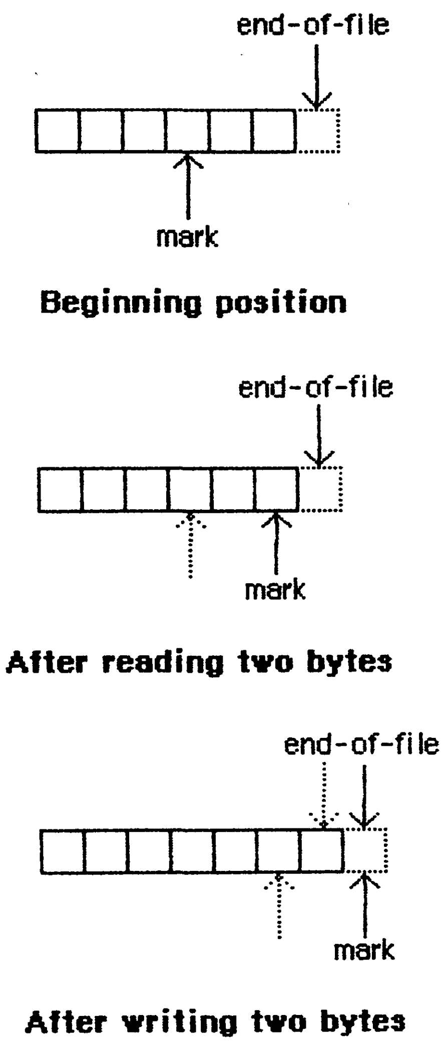


Figure 3. End-of-File and Mark

The current position marker, or mark, is the number of the next byte that will be read or written. The value of the mark can't exceed the value of the logical end-of-file. The mark automatically moves forward one byte for every byte read from or written to the file. If, during a write operation, the mark meets the logical end-of-file, both are moved forward one position for every additional byte written to the file. Figure 4 shows the movement of the mark and logical end-of-file.



Beginning position

After reading two bytes

After writing two bytes

Figure 4. Movement of Logical End-of-File and Mark

If, during a write operation, the mark must move past the physical end-of-file, another allocation block is added to the file--the physical end-of-file is placed one byte beyond the end of the new allocation block, and the mark and logical end-of-file are placed at the first byte of the new allocation block.

An application can move the logical end-of-file to anywhere from the beginning of the file to the physical end-of-file (the mark is adjusted accordingly). If the logical end-of-file is moved to a position more than one allocation block short of the current physical end-of-file, the unneeded allocation block will be deleted from the file. The mark can be placed anywhere from the first byte in the file to the logical end-of-file.

Accessing Files

A file can be open or closed. An application can only perform certain operations, such as reading and writing, on open files; other operations, such as deleting, can only be performed on closed files.

To open a file, you must identify the file and the volume containing it. When a file is opened, the File Manager creates an access path, a description of the route to be followed when accessing the file. The access path specifies the volume on which the file is located (by volume reference number, drive number, or volume name) and the location of the file on the volume. Every access path is assigned a unique path reference number used to refer to it. You should always refer to a file via its path reference number, so that files with the same name aren't confused with one another.

A file can have one access path open for writing or for both reading and writing, and one or more access paths for reading only; there cannot be more than one access path that writes to a file. Each access path is separate from all other access paths to the file. A maximum of 12 access paths can be open at one time. Each access path can move its own mark and read at the position it indicates. All access paths to the same file share common logical and physical end-of-file markers.

The File Manager reads descriptive information about a newly opened file from its volume and stores it in memory. For example, each file has open permission information, which indicates whether data can only be read from it, or both read from and written to it. Each access path contains read/write permission information that specifies whether data is allowed to be read from the file, written to the file, both read and written, or whatever the file's open permission allows. If an application wants to write data to a file, both types of permission information must allow writing; if either type allows reading only, then no data can be written.

When an application requests that data be read from a file, the File Manager reads the data from the file and transfers it to the application's data buffer. Any part of the data that can be transferred in entire 512-byte blocks is transferred directly. Any part of the data composed of fewer than 512 bytes is also read from the file in one 512-byte block, but placed in temporary storage space in memory. Then, only the bytes containing the requested data are transferred to the application.

When an application writes data to a file, the File Manager transfers the data from the application's data buffer and writes it to the file. Any part of the data that can be transferred in entire 512-byte blocks is written directly. Any part of the data composed of fewer than 512 bytes is placed in temporary storage space in memory until 512 bytes have accumulated; then the entire block is written all at once.

Normally the temporary space in memory used for all reading and writing is the volume buffer, but an application can specify that an access path buffer be used instead for a particular access path (Figure 5).

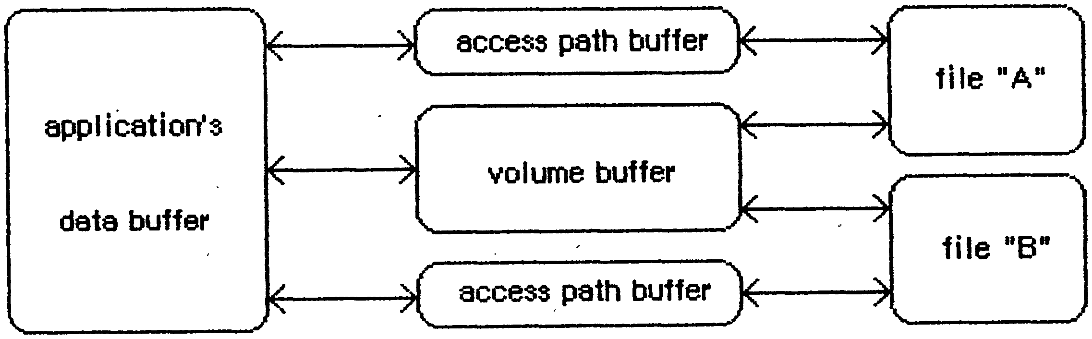


Figure 5. Buffers For Transferring Data

(warning)

You must lock every access path buffer you use, so its location doesn't change while the file is open.

Your application can lock a file to prevent unauthorized writing to it. Locking a file ensures that none of the data in it can be changed *** Currently, the Finder won't let you rename or delete a locked file, but it will let you change the data the file contains ***.

(note)

Advanced programmers: The File Manager can also read a continuous stream of characters or a line of characters. In the first case, you ask the File Manager to read a specific number of bytes: when that many have been read or when the mark has reached the logical end-of-file, the read operation terminates. In the second case, called newline mode, the read will terminate when either of the above conditions is fulfilled or when a specified character, the newline character, is read. The newline character is usually Return (ASCII code $ØD), but can be any character whose ASCII code is between $ØØ and $FF, inclusive. Information about newline mode is associated with each access path to a file, and can differ from one access path to another.

FILE INFORMATION USED BY THE FINDER

A file directory on a volume lists information about all the files on the volume. The information used by the Finder is contained in a data structure of type FInfo:

```
TYPE FInfo = RECORD
               fdType:     OSType;  {type of file}
               fdCreator:  OSType;  {file's creator}
               fdFlags:    INTEGER; {flags}
               fdLocation: Point;   {file's location}
               fdFldr:     INTEGER  {file's window}
             END;
```

Normally an application need only set the file type and creator when a file is created, and the Finder will manipulate the other fields. (File type and creator are discussed in The Structure of a Macintosh Application.) Advanced programmers may be interested in changing the contents of the other fields as well.

FdFlags indicates whether the file's icon is invisible, whether the file has a bundle, and other characteristics used internally by the Finder:

Bit	Meaning if set
5	File has a bundle
6	File's icon is invisible

Masks for these two bits are available as predefined constants:

```
CONST fHasBundle = 32; {set if file has a bundle}
      fInvisible = 64; {set if file's icon is invisible}
```

When you first install an application, you'll need to set its "bundle bit", as described in The Structure of a Macintosh Application. Whenever you create a file with a bundle, you'll need to set its bundle bit.

The next two fields indicate where the file's icon will appear if the icon is visible. FdLocation contains the location of the file's icon in its window, given in the local coordinate system of the window. FdFldr indicates the window in which the file's icon will appear, and may contain one of the following predefined constants:

```
CONST fTrash   = -3; {file is in trash window}
      fDesktop = -2; {file is on desktop}
      fDisk    =  Ø; {file is in disk window}
```

If fdFldr contains a positive number, the file's icon will appear in a folder; the numbers that identify folders are assigned by the Finder. Advanced programmers can get the folder number of an existing file, and place additional files in that same folder.

USING THE FILE MANAGER

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

You can call File Manager routines via three different methods: high-level Pascal calls, low-level Pascal calls, and assembly language. The high-level Pascal calls are designed for Pascal programmers interested in using the File Manager in a simple manner; they provide adequate file I/O and don't require much special knowledge to use. The low-level Pascal and assembly-language calls are designed for advanced Pascal programmers and assembly-language programmers interested in using the File Manager to its fullest capacity; they require some special knowledge to be used most effectively.

Information for all programmers follows here. The next two sections contain special information for high-level Pascal programmers and for low-level Pascal and assembly-language programmers.

(note)

> The names used to refer to routines here are actually the assembly-language macro names for the low-level routines, but the Pascal routine names are very similar.

The File Manager is automatically initialized each time the system is started up.

To create a new, empty file, call Create. Create allows you to set some of the information stored on the volume about the file.

To open a file, call Open. The File Manager creates an access path and returns a path reference number that you'll use every time you want to refer to it. Before you open a file, you may want to call the Standard File Package, which presents the standard interface through which the user can specify the file to be opened. The Standard File Package will return the name of the file, the volume reference number of the volume containing the file, and additional information. (If the user inserts an unmounted volume into a drive, the Standard File Package will automatically call the Disk Initialization Package to attempt to mount it.)

After opening a file, you can transfer data from it to an application's data buffer with Read, and send data from an application's data buffer to the file with Write. Read and Write allow you to specify a byte position within the data buffer, a number of bytes to transfer, and the location within the file. You can't use Write on a file whose open permission only allows reading, or on a file on a locked volume.

Once you've completed whatever reading and writing you want to do, call Close to close the file. Close writes the contents of the file's access path buffer to the volume and deletes the access path. You can remove a closed file (both forks) from a volume by calling Delete.

To protect against power loss or unexpected disk ejection, you should periodically call FlushVol (probably after each time you close a file), which writes the contents of the volume buffer and all access path buffers (if any) to the volume and updates the descriptive information contained on the volume.

Whenever your application is finished with a disk, or the user chooses Eject from a menu, call Eject. Eject calls FlushVol, places the volume off-line, and then physically ejects the volume from its drive.

The preceding paragraphs covered the simplest File Manager routines: Open, Read, Write, Close, FlushVol, Eject, and Create. The remainder of this section describes the less commonly used routines, some of which are available only to advanced programmers. Skip the remainder of this section if the preceding paragraphs have provided you with all the information you want to know about using the File Manager.

When the Toolbox Event Manager function GetNextEvent receives a disk-inserted event, it calls the Desk Manager function SystemEvent. SystemEvent calls the File Manager function MountVol, which attempts to mount the volume on the disk. GetNextEvent then returns the disk-inserted event: the low-order word of the event message contains the number of the drive, and the high-order word contains the result code of the attempted mounting. If the result code indicates that an error occurred, you'll need to call the Disk Initialization Package to allow the user to initialize or eject the volume.

(note)

> Applications that rely on the Operating System Event Manager function GetOSEvent to learn about events (and don't call GetNextEvent) must explicitly call MountVol to mount volumes.

After a volume has been mounted, your application can call GetVolInfo, which will return the name of the volume, the amount of unused space on the volume, and a volume reference number that you can use every time you refer to that volume.

To minimize the amount of memory used by mounted volumes, an application can unmount or place off-line any volumes that aren't currently being used. To unmount a volume, call UnmountVol, which flushes a volume (by calling FlushVol) and releases all of the memory used for it (releasing about 1 to 1.5K bytes). To place a volume off-line, call OffLine, which flushes a volume (by calling FlushVol) and releases all of the memory used for it except for 94 bytes of descriptive information about the volume. Off-line volumes are placed on-line by the File Manager as needed, but your application must remount any unmounted volumes it wants to access. The File Manager itself may place volumes off-line during its normal operation.

If you would like all File Manager calls to apply to one volume, you can specify that volume as the default. You can use SetVol to set the default volume to any mounted volume, and GetVol to learn the name and volume reference number of the default volume.

Normally, volume initialization and naming is handled by the Standard File Package, which calls the Disk Initialization Package. If you want to initialize a volume explicitly or erase all files from a volume, you can call the Disk Initialization Package directly. When you want to change the name of a volume, call the File Manager function Rename.

Applications normally will use the Resource Manager to open resource forks and change the information contained within, but programmers writing unusual applications (such as a disk-copying utility) might want to use the File Manager to open resource forks. This is done by calling OpenRF. As with Open, the File Manager creates an access path and returns a path reference number that you'll use every time you want to refer to this resource fork.

As an alternative to specifying byte positions within a file with Read and Write, you can specify the byte position of the mark by calling SetFPos. GetFPos returns the byte position of the mark.

Whenever a disk has been reconstructed in an attempt to salvage lost files (because its directory or other file-access information has been destroyed), the logical end-of-file of each file will probably be equal to each physical end-of-file, regardless of where the actual logical end-of-file is. The first time an application attempts to read from a file on a reconstructed volume, it will blindly pass the correct logical end-of-file and read misinformation until it reaches the new, incorrect logical end-of-file. To prevent this from occurring, an application should always maintain an independent record of the logical end-of-file of each file it uses. To determine the File Manager's conception of the length of a file, or find out how many bytes have yet to be read from it, call GetEOF, which returns the logical end-of-file. You can change the length of a file by calling SetEOF.

Allocation blocks are automatically added to and deleted from a file as necessary. If this happens to a number of files alternately, each of the files will be contained in allocation blocks scattered throughout the volume, which increases the time required to access those files. To prevent such fragmentation of files, you can allocate a number of contiguous allocation blocks to an open file by calling Allocate.

Instead of calling FlushVol, an unusual application might call FlushFile. FlushFile forces the contents of a file's volume buffer and access path buffer (if any) to be written to its volume. FlushFile doesn't update the descriptive information contained on the volume, so the volume information won't be correct until you call FlushVol.

To get information about a file (such as its name and creation date) stored on a volume, call GetFileInfo. You can change this information by calling SetFileInfo. Changing the name or version number of a file is accomplished by calling Rename or SetFilType, respectively; they will have a similar effect, since both the file name and version number are needed to identify a file. You can lock or unlock a file by calling SetFilLock or RstFilLock, respectively.

You can't use Write, Allocate, or SetEOF on a locked file, a file whose open permission only allows reading, or a file on a locked volume. You can't use Rename or SetFilType on a file on a locked volume.

HIGH-LEVEL FILE MANAGER ROUTINES

This section describes all the high-level Pascal routines of the File Manager. Assembly-language programmers cannot call these routines. For information on calling the low-level Pascal and assembly-language routines, see the next section.

When accessing a volume, you must identify it by its volume name, its volume reference number, or the drive number of its drive--or allow the default volume to be accessed. The parameter names used in identifying a volume are volName, vRefNum, and drvNum. VRefNum and drvNum are both integers. VolName is a pointer, of type StringPtr, to a volume name.

The File Manager determines which volume to access by using one of the following:

1. VolName. (If volName points to a zero-length name, an error is returned.)

2. If volName is NIL or points to an improper volume name, then vRefNum or drvNum (only one is given per routine).

3. If vRefNum or drvNum is zero, the default volume. (If there isn't a default volume, an error is returned.)

(warning)

> Before you pass a parameter of type StringPtr to a File Manager routine such as GetVol, be sure that memory has been allocated for the variable. For example, the following statements will ensure that memory is allocated for the variable myStr:

```
VAR myStr: Str255;
. . .
BEGIN
    result := GetVol(@myStr, myRefNum);
    . . .
END;
```

When accessing a closed file on a volume, you must identify the volume by the method given above, and identify the file by its name in the fileName parameter. (The high-level File Manager routines will work only with files having a version number of Ø.) FileName can contain either the file name alone or the file name prefixed by a volume name.

(note)

> Although fileName can include both the volume name and the file name, applications shouldn't encourage users to prefix a file name with a volume name.

You cannot specify an access path buffer when calling high-level Pascal routines. All access paths open on a volume will share the volume buffer, causing a slight increase in the amount of time required to access files.

All File Manager routines return a result code of type OSErr as their function result. Each routine description lists all of the applicable result codes, along with a short description of what the result code means. Lengthier explanations of all the result codes can be found in the summary at the end of this manual.

Accessing Volumes

```
FUNCTION GetVInfo (drvNum: INTEGER; volName: StringPtr; VAR vRefNum:
            INTEGER; VAR freeBytes: LongInt) : OSErr;
```

GetVInfo returns the name, reference number, and available space (in bytes), in volName, vRefNum, and freeBytes, for the volume in the specified drive.

Result codes		
	noErr	No error
	nsvErr	No default volume
	paramErr	Bad drive number

```
FUNCTION GetVol (volName: StringPtr; VAR vRefNum: INTEGER) : OSErr;
```

GetVol returns the name of the default volume in volName and its volume reference number in vRefNum.

Result codes		
	noErr	No error
	nsvErr	No default volume

```
FUNCTION SetVol (volName: StringPtr; vRefNum: INTEGER) : OSErr;
```

SetVol sets the default volume to the mounted volume specified by volName or vRefNum.

Result codes		
	noErr	No error
	bdNamErr	Bad volume name
	nsvErr	No such volume
	paramErr	No default volume

FUNCTION FlushVol (volName: StringPtr; vRefNum: INTEGER) : OSErr;

On the volume specified by volName or vRefNum, FlushVol writes the contents of the associated volume buffer and descriptive information about the volume (if they've changed since the last time FlushVol was called).

Result codes	noErr	No error
	bdNamErr	Bad volume name
	extFSErr	External file system
	ioErr	Disk I/O error
	nsDrvErr	No such drive
	nsvErr	No such volume
	paramErr	No default volume

FUNCTION UnmountVol (volName: StringPtr; vRefNum: INTEGER) : OSErr;

UnmountVol unmounts the volume specified by volName or vRefNum, by calling FlushVol to flush the volume buffer, closing all open files on the volume, and releasing the memory used for the volume.

(warning)
Don't unmount the startup volume.

Result codes	noErr	No error
	bdNamErr	Bad volume name
	extFSErr	External file system
	ioErr	Disk I/O error
	nsDrvErr	No such drive
	nsvErr	No such volume
	paramErr	No default volume

FUNCTION Eject (volName: StringPtr; vRefNum: INTEGER) : OSErr;

Eject calls FlushVol to flush the volume specified by volName or vRefNum, places the volume offline, and then ejects the volume.

Result codes	noErr	No error
	bdNamErr	Bad volume name
	extFSErr	External file system
	ioErr	Disk I/O error
	nsDrvErr	No such drive
	nsvErr	No such volume
	paramErr	No default volume

Changing File Contents

```
FUNCTION Create (fileName: Str255; vRefNum: INTEGER; creator: OSType;
            fileType: OSType) : OSErr;
```

Create creates a new file with the specified name, file type, and creator, on the specified volume. (File type and creator are discussed in *The Structure of a Macintosh Application*.) The new file is unlocked and empty. Its modification and creation dates are set to the time of the system clock.

Result codes	noErr	No error
	bdNamErr	Bad file name
	dupFNErr	Duplicate file name
	dirFulErr	Directory full
	extFSErr	External file system
	ioErr	Disk I/O error
	nsvErr	No such volume
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock

```
FUNCTION FSOpen (fileName: Str255; vRefNum: INTEGER; VAR refNum:
            INTEGER) : OSErr;
```

FSOpen creates an access path to the file having the name fileName on the specified volume. A path reference number is returned in refNum. The access path's read/write permission is set to whatever the file's open permission allows.

Result codes	noErr	No error
	bdNamErr	Bad file name
	extFSErr	External file system
	fnfErr	File not found
	ioErr	Disk I/O error
	mFulErr	Memory full
	nsvErr	No such volume
	opWrErr	File already open for writing
	tmfoErr	Too many files open

```
FUNCTION FSRead (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr) :
           OSErr;
```

FSRead attempts to read the number of bytes specified by the count parameter from the open file whose access path is specified by refNum, and transfer them to the data buffer pointed to by buffPtr. The read operation begins at the mark, so you might want to precede this with a call to SetFPos. If you try to read past the logical end-of-file, FSRead moves the mark to the end-of-file and returns eofErr as its function result. After the read is completed, the number of bytes actually read is returned in the count parameter.

Result codes	noErr	No error
	eofErr	End-of-file
	extFSErr	External file system
	fnOpnErr	File not open
	ioErr	Disk I/O error
	paramErr	Negative count
	rfNumErr	Bad reference number

```
FUNCTION FSWrite (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr) :
           OSErr;
```

FSWrite takes the number of bytes specified by the count parameter from the buffer pointed to by buffPtr and attempts to write them to the open file whose access path is specified by refNum. The write operation begins at the mark, so you might want to precede this with a call to SetFPos. After the write is completed, the number of bytes actually written is returned in the count parameter.

Result codes	noErr	No error
	dskFulErr	Disk full
	fLckdErr	File locked
	fnOpnErr	File not open
	ioErr	Disk I/O error
	paramErr	Negative count
	rfNumErr	Bad reference number
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock
	wrPermErr	Read/write or open permission doesn't allow writing

```
FUNCTION GetFPos (refNum: INTEGER; VAR filePos: LongInt) : OSErr;
```

GetFPos returns, in filePos, the mark of the open file whose access path is specified by refNum.

Result codes	noErr	No error
	extFSErr	External file system
	fnOpnErr	File not open
	ioErr	Disk I/O error
	rfNumErr	Bad reference number

```
FUNCTION SetFPos (refNum: INTEGER; posMode: INTEGER; posOff: LongInt) :
            OSErr;
```

SetFPos sets the mark of the open file whose access path is specified by refNum, to the position specified by posMode and posOff. PosMode indicates whether the mark should be set relative to the beginning of the file, the logical end-of-file, or the mark; it must contain one of the following predefined constants:

```
CONST fsAtMark    = Ø; {at current position of mark }
                       { (posOff ignored)}
      fsFromStart = 1; {offset relative to beginning of file}
      fsFromLEOF  = 2; {offset relative to logical end-of-file}
      fsFromMark  = 3; {offset relative to current mark}
```

PosOff specifies the byte offset (either positive or negative) relative to posMode where the mark should actually be set. If you try to set the mark past the logical end-of-file, SetFPos moves the mark to the end-of-file and returns eofErr as its function result.

Result codes	noErr	No error
	eofErr	End-of-file
	extFSErr	External file system
	fnOpnErr	File not open
	ioErr	Disk I/O error
	posErr	Tried to position before start of file
	rfNumErr	Bad reference number

```
FUNCTION GetEOF (refNum: INTEGER; VAR logEOF: LongInt) : OSErr;
```

GetEOF returns, in logEOF, the logical end-of-file of the open file whose access path is specified by refNum.

Result codes	noErr	No error
	extFSErr	External file system
	fnOpnErr	File not open
	ioErr	Disk I/O error
	rfNumErr	Bad reference number

FUNCTION SetEOF (refNum: INTEGER; logEOF: LongInt) : OSErr;

SetEOF sets the logical end-of-file of the open file whose access path is specified by refNum, to the position specified by logEOF. If you attempt to set the logical end-of-file beyond the physical end-of-file, the physical end-of-file is set to one byte beyond the end of the next free allocation block; if there isn't enough space on the volume, no change is made, and SetEOF returns dskFulErr as its function result. If logEOF is Ø, all space on the volume occupied by the file is released.

Result codes	noErr	No error
	dskFulErr	Disk full
	extFSErr	External file system
	fLckdErr	File locked
	fnOpnErr	File not open
	ioErr	Disk I/O error
	rfNumErr	Bad reference number
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock
	wrPermErr	Read/write or open permission doesn't allow writing

FUNCTION Allocate (refNum: INTEGER; VAR count: LongInt) : OSErr;

Allocate adds the number of bytes specified by the count parameter to the open file whose access path is specified by refNum, and sets the physical end-of-file to one byte beyond the last block allocated. The number of bytes allocated is always rounded up to the nearest multiple of the allocation block size, and returned in the count parameter. If there isn't enough empty space on the volume to satisfy the allocation request, the rest of the space on the volume is allocated, and Allocate returns dskFulErr as its function result.

Result codes	noErr	No error
	dskFulErr	Disk full
	fLckdErr	File locked
	fnOpnErr	File not open
	ioErr	Disk I/O error
	rfNumErr	Bad reference number
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock
	wrPermErr	Read/write or open permission doesn't allow writing

```
FUNCTION FSClose (refNum: INTEGER) : OSErr;
```

FSClose removes the access path specified by refNum, writes the contents of the volume buffer to the volume, and updates the file's entry in the file directory.

(note)

> Some information stored on the volume won't be correct until FlushVol is called.

<u>Result codes</u>	noErr	No error
	extFSErr	External file system
	fnfErr	File not found
	fnOpnErr	File not open
	ioErr	Disk I/O error
	nsvErr	No such volume
	rfNumErr	Bad reference number

Changing Information About Files

All of the routines described in this section affect both forks of the file, and don't require the file to be open.

```
FUNCTION GetFInfo (fileName: Str255; vRefNum: INTEGER; VAR fndrInfo:
            FInfo) : OSErr;
```

For the file having the name fileName on the specified volume, GetFInfo returns information used by the Finder in fndrInfo (see the section "File Information Used by the Finder").

<u>Result codes</u>	noErr	No error
	bdNamErr	Bad file name
	extFSErr	External file system
	fnfErr	File not found
	ioErr	Disk I/O error
	nsvErr	No such volume
	paramErr	No default volume

```
FUNCTION SetFInfo (fileName: Str255; vRefNum: INTEGER; fndrInfo: FInfo)
            : OSErr;
```

For the file having the name fileName on the specified volume, SetFInfo sets information needed by the Finder to fndrInfo (see the section "File Information Used by the Finder").

<u>Result codes</u>	noErr	No error
	extFSErr	External file system
	fLckdErr	File locked
	fnfErr	File not found
	ioErr	Disk I/O error
	nsvErr	No such volume

	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock

```
FUNCTION SetFLock (fileName: Str255; vRefNum: INTEGER) : OSErr;
```

SetFLock locks the file having the name fileName on the specified volume. Access paths currently in use aren't affected.

<u>Result codes</u>	noErr	No error
	extFSErr	External file system
	fnfErr	File not found
	ioErr	Disk I/O error
	nsvErr	No such volume
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock

```
FUNCTION RstFLock (fileName: Str255; vRefNum: INTEGER) : OSErr;
```

RstFLock unlocks the file having the name fileName on the specified volume. Access paths currently in use aren't affected.

<u>Result codes</u>	noErr	No error
	extFSErr	External file system
	fnfErr	File not found
	ioErr	Disk I/O error
	nsvErr	No such volume
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock

```
FUNCTION Rename (oldName: Str255; vRefNum: INTEGER; newName: Str255) :
            OSErr;
```

Given a file name in oldName, Rename changes the name of the file to newName. Access paths currently in use aren't affected. Given a volume name in oldName or a volume reference number in vRefNum, Rename changes the name of the specified volume to newName.

<u>Result codes</u>	noErr	No error
	bdNamErr	Bad file name
	dirFulErr	Directory full
	dupFNErr	Duplicate file name
	extFSErr	External file system
	fLckdErr	File locked
	fnfErr	File not found
	fsRnErr	Renaming difficulty
	ioErr	Disk I/O error
	nsvErr	No such volume
	paramErr	No default volume
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock

FUNCTION FSDelete (fileName: Str255; vRefNum: INTEGER) : OSErr;

FSDelete removes the closed file having the name fileName from the specified volume.

(note)
This function will delete **both** forks of the file.

Result codes		
	noErr	No error
	bdNamErr	Bad file name
	extFSErr	External file system
	fBsyErr	File busy
	fLckdErr	File locked
	fnfErr	File not found
	ioErr	Disk I/O error
	nsvErr	No such volume
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock

LOW-LEVEL FILE MANAGER ROUTINES

This section contains special information for programmers using the low-level Pascal or assembly-language routines of the File Manager, and describes them in detail. For more information on using assembly language, see Programming Macintosh Applications in Assembly Language.

You can execute most File Manager routines either synchronously (meaning that the application must wait until the routine is completed) or asynchronously (meaning that the application is free to perform other tasks while the routine is executing). MountVol, UnmountVol, Eject, and OffLine cannot be executed asynchronously, because they use the Memory Manager to allocate and deallocate memory.

When an application calls a File Manager routine asynchronously, an I/O request is placed in the file I/O queue, and control returns to the calling application--even before the actual I/O is completed. Requests are taken from the queue one at a time (in the same order that they were entered), and processed. Only one request may be processed at any given time.

The calling application may specify a completion routine to be executed as soon as the I/O operation has been completed.

At any time, you can use the InitQueue procedure to clear all queued File Manager calls except the current one. InitQueue is especially useful when an error occurs and you no longer wish queued calls to be executed.

Routine parameters passed by an application to the File Manager and returned by the File Manager to an application are contained in a parameter block, which is memory space in the heap or stack. Most low-level Pascal calls to the File Manager are of the form

```
PBCallName (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

PBCallName is the name of the routine. ParamBlock points to the parameter block containing the parameters for the routine. If async is TRUE, the call will be executed asynchronously; if FALSE, it will be executed synchronously. Each call returns an integer result code of type OSErr. Each routine description lists all of the applicable result codes, along with a short description of what the result code means. Lengthier explanations of all the result codes can be found in the summary at the end of this manual.

Assembly-language note: When you call a File Manager routine, AØ must point to a parameter block containing the parameters for the routine. If you want the routine to be executed asynchronously, set bit 1Ø of the routine trap word. You can do this by supplying the word ASYNC as the second argument to the routine macro. For example:

```
_Read  paramBlock,ASYNC
```

You can set or test bit 1Ø of a trap word by using the global constant asynTrpBit.

If you want a routine to be executed immediately (bypassing the file I/O queue), set bit 9 of the routine trap word. This can be accomplished by supplying the word IMMED as the second argument to the routine macro. For example:

```
_Write  paramBlock,IMMED
```

You can set or test bit 9 of a trap word by using the global constant noQueueBit. You can specify either ASYNC or IMMED, but not both.

All routines except InitQueue return a result code in DØ.

Routine Parameters

There are three different kinds of parameter blocks you'll pass to File Manager routines. Each kind is used with a particular set of routine calls: I/O routines, file information routines, and volume information routines.

The lengthy, variable-length data structure of a parameter block is given below. The Device Manager and File Manager use this same data structure, but only the parts relevant to the File Manager are shown here. Each kind of parameter block contains eight fields of standard information and nine to 16 fields of additional information:

```
TYPE ParamBlkType = (ioParam, fileParam, volumeParam, cntrlParam);

     ParamBlockRec = RECORD
                       qLink:        QElemPtr;  {next queue entry}
                       qType:        INTEGER;   {queue type}
                       ioTrap:       INTEGER;   {routine trap}
                       ioCmdAddr:    Ptr;       {routine address}
                       ioCompletion: ProcPtr;   {completion routine}
                       ioResult:     OSErr;     {result code}
                       ioNamePtr:    StringPtr; {volume or file name}
                       ioVRefNum:    INTEGER;   {volume reference or }
                                                { drive number}
                       CASE ParamBlkType OF
                        ioParam:
                         . . . {I/O routine parameters}
                        fileParam:
                         . . . {file information routine parameters}
                        volumeParam:
                         . . . {volume information routine parameters}
                        cntrlParam:
                         . . . {Control and Status call parameters}
                     END;

     ParmBlkPtr = ^ParamBlockRec;
```

The first four fields in each parameter block are handled entirely by the File Manager, and most programmers needn't be concerned with them; programmers who are interested in them should see the section "Data Structures in Memory".

IOCompletion contains the address of a completion routine to be executed at the end of an asynchronous call; it should be NIL for asynchronous calls with no completion routine, and is automatically set to NIL for all synchronous calls. For asynchronous calls, ioResult is positive while the routine is executing, and returns the result code. Your application can poll ioResult during the asynchronous execution of a routine to determine when the routine has completed. Completion routines are executed after ioResult is returned.

IONamePtr points to either a volume name or a file name (which can be prefixed by a volume name).

(note)

> Although ioNamePtr can include both the volume name and the file name, applications shouldn't encourage users to prefix a file name with a volume name.

IOVRefNum contains either the reference number of a volume or the drive number of a drive containing a volume.

For routines that access volumes, the File Manager determines which volume to access by using one of the following:

1. IONamePtr, a pointer to the volume name.

2. If ioNamePtr is NIL, or points to an improper volume name, then ioVRefNum. (If ioVRefNum is negative, it's a volume reference number; if positive, it's a drive number.)

3. If ioVRefNum is Ø, the default volume. (If there isn't a default volume, an error is returned.)

For routines that access closed files, the File Manager determines which file to access by using ioNamePtr, a pointer to the name of the file (and possibly also of the volume).

- If the string pointed to by ioNamePtr doesn't include the volume name, the File Manager uses steps 2 and 3 above to determine the volume.

- If ioNamePtr is NIL or points to an improper file name, an error is returned.

The first eight fields are adequate for a few calls, but most of the File Manager routines require more fields, as described below. The parameters used with Control and Status calls are described in the Device Manager manual *** doesn't yet exist ***.

I/O Parameters

When you call one of the I/O routines, you'll use these nine additional fields after the standard 8-field parameter block:

```
ioParam:
 (ioRefNum:    INTEGER;    {path reference number}
  ioVersNum:   SignedByte; {version number}
  ioPermssn:   SignedByte; {read/write permission}
  ioMisc:      Ptr;        {miscellaneous}
  ioBuffer:    Ptr;        {data buffer}
  ioReqCount:  LongInt;    {requested number of bytes}
  ioActCount:  LongInt;    {actual number of bytes}
  ioPosMode:   INTEGER;    {newline character and type of }
                           { positioning operation}
  ioPosOffset: LongInt);   {size of positioning offset}
```

For routines that access open files, the File Manager determines which file to access by using the path reference number in ioRefNum. IOPermssn requests permission to read or write via an access path, and must contain one of the following predefined constants:

```
CONST fsCurPerm  = Ø; {whatever is currently allowed}
      fsRdPerm   = 1; {request to read only
      fsWrPerm   = 2; {request to write only}
      fsRdWrPerm = 3; {request to read and write}
```

This request is compared with the open permission of the file. If the open permission doesn't allow I/O as requested, an error will be returned.

The content of ioMisc depends on the routine called; it contains either a pointer to an access path buffer, a new logical end-of-file, a new version number, or a pointer to a new volume or file name. Since ioMisc is of type Ptr, while end-of-file is LongInt and version number is SignedByte, you'll need to perform type conversions to correctly interpret the value of ioMisc.

IOBuffer points to a data buffer into which data is written by Read calls and from which data is read by Write calls. IOReqCount specifies the requested number of bytes to be read, written, or allocated. IOActCount contains the number of bytes actually read, written, or allocated.

IOPosMode and ioPosOffset contain positioning information used for Read, Write, and SetFPos calls. Bits Ø and 1 of ioPosMode indicate how to position the mark, and you can use the following predefined constants to set or test their value:

```
CONST fsAtMark    = Ø; {at current position of mark }
                       { (ioPosOffset ignored)}
      fsFromStart = 1; {offset relative to beginning of file}
      fsFromLEOF  = 2; {offset relative to logical end-of-file}
      fsFromMark  = 3; {offset relative to current mark}
```

IOPosOffset specifies the byte offset (either positive or negative) relative to ioPosMode where the operation will be performed.

> Assembly-language note: If bit 6 of ioPosMode is set, the File Manager will verify that all data read into memory by a Read call exactly matches the data on the volume (ioErr will be returned if any of the data doesn't match).

(note)

> Advanced programmers: Bit 7 of ioPosMode is the newline flag--set if read operations should terminate at newline characters, and clear if reading should terminate at the end of the access path buffer or volume buffer. The high-order byte of ioPosMode contains the ASCII code of the newline character.

File Information Parameters

When you call the PBGetFileInfo and PBSetFileInfo functions, you'll use the following 16 additional fields after the standard 8-field parameter block:

```
fileParam:
 (ioFRefNum:     INTEGER;    {path reference number}
  ioFVersNum:    SignedByte; {version number}
  filler1:       SignedByte; {not used}
  ioFDirIndex:   INTEGER;    {file number}
  ioFlAttrib:    SignedByte; {file attributes}
  ioFlVersNum:   SignedByte; {version number}
  ioFlFndrInfo:  FInfo;      {information used by the Finder}
  ioFlNum:       LongInt;    {file number}
  ioFlStBlk:     INTEGER;    {first allocation block of data fork}
  ioFlLgLen:     LongInt;    {logical end-of-file of data fork}
  ioFlPyLen:     LongInt;    {physical end-of-file of data fork}
  ioFlRStBlk:    INTEGER;    {first allocation block of resource fork}
  ioFlRLgLen:    LongInt;    {logical end-of-file of resource fork}
  ioFlRPyLen:    LongInt;    {physical end-of-file of resource fork}
  ioFlCrDat:     LongInt;    {date and time of creation}
  ioFlMdDat:     LongInt);   {date and time of last modification}
```

IOFDirIndex contains the file number, another method of referring to a file; most programmers needn't be concerned with file numbers, but those interested can read the section "Data Organization on Volumes".

Assembly-language note: IOFlAttrib contains eight bits of file attributes: if bit 7 is set, the file is open; if bit Ø is set, the file is locked.

IOFlStBlk and ioFlRStBlk contain Ø if the file's data or resource fork is empty, respectively. The date and time in the ioFlCrDat and ioFlMdDat fields are specified in seconds since 12:ØØ AM, January 1, 19Ø4.

Volume Information Parameters

When you call GetVolInfo, you'll use the following 14 additional fields:

```
volumeParam:
 (filler2:     LongInt;  {not used}
  ioVolIndex:  INTEGER;  {volume index}
  ioVCrDate:   LongInt;  {date and time of initialization}
  ioVLsBkUp:   LongInt;  {date and time of last volume backup}
  ioVAtrb:     INTEGER;  {bit 15=1 if volume locked}
  ioVNmFls:    INTEGER;  {number of files in file directory}
  ioVDirSt:    INTEGER;  {first block of file directory}
  ioVBlLn:     INTEGER;  {number of blocks in file directory}
  ioVNmAlBlks: INTEGER;  {number of allocation blocks on volume}
  ioVAlBlkSiz: LongInt;  {number of bytes per allocation block}
  ioVClpSiz:   LongInt;  {number of bytes to allocate}
  ioAlBlSt:    INTEGER;  {first block in volume block map}
  ioVNxtFNum:  LongInt;  {next free file number}
  ioVFrBlk:    INTEGER); {number of free allocation blocks}
```

IOVolIndex contains the volume index, another method of referring to a volume; the first volume mounted has an index of 1, and so on. Most programmers needn't be concerned with the parameters providing information about file directories and block maps (such as ioVNmFls), but interested programmers can read the section "Data Organization on Volumes".

Routine Descriptions

This section describes the procedures and functions. Each routine description includes the low-level Pascal form of the call and the routine's assembly-language macro. A list of the fields in the parameter block affected by the call is also given.

Assembly-language note: The field names given in these descriptions are those of the ParamBlockRec data type; see the "Summary of the File Manager" for the equivalent assembly-language equates.

The number next to each parameter name indicates the byte offset of the parameter from the start of the parameter block pointed to by AØ; only assembly-language programmers need be concerned with it. An arrow drawn next to each parameter name indicates whether it's an input, output, or input/output parameter:

Arrow	Meaning
-->	Parameter must be passed to the routine
<--	Parameter will be returned by the routine
<-->	Parameter must be passed to and will be returned by the routine

Initializing the File I/O Queue

PROCEDURE InitQueue;

Trap macro _InitQueue

InitQueue clears all queued File Manager calls except the current one. There are no parameters or result codes associated with InitQueue.

Accessing Volumes

FUNCTION PBMountVol (paramBlock: ParmBlkPtr) : OSErr;

Trap macro _MountVol

Parameter block

<--	16	ioResult	word
<->	22	ioVRefNum	word

Result codes

noErr	No error
badMDBErr	Master directory block is bad
extFSErr	External file system
ioErr	Disk I/O error
mFulErr	Memory full
noMacDskErr	Not a Macintosh volume
nsDrvErr	No such drive
paramErr	Bad drive number
volOnLinErr	Volume already on-line

PBMountVol mounts the volume in the drive whose number is ioVRefNum, and returns a volume reference number in ioVRefNum. If there are no volumes already mounted, this volume becomes the default volume. PBMountVol is always executed synchronously.

FUNCTION PBGetVolInfo (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _GetVolInfo

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
<->	18	ioNamePtr	pointer
<->	22	ioVRefNum	word
-->	28	ioVolIndex	word
<--	3Ø	ioVCrDate	long word
<--	34	ioVLsBkUp	long word
<--	38	ioVAtrb	word
<--	4Ø	ioVNmFls	word
<--	42	ioVDirSt	word
<--	44	ioVBlLn	word
<--	46	ioVNmAlBlks	word
<--	48	ioVAlBlkSiz	long word
<--	52	ioVClpSiz	long word
<--	56	ioAlBlSt	word
<--	58	ioVNxtFNum	long word
<--	62	ioVFrBlk	word

Result codes

noErr	No error
nsvErr	No such volume
paramErr	No default volume

PBGetVolInfo returns information about the specified volume. If ioVolIndex is positive, the File Manager attempts to use it to find the volume. If ioVolIndex is negative, the File Manager uses ioNamePtr and ioVRefNum in the standard way to determine which volume. If ioVolIndex is Ø, the File Manager attempts to access the volume by using ioVRefNum only. The volume reference number is returned in ioVRefNum, and the volume name is returned in ioNamePtr (unless ioNamePtr is NIL).

FUNCTION PBGetVol (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _GetVol

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
<--	18	ioNamePtr	pointer
<--	22	ioVRefNum	word

Result codes	noErr	No error
	nsvErr	No default volume

PBGetVol returns the name of the default volume in ioNamePtr and its volume reference number in ioVRefNum (unless ioNamePtr is NIL).

FUNCTION PBSetVol (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _SetVol

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word

Result codes	noErr	No error
	bdNamErr	Bad volume name
	nsvErr	No such volume
	paramErr	No default volume

PBSetVol sets the default volume to the mounted volume specified by ioNamePtr or ioVRefNum.

FUNCTION PBFlshVol (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _FlushVol

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word

Result codes

noErr	No error
bdNamErr	Bad volume name
extFSErr	External file system
ioErr	Disk I/O error
nsDrvErr	No such drive
nsvErr	No such volume
paramErr	No default volume

PBFlshVol writes descriptive information, the contents of the associated volume buffer, and all access path buffers to the volume specified by ioNamePtr or ioVRefNum, to the volume (if they've changed since the last time PBFlshVol was called). The volume modification date is set to the current time.

FUNCTION PBUnmountVol (paramBlock: ParmBlkPtr) : OSErr;

Trap macro _UnmountVol

Parameter block

<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word

Result codes

noErr	No error
bdNamErr	Bad volume name
extFSErr	External file system
ioErr	Disk I/O error
nsDrvErr	No such drive
nsvErr	No such volume
paramErr	No default volume

PBUnmountVol unmounts the volume specified by ioNamePtr or ioVRefNum, by calling PBFlshVol to flush the volume, closing all open files on the volume, and releasing all the memory used for the volume. PBUnmountVol is always executed synchronously.

(warning)
Don't unmount the startup volume.

FUNCTION PBOffLine (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _OffLine

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word

Result codes

noErr	No error
bdNamErr	Bad volume name
extFSErr	External file system
ioErr	Disk I/O error
nsDrvErr	No such drive
nsvErr	No such volume
paramErr	No default volume

PBOffLine places off-line the volume specified by ioNamePtr or ioVRefNum, by calling PBFlshVol to flush the volume, and releasing all the memory used for the volume except for 94 bytes of descriptive information.

FUNCTION PBEject (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

<u>Trap macro</u> _Eject

<u>Parameter block</u>

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word

<u>Result codes</u>

noErr	No error
bdNamErr	Bad volume name
extFSErr	External file system
ioErr	Disk I/O error
nsDrvErr	No such drive
nsvErr	No such volume
paramErr	No default volume

PBEject calls PBOffLine to place the volume specified by ioNamePtr or ioVRefNum off-line, and then ejects the volume.

You may call PBEject asynchronously; the first part of the call is executed synchronously, and the actual ejection is executed asynchronously.

Changing File Contents

FUNCTION PBCreate (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Create

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte

Result codes

noErr	No error
bdNamErr	Bad file name
dupFNErr	Duplicate file name
dirFulErr	Directory full
extFSErr	External file system
ioErr	Disk I/O error
nsvErr	No such volume
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

PBCreate creates a new file having the name ioNamePtr and the version number ioVersNum, on the specified volume. The new file is unlocked and empty. Its modification and creation dates are set to the time of the system clock. The application should call PBSetFInfo to fill in the information needed by the Finder.

FUNCTION PBOpen (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Open

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
<--	24	ioRefNum	word
-->	26	ioVersNum	byte
-->	27	ioPermssn	byte
-->	28	ioMisc	pointer

Result codes

noErr	No error
bdNamErr	Bad file name
extFSErr	External file system
fnfErr	File not found
ioErr	Disk I/O error
mFulErr	Memory full
nsvErr	No such volume
opWrErr	File already open for writing
tmfoErr	Too many files open

PBOpen creates an access path to the file having the name ioNamePtr and the version number ioVersNum, on the specified volume. A path reference number is returned in ioRefNum.

IOMisc either points to a 522-byte portion of memory to be used as the access path's buffer, or is NIL if you want the volume buffer to be used instead.

(warning)

> All access paths to a single file that's opened multiple times should share the same buffer so that they will read and write the same data.

IOPermssn specifies the path's read/write permission. A path can be opened for writing even if it accesses a file on a locked volume, and an error won't be returned until a PBWrite, PBSetEOF, or PBAllocate call is made.

If you attempt to open a locked file for writing, PBOpen will return opWrErr as its function result. If you attempt to open a file for writing and it already has an access path that allows writing, PBOpen will return the reference number of the existing access path in ioRefNum and opWrErr as its function result.

FUNCTION PBOpenRF (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

<u>Trap macro</u> _OpenRF

<u>Parameter block</u>

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
<--	24	ioRefNum	word
-->	26	ioVersNum	byte
-->	27	ioPermssn	byte
-->	28	ioMisc	pointer

<u>Result codes</u>

noErr	No error
bdNamErr	Bad file name
extFSErr	External file system
fnfErr	File not found
ioErr	Disk I/O error
mFulErr	Memory full
nsvErr	No such volume
opWrErr	File already open for writing
permErr	Open permission doesn't allow reading
tmfoErr	Too many files open

PBOpenRF is identical to PBOpen, except that it opens the file's resource fork instead of its data fork.

```
FUNCTION PBRead (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _Read

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word
-->	32	ioBuffer	pointer
-->	36	ioReqCount	long word
<--	4Ø	ioActCount	long word
-->	44	ioPosMode	word
<->	46	ioPosOffset	long word

Result codes

noErr	No error
eofErr	End-of-file
extFSErr	External file system
fnOpnErr	File not open
ioErr	Disk I/O error
paramErr	Negative ioReqCount
rfNumErr	Bad reference number

PBRead attempts to read ioReqCount bytes from the open file whose access path is specified by ioRefNum, and transfer them to the data buffer pointed to by ioBuffer. If you try to read past the logical end-of-file, PBRead moves the mark to the end-of-file and returns eofErr as its function result. After the read operation is completed, the mark is returned in ioPosOffset and the number of bytes actually read is returned in ioActCount.

(note)

> Advanced programmers: IOPosMode contains the newline character (if any), and indicates whether the read should begin relative to the beginning of the file, the mark, or the end-of-file. The byte offset from the position indicated by ioPosMode, where the read should actually begin, is given by ioPosOffset. If a newline character is not specified, the data will be read one byte at a time until ioReqCount bytes have been read or the end-of-file is reached. If a newline character is specified, the data will be read one byte at a time until the newline character is encountered, the end-of-file is reached, or ioReqCount bytes have been read.

FUNCTION PBWrite (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Write

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word
-->	32	ioBuffer	pointer
-->	36	ioReqCount	long word
<--	40	ioActCount	long word
-->	44	ioPosMode	word
-->	46	ioPosOffset	long word

Result codes

noErr	No error
dskFulErr	Disk full
fLckdErr	File locked
fnOpnErr	File not open
ioErr	Disk I/O error
paramErr	Negative ioReqCount
posErr	Position is beyond end-of-file
rfNumErr	Bad reference number
vLckdErr	Software volume lock
wPrErr	Hardware volume lock
wrPermErr	Read/write or open permission doesn't allow writing

PBWrite takes ioReqCount bytes from the buffer pointed to by ioBuffer and attempts to write them to the open file whose access path is specified by ioRefNum. After the write operation is completed, the mark is returned in ioPosOffset, and the number of bytes actually written is returned in ioActCount.

IOPosMode indicates whether the write should begin relative to the beginning of the file, the mark, or the end-of-file. The byte offset from the position indicated by ioPosMode, where the write should actually begin, is given by ioPosOffset.

FUNCTION PBGetFPos (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _GetFPos

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	22	ioRefNum	word
<--	36	ioReqCount	long word
<--	4Ø	ioActCount	long word
<--	44	ioPosMode	word
<--	46	ioPosOffset	long word

Result codes

noErr	No error
extFSErr	External file system
fnOpnErr	File not open
ioErr	Disk I/O error
rfNumErr	Bad reference number

PBGetFPos returns, in ioPosOffset, the mark of the open file whose access path is specified by ioRefNum. PBGetFPos sets ioReqCount, ioActCount, and ioPosMode to Ø.

FUNCTION PBSetFPos (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _SetFPos

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	22	ioRefNum	word
-->	44	ioPosMode	word
-->	46	ioPosOffset	long word

Result codes

noErr	No error
eofErr	End-of-file
extFSErr	External file system
fnOpnErr	File not open
ioErr	Disk I/O error
posErr	Tried to position before start of file
rfNumErr	Bad reference number

PBSetFPos sets the mark of the open file whose access path is specified by ioRefNum, to the position specified by ioPosMode and ioPosOffset. IoPosMode indicates whether the mark should be set relative to the beginning of the file, the mark, or the logical end-of-file. The byte offset from the position given by ioPosMode, where the mark should actually be set, is given by ioPosOffset. If you try to set the mark past the logical end-of-file, PBSetFPos moves the mark to the end-of-file and returns eofErr as its function result.

FUNCTION PBGetEOF (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _GetEOF

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	22	ioRefNum	word
<--	28	ioMisc	long word

Result codes

noErr	No error
extFSErr	External file system
fnOpnErr	File not open
ioErr	Disk I/O error
rfNumErr	Bad reference number

PBGetEOF returns, in ioMisc, the logical end-of-file of the open file whose access path is specified by ioRefNum.

FUNCTION PBSetEOF (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _SetEOF

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	22	ioRefNum	word
-->	28	ioMisc	long word

Result codes

noErr	No error
dskFulErr	Disk full
extFSErr	External file system
fLckdErr	File locked
fnOpnErr	File not open
ioErr	Disk I/O error
rfNumErr	Bad reference number
vLckdErr	Software volume lock
wPrErr	Hardware volume lock
wrPermErr	Read/write or open permission doesn't allow writing

PBSetEOF sets the logical end-of-file of the open file whose access path is specified by ioRefNum, to ioMisc. If the logical end-of-file is set beyond the physical end-of-file, the physical end-of-file is set to one byte beyond the end of the next free allocation block; if there isn't enough space on the volume, no change is made, and PBSetEOF returns dskFulErr as its function result. If ioMisc is Ø, all space on the volume occupied by the file is released.

FUNCTION PBAllocate (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Allocate

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	22	ioRefNum	word
-->	36	ioReqCount	long word
<--	40	ioActCount	long word

Result codes

noErr	No error
dskFulErr	Disk full
fLckdErr	File locked
fnOpnErr	File not open
ioErr	Disk I/O error
rfNumErr	Bad reference number
vLckdErr	Software volume lock
wPrErr	Hardware volume lock
wrPermErr	Read/write or open permission doesn't allow writing

PBAllocate adds ioReqCount bytes to the open file whose access path is specified by ioRefNum, and sets the physical end-of-file to one byte beyond the last block allocated. The number of bytes allocated is always rounded up to the nearest multiple of the allocation block size, and returned in ioActCount. If there isn't enough empty space on the volume to satisfy the allocation request, PBAllocate allocates the rest of the space on the volume and returns dskFulErr as its function result.

FUNCTION PBFlshFile (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _FlushFile

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	22	ioRefNum	word

Result codes

noErr	No error
extFSErr	External file system
fnfErr	File not found
fnOpnErr	File not open
ioErr	Disk I/O error
nsvErr	No such volume
rfNumErr	Bad reference number

PBFlshFile writes the contents of the access path buffer indicated by ioRefNum to the volume, and updates the file's entry in the file directory.

(warning)

> Some information stored on the volume won't be correct until PBFlshVol is called.

FUNCTION PBClose (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Close

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word

Result codes

noErr	No error
extFSErr	External file system
fnfErr	File not found
fnOpnErr	File not open
ioErr	Disk I/O error
nsvErr	No such volume
rfNumErr	Bad reference number

PBClose writes the contents of the access path buffer specified by ioRefNum to the volume and removes the access path.

(warning)

> Some information stored on the volume won't be correct until PBFlshVol is called.

Changing Information About Files

All of the routines described in this section affect both forks of a file.

FUNCTION PBGetFInfo (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _GetFileInfo

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
<--	24	ioRefNum	word
-->	26	ioVersNum	byte
-->	28	ioFDirIndex	word
<--	3Ø	ioFlAttrib	byte
<--	31	ioFlVersNum	byte
<--	32	ioFndrInfo	16 bytes
<--	48	ioFlNum	long word
<--	52	ioFlStBlk	word
<--	54	ioFlLgLen	long word
<--	58	ioFlPyLen	long word
<--	62	ioFlRStBlk	word
<--	64	ioFlRLgLen	long word
<--	68	ioFlRPyLen	long word
<--	72	ioFlCrDat	long word
<--	76	ioFlMdDat	long word

Result codes

noErr	No error
bdNamErr	Bad file name
extFSErr	External file system
fnfErr	File not found
ioErr	Disk I/O error
nsvErr	No such volume
paramErr	No default volume

PBGetFInfo returns information about the specified file. If ioFDirIndex is positive, the File Manager returns information about the file whose file number is ioFDirIndex on the specified volume (see the section "Data Organization on Volumes" if you're interested in using this method). If ioFDirIndex is negative or Ø, the File Manager returns information about the file having the name ioNamePtr and the version number ioVersNum, on the specified volume. Unless ioNamePtr is NIL, ioNamePtr returns a pointer to the name of the file. If the file is open, the reference number of the first access path found is returned in ioRefNum.

FUNCTION PBSetFInfo (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

<u>Trap macro</u> _SetFileInfo

<u>Parameter block</u>

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte
-->	32	ioFndrInfo	16 bytes
-->	72	ioFlCrDat	long word
-->	76	ioFlMdDat	long word

<u>Result codes</u>

noErr	No error
bdNamErr	Bad file name
extFSErr	External file system
fLckdErr	File locked
fnfErr	File not found
ioErr	Disk I/O error
nsvErr	No such volume
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

PBSetFInfo sets information (including creation and modification dates, and information needed by the Finder) about the file having the name ioNamePtr and the version number ioVersNum on the specified volume. You should call PBGetFInfo just before PBSetFInfo, so the current information is present in the parameter block.

```
FUNCTION PBSetFLock (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _SetFilLock

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte

Result codes

noErr	No error
extFSErr	External file system
fnfErr	File not found
ioErr	Disk I/O error
nsvErr	No such volume
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

PBSetFLock locks the file having the name ioNamePtr and the version number ioVersNum on the specified volume. Access paths currently in use aren't affected.

```
FUNCTION PBRstFLock (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _RstFilLock

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte

Result codes

noErr	No error
extFSErr	External file system
fnfErr	File not found
ioErr	Disk I/O error
nsvErr	No such volume
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

PBRstFLock unlocks the file having the name ioNamePtr and the version number ioVersNum on the specified volume. Access paths currently in use aren't affected.

FUNCTION PBSetFVers (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _SetFilType

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte
-->	28	ioMisc	byte

Result codes

noErr	No error
bdNamErr	Bad file name
dupFNErr	Duplicate file name and version
extFSErr	External file system
fLckdErr	File locked
fnfErr	File not found
nsvErr	No such volume
ioErr	Disk I/O error
paramErr	No default volume
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

PBSetFVers changes the version number of the file having the name ioNamePtr and version number ioVersNum on the specified volume, to ioMisc. Access paths currently in use aren't affected.

(warning)

> The Resource Manager and Segment Loader operate only on files with version number Ø; changing the version number of a file to a nonzero number will prevent them from operating on it.

FUNCTION PBRename (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Rename

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte
-->	28	ioMisc	pointer

Result codes

noErr	No error
bdNamErr	Bad file name
dirFulErr	Directory full
dupFNErr	Duplicate file name and version
extFSErr	External file system
fLckdErr	File locked
fnfErr	File not found
fsRnErr	Renaming difficulty
ioErr	Disk I/O error
nsvErr	No such volume
paramErr	No default volume
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

Given a file name in ioNamePtr and a version number in ioVersNum, Rename changes the name of the specified file to ioMisc; given a volume name in ioNamePtr or a volume reference number in ioVRefNum, it changes the name of the specified volume to ioMisc. Access paths currently in use aren't affected.

```
FUNCTION PBDelete (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _Delete

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte

Result codes

noErr	No error
bdNamErr	Bad file name
extFSErr	External file system
fBsyErr	File busy
fLckdErr	File locked
fnfErr	File not found
nsvErr	No such volume
ioErr	Disk I/O error
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

PBDelete removes the closed file having the name ioNamePtr and the version number ioVersNum, from the specified volume.

(note)
This function will delete **both** forks of the file.

DATA ORGANIZATION ON VOLUMES

This section explains how information is organized on volumes. Most of the information is accessible only through assembly language, but some advanced Pascal programmers may be interested.

The File Manager communicates with device drivers that read and write data via block-level requests to devices containing Macintosh-initialized volumes. (Macintosh-initialized volumes are volumes initialized by the Disk Initialization Package.) The actual type of volume and device is unimportant to the File Manager; the only requirements are that the volume was initialized by the Disk Initialization Package and that the device driver is able to communicate via block-level requests.

The 3 1/2-inch built-in and optional external drives are accessed via the Disk Driver. If you want to use the File Manager to access files on Macintosh-initialized volumes on other types of devices, you must write a device driver that can read and write data via block-level requests to the device on which the volume will be mounted. If you want to access files on nonMacintosh-initialized volumes, you must write your own external file system (see the section "Using an External File System").

The information on all block-formatted volumes is organized in logical blocks and allocation blocks. Logical blocks contain a number of bytes of standard information (512 bytes on Macintosh-initialized volumes), and an additional number of bytes of information specific to the disk driver (12 bytes on Macintosh-initialized volumes). Allocation blocks are composed of any integral number of logical blocks, and are simply a means of grouping logical blocks together in more convenient parcels.

The remainder of this section applies only to Macintosh-initialized volumes. NonMacintosh-initialized volumes must be accessed via an external file system, and the information on them must be organized by an external initializing program.

A Macintosh-initialized volume contains information needed to start up the system in logical blocks Ø and 1 (Figure 6). Logical block 2 of the volume begins the master directory block. The master directory block contains volume information and the volume allocation block map, which records whether each block on the volume is unused or what part of a file it contains data from.

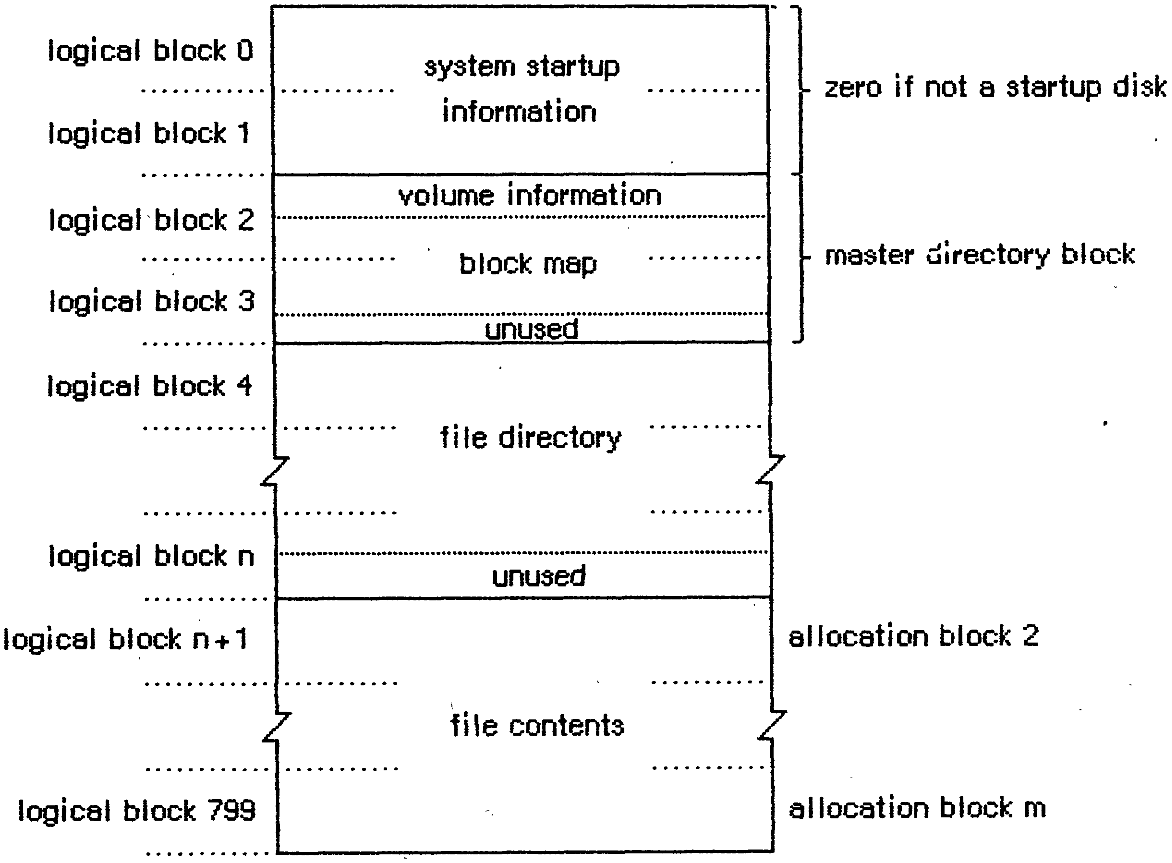


Figure 6. A 4ØØK-Byte Volume With 1K-Byte Allocation Blocks

The master directory "block" always occupies two blocks--the Disk Initialization Package varies the allocation block size as necessary to achieve this constraint.

In the next logical block following the block map begins the file directory, which contains descriptions and locations of all the files on the volume. The rest of the logical blocks on the volume contain files or garbage (such as parts of deleted files). The exact format of the volume information, volume allocation block map, file directory, and files is explained in the following sections.

Volume Information

The volume information is contained in the first 64 bytes of the master directory block (Figure 7). This information is written on the volume when it's initialized, and modified thereafter by the File Manager.

byte 0	drSigWord (word)	always $D2D7
2	drCrDate (long word)	date and time of intialization
6	drLsBkUp (long word)	date and time of last backup
10	drAtrb (word)	volume attributes
12	drNmFls (word)	number of files in file directory
14	drDirSt (word)	first logical block of file directory
16	drBlLen (word)	number of logical blocks in file directory
18	drNmAlBlks (word)	number of allocation blocks on volume
20	drAlBlkSiz (long word)	size of allocation blocks
24	drClpSiz (long word)	number of bytes to allocate
28	drAlBlSt (word)	logical block number of first allocation block
30	drNxtFNum (long word)	next unused file number
34	drFreeBks (word)	number of unused allocation blocks
36	drVN (byte)	length of volume name
37	drVN+1 (bytes)	characters of volume name

Figure 7. Volume Information

DrAtrb contains the volume attributes. Its bits, if set, indicate the following:

Bit	Meaning
7	Volume is locked by hardware
15	Volume is locked by software

DrClpSiz contains the minimum number of bytes to allocate each time the Allocate function is called, to minimize fragmentation of files; it's always a multiple of the allocation block size. DrNxtFNum contains the next unused file number (see the "File Directory" section below for an explanation of file numbers).

Volume Allocation Block Map

The volume allocation block map represents every allocation block on the volume with a 12-bit entry indicating whether the block is unused or allocated to a file. It begins in the master directory block at the byte following the volume information, and continues for as many logical blocks as needed. For example, a 4ØØK-byte volume with a 1Ø-block file directory and 1K-byte allocation blocks would have a 591-byte block map.

The first entry in the block map is for block number 2; the block map doesn't contain entries for the startup blocks. Each entry specifies whether the block is unused, whether it's the last block in the file, or which allocation block is next in the file:

Entry	Meaning
Ø	Block is unused
1	Block is the last block of the file
2..4Ø95	Number of next block in the file

For instance, assume that there's one file on the volume, stored in allocation blocks 8, 11, 12, and 17; the first 16 entries of the block map would read

Ø Ø Ø Ø Ø Ø 11 Ø Ø 12 17 Ø Ø Ø Ø 1

The first allocation block on a volume typically follows the file directory. The first allocation block is number 2 because of the special meaning of numbers Ø and 1.

(note)

> As explained below, it's possible to begin the allocation blocks immediately following the master directory block and place the file directory somewhere within the allocation blocks. In this case, the allocation blocks occupied by the file directory must be marked with $FFF's in the allocation block map.

File Directory

The file directory contains an entry for each file. Each entry lists information about one file on the volume, including its name and location. Each file is listed by its own unique file number, which the File Manager uses to distinguish it from other files on the volume.

A file directory entry contains 51 bytes plus one byte for each character in the file name (Figure 8); if the file names average 2Ø characters, a directory can hold seven file entries per logical block. Entries are always an integral number of words and don't cross logical block boundaries. The length of a file directory depends on the maximum number of files the volume can contain; for example, on a 4ØØK-byte volume the file directory occupies 12 logical blocks.

The file directory conventionally follows the block map and precedes the allocation blocks, but a volume-initializing program could actually place the file directory anywhere within the allocation blocks as long as the blocks occupied by the file directory are marked with $FFF's in the block map.

byte 0	flFlags (byte)	bit 7=1 if entry used; bit 0=1 if file locked
1	flTyp (byte)	version number
2	flUsrWds (16 bytes)	information used by the Finder
18	flFlNum (long word)	file number
22	flStBlk (word)	first allocation block of data fork
24	flLgLen (long word)	data fork's logical end-of-file
28	flPyLen (long word)	data fork's physical end-of-file
32	flRStBlk (word)	first allocation block of resource fork
34	flRLgLen (long word)	resource fork's logical end-of-file
38	flRPyLen (long word)	resource fork's physical end-of-file
42	flCrDat (long word)	date and time file was created
46	flMdDat (long word)	date and time file was last modified
50	flName (byte)	length of file name
51	flNam+1 (bytes)	characters of file name

Figure 8. A File Directory Entry

FlStBlk and flRStBlk are Ø if the data or resource fork doesn't exist. FlCrDat and flMdDat are given in seconds since 12:ØØ AM, January 1, 19Ø4.

Each time a new file is created, an entry for the new file is placed in the file directory. Each time a file is deleted, its entry in the file directory is cleared, and all blocks used by that file on the volume are released.

File Tags on Volumes

As mentioned previously, logical blocks contain 512 bytes of standard information preceded by 12 bytes of file tags (Figure 9). The file tags are designed to allow easy reconstruction of files from a volume whose directory or other file-access information has been destroyed.

byte 0	file number (long word)	file number
4	fork type (byte)	bit 1 = 1 if resource fork
5	file attributes (byte)	bit 7 = 1 if open; bit 0 = 1 if locked
6	file sequence (word)	logical block sequence number
8	mod date (long word)	date and time last modified

Figure 9. File Tags on Volumes

The file sequence indicates which relative portion of a file the block contains--the first logical block of a file has a sequence number of Ø, the second a sequence number of 1, and so on.

DATA STRUCTURES IN MEMORY

This section describes the memory data structures used by the File Manager and any external file system that accesses files on Macintosh-initialized volumes. Most of this data is accessible only through assembly language, but some advanced Pascal programmers may be interested.

The data structures in memory used by the File Manager and all external file systems include:

- the file I/O queue, listing the currently executing routine (if any), and any asynchronous routines awaiting execution
- the volume-control-block queue, listing information about each mounted volume
- copies of volume allocation block maps; one for each on-line volume
- the file-control-block buffer, listing information about each access path
- volume buffers; one for each on-line volume
- optional access path buffers; one for each access path
- the drive queue, listing information about each drive connected to the Macintosh

The File I/O Queue

The file I/O queue is a standard Operating System queue (described in the appendix) that contains a list of all asynchronous routines awaiting execution. Each time a routine is called, an entry is placed in the queue; each time a routine is completed, its entry is removed from the queue.

The file I/O queue uses entries of type ioQType, each of which consists of a parameter block for the routine that was called. The structure of this block is shown in part below:

```
TYPE ParamBlockRec = RECORD
                       qLink:     QElemPtr; {next queue entry}
                       qType:     INTEGER;  {queue type}
                       ioTrap:    INTEGER;  {routine trap}
                       ioCmdAddr: Ptr;      {routine address}
                         . . .              {rest of block}
                     END;
```

QLink points to the next entry in the queue, and qType indicates the queue type, which must always be ORD(ioQType). IOTrap and ioCmdAddr contain the trap word and address of the File Manager routine that was called. You can get a pointer to the file I/O queue by calling the File Manager function GetFSQHdr.

FUNCTION GetFSQHdr : QHdrPtr; [Pascal only]

GetFSQHdr returns a pointer to the file I/O queue.

Assembly-language note: To access the contents of the file I/O queue from assembly language, you can use offsets from the address of the global variable fsQHdr. Bit 7 of the queue flags is set if there are any entries in the queue; you can use the global constant qInUse to test the value of bit 7.

Volume Control Blocks

Each time a volume is mounted, its volume information is read from the volume and used to build a new volume control block in the volume-control-block queue (unless an ejected or off-line volume is being remounted). A copy of the volume block map is also read from the volume and placed in the system heap, and a volume buffer is created on the system heap.

The volume-control-block queue is a list of the volume control blocks for all mounted volumes, maintained on the system heap. It's a standard Operating System queue (described in the appendix), and each entry in the volume-control-block queue is a volume control block. A volume control block is a 94-byte nonrelocatable block that contains volume-specific information, including the first 64 bytes of the master directory block (bytes 8 to 72 of the volume control block match bytes Ø to 64 of the volume information). It has the following structure:

```
TYPE VCB = RECORD
             qLink:        QElemPtr;     {next queue entry}
             qType:        INTEGER;      {not used}
             vcbFlags:     INTEGER;      {bit 15=1 if dirty}
             vcbSigWord:   INTEGER;      {always $D2D7}
             vcbCrDate:    LongInt;      {date volume was initialized}
             vcbLsBkUp:    LongInt;      {date of last backup}
             vcbAtrb:      INTEGER;      {volume attributes}
             vcbNmFls:     INTEGER;      {number of files in directory}
             vcbDirSt:     INTEGER;      {directory's first block}
             vcbBlLn:      INTEGER;      {length of file directory}
             vcbNmBlks:    INTEGER;      {number of allocation blocks}
             vcbAlBlkSiz:  LongInt;      {size of allocation blocks}
             vcbClpSiz:    LongInt;      {number of bytes to allocate}
             vcbAlBlSt:    INTEGER;      {first block in block map}
             vcbNxtFNum:   LongInt;      {next unused file number}
             vcbFreeBks:   INTEGER;      {number of unused blocks}
             vcbVN:        STRING[27];   {volume name}
             vcbDrvNum:    INTEGER;      {drive number}
             vcbDRefNum:   INTEGER;      {driver reference number}
             vcbFSID:      INTEGER;      {file system identifier}
             vcbVRefNum:   INTEGER;      {volume reference number}
             vcbMAdr:      Ptr;          {location of block map}
             vcbBufAdr:    Ptr;          {location of volume buffer}
             vcbMLen:      INTEGER;      {number of bytes in block map}
             vcbDirIndex:  INTEGER;      {used internally}
             vcbDirBlk:    INTEGER       {used internally}
           END;
```

Bit 15 of vcbFlags is set if the volume information has been changed by a routine call since the volume was last affected by a FlushVol call. VCBAtr contains the volume attributes. Each bit, if set, indicates the following:

Bit	Meaning
Ø-2	Inconsistencies were found between the volume information and the file directory when the volume was mounted
6	Volume is busy (one or more files are open)
7	Volume is locked by hardware
15	Volume is locked by software

VCBDirSt contains the number of the first logical block of the file directory; vcbNmBlks, the number of allocation blocks on the volume; vcbAlBlSt, the number of the first logical block in the block map; and vcbFreeBks, the number of unused allocation blocks on the volume.

VCBDrvNum contains the drive number of the drive on which the volume is mounted; vcbDRefNum contains the driver reference number of the driver used to access on volume is mounted. When a mounted volume is placed off-line, vcbDrvNum is cleared. When ejected, vcbDrvNum is cleared and vcbDRefNum is set to the negative of vcbDrvNum (becoming a positive number). VCBFSID identifies the file system handling the volume; it's Ø for volumes handled by the File Manager, and nonzero for volumes handled by other file systems.

When a volume is placed off-line, its buffer and block map are deallocated. When a volume is unmounted, its volume control block is removed from the volume-control-block queue.

You can get a pointer to the volume-control-block queue by calling the File Manager function GetVCBQHdr.

```
FUNCTION GetVCBQHdr : QHdrPtr;   [Pascal only]
```

GetVCBQHdr returns a pointer to the volume-control-block queue.

> Assembly-language note: To access the contents of the volume-control-block queue from assembly language, you can use offsets from the address of the global variable vcbQHdr. Bit 7 of the queue flags is set if there are any entries in the queue; you can use the global constant qInUse to test the value of bit 7. The default volume's volume control block is pointed to by the global variable defVCBPtr.

File Control Blocks

Each time a file is opened, the file's directory entry is used to build a 3Ø-byte file control block in the file-control-block buffer, which contains information about all access paths. The file-control-block buffer can contain up to 12 file control blocks (since up to 12 paths can be open at once), and is a 362-byte (2 + 3Ø bytes*12 paths) nonrelocatable block on the system heap (see Figure 1Ø).

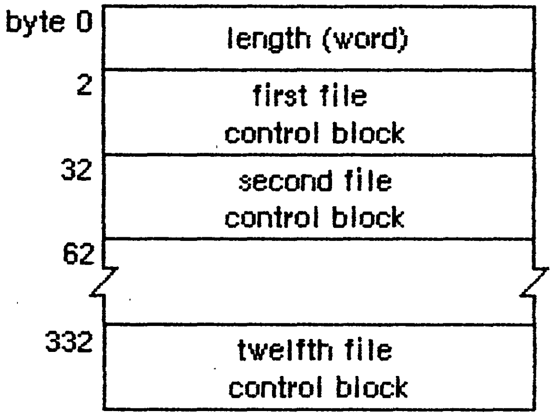


Figure 1Ø. The File-Control-Block Buffer

You can refer to the file-control-block buffer by using the global variable fcbSPtr, which points to the length word. Each file control block contains 3Ø bytes of information about an access path (Figure 11).

byte 0	fcbFlNum (long word)	file number
4	fcbMdRByt (byte)	flags
5	fcbTypByt (byte)	version number
6	fcbSBlk (word)	first allocation block of file
8	fcbEOF (long word)	logical end-of-file
12	fcbPLen (long word)	physical end-of-file
16	fcbCrPs (long word)	mark
20	fcbVPtr (pointer)	location of volume control block
24	fcbBfAdr (pointer)	location of access path buffer
28	fcbFlPos (word)	for internal use of File Manager

Figure 11. A File Control Block

Bit 7 of fcbMdRByt is set if the file has been changed since it was last flushed; bit 1 is set if the entry describes a resource fork; bit Ø is set if data can be written to the file.

Files Tags in Memory

As mentioned previously, logical blocks on Macintosh-initialized volumes contain 12 bytes of file tags. Normally, you'll never need to know about file tags, and the File Manager will let you read and write only the 512 bytes of standard information in each logical block. The File Manager automatically removes the file tags from each logical block it reads into memory (Figure 12) and places them at the location referred to by the global variable tagData + 2. It replaces the last four bytes of the file tags with the number of the logical block from which the file was read (leaving a total of ten bytes).

byte 0	file number (long word)	file number
4	fork type (byte)	bit 1 = 1 if resource fork
5	file attributes (byte)	bit 0 = 1 if locked
6	file sequence (word)	logical block sequence number
8	logical block number (word)	logical block

Figure 12. File Tags in Memory

(note)

> Access path buffers and volume buffers are 522 bytes long in order to contain the ten bytes of file tags and 512 bytes of standard information.

The Drive Queue

Disk drives connected to the Macintosh are opened when the system starts up, and information describing each is placed in the drive queue. It's a standard Operating System queue (described in the appendix), and each entry in the drive queue has the following structure:

```
TYPE DrvQEl = RECORD
              { flags:      LongInt; }
                qLink:      QElemPtr; {next queue entry}
                qType:      INTEGER;  {not used}
                dQDrive:    INTEGER;  {drive number}
                dQRefNum:   INTEGER;  {driver reference number}
                dQFSID:     INTEGER;  {file-system identifier}
                dQDrvSize:  INTEGER   {optional:  number of blocks}
              END;
```

QDrvNum contains the drive number of the drive on which the volume is mounted; qDRefNum contains the driver reference number of the driver controlling the device on which the volume is mounted. QFSID identifies the file system handling the volume in the drive; it's Ø for volumes handled by the File Manager, and nonzero for volumes handled by other file systems. If the volume isn't a 3-1/2 inch disk, dQDrvSize contains the number of 512-byte blocks on the volume mounted in this drive; if the volume is a 3-1/2 inch disk, this field isn't used.

Assembly-language note: The first four bytes in a drive queue entry are accessible only from assembly language, and contain the following:

Byte	Contents
Ø	Bit 7=1 if volume is locked
1	Ø if no disk in drive; 1 or 2 if disk in drive; 8 if nonejectable disk in drive; $FC-$FF if disk was ejected within last 1.5 seconds
2	used internally during system startup
3	Bit 7=Ø if disk is single-sided

You can get a pointer to the drive queue by calling the File Manager function GetDrvQHdr:

```
FUNCTION GetDrvQHdr : QHdrPtr;  [Pascal only]
```

GetDrvQHdr returns a pointer to the qFlags field.

Assembly-language note: To access the contents of the drive queue from assembly language, you can use offsets from the address of the global variable drvQHdr.

The drive queue can support any number of drives, limited only by memory space.

USING AN EXTERNAL FILE SYSTEM

The File Manager is used to access files on Macintosh-initialized volumes. If you want to access files on nonMacintosh-initialized volumes, you must write your own external file system and volume-initializing program. After the external file system has been written, it must be used in conjunction with the File Manager as described in this section.

Before any File Manager routines are called, you must place the memory location of the external file system in the global variable toExtFS, and link the drive(s) accessed by your file system into the drive queue. As each nonMacintosh-initialized volume is mounted, you must create your own volume control block for each mounted volume and link each one into the volume-control-block queue. As each access path is opened, you must create your own file control block and add it to the file-control-block buffer.

All SetVol, GetVol, and GetVolInfo calls then can be handled by the File Manager via the volume-control-block queue and drive queue; external file systems needn't support these calls.

When an application calls any other File Manager routine accessing a nonMacintosh-initialized volume, the File Manager passes control to the address contained in toExtFS (if toExtFS is Ø, the File Manager returns directly to the application with the result code extFSErr). The external file system must then use the information in the file I/O queue to handle the call as it wishes, set the result code noErr, and return control to the File Manager. Control is passed to an external file system for the following specific routine calls:

- for MountVol if the drive queue entry for the requested drive has a nonzero file-system identifier

- for Create, Open, OpenRF, GetFileInfo, SetFileInfo, SetFilLock, RstFilLock, SetFilType, Rename, Delete, FlushVol, Eject, OffLine, and UnmountVol, if the volume control block for the requested file or volume has a nonzero file-system identifier

- for Close, Read, Write, Allocate, GetEOF, SetEOF, GetFPos, SetFPos, and FlushFile, if the file control block for the requested file points to a volume control block with a nonzero file-system identifier

APPENDIX -- OPERATING SYSTEM QUEUES

*** This appendix will eventually be part of the Operating System Utilities manual. ***

Some of the information used by the Operating System is stored in data structures called queues. A queue is a list of identically structured entries linked together by pointers. Queues are used to keep track of vertical retrace tasks, I/O requests, disk drives, events, and mounted volumes.

The structure of a standard Operating System queue is as follows:

```
TYPE QHdr = RECORD
              qFlags: INTEGER;  {queue flags}
              qHead:  QElemPtr; {first queue entry}
              qTail:  QElemPtr  {last queue entry}
            END;

     QHdrPtr = ^QHdr;
```

QFlags contains information that's different for each queue type. QHead points to the first entry in the queue, and qTail points to the last entry in the queue. The entries within each type of queue are different, since each type of queue contains different information. The Operating System uses the following variant record to access queue entries:

```
TYPE QTypes = (dummyType,
               vType,     {vertical retrace queue}
               ioQType,   {I/O request queue}
               drvQType,  {drive queue}
               evType,    {event queue}
               fsQType);  {volume-control-block queue}

     QElem = RECORD
               CASE QTypes OF
                 (vblQElem: VBLTask);
                 (ioQElem:  ParamBlockRec);
                 (drvQElem: DrvQEl);
                 (evQElem:  EvQEl);
                 (vcbQElem: VCB)
             END;

     QElemPtr = ^QElem;
```

The exact structure of the entries in each type of Operating System queue is described in the manual that discusses that queue in detail.

Assembly-language note: The values given in the Pascal QTypes set are available to assembly-language programmers as the global constants vType, ioQType, evType, and fsQType (there is no global constant corresponding to drvQType).

SUMMARY OF THE FILE MANAGER

Constants

```
CONST { Flags in file information used by the Finder }

      fHasBundle = 32; {set if file has a bundle}
      fInvisible = 64; {set if file's icon is invisible}
      fTrash     = -3; {file is in trash window}
      fDesktop   = -2; {file is on desktop}
      fDisk      =  Ø; {file is in disk window}

      { Values for posMode and ioPosMode }

      fsAtMark    = Ø; {at current position of mark }
                       { (posOff or ioPosOffset ignored)}
      fsFromStart = 1; {offset relative to beginning of file}
      fsFromLEOF  = 2; {offset relative to logical end-of-file}
      fsFromMark  = 3; {offset relative to current mark}

      { Values for requesting read/write access }

      fsCurPerm  = Ø; {whatever is currently allowed}
      fsRdPerm   = 1; {request to read only}
      fsWrPerm   = 2; {request to write only}
      fsRdWrPerm = 3; {request to read and write}
```

(See also the result codes at end of this summary.)

Data Structures

```
TYPE FInfo = RECORD
               fdType:     OSType;  {file type}
               fdCreator:  OSType;  {file's creator}
               fdFlags:    INTEGER; {flags}
               fdLocation: Point;   {file's location}
               fdFldr:     INTEGER  {file's window}
             END;

     ParamBlkPtr    = ^ParamBlockRec;

     ParamBlkType   = (ioParam, fileParam, volumeParam, cntrlParam);
```

```
ParamBlockRec = RECORD
 qLink:        QElemPtr;  {next queue entry}
 qType:        INTEGER;   {queue type}
 ioTrap:       INTEGER;   {routine trap}
 ioCmdAddr:    Ptr;       {routine address}
 ioCompletion: ProcPtr;   {completion routine}
 ioResult:     OSErr;     {result code}
 ioNamePtr:    StringPtr; {volume or file name}
 ioVRefNum:    INTEGER;   {volume reference or }
                          { drive number}
 CASE ParamBlkType OF
  ioParam:
   (ioRefNum:    INTEGER;    {path reference number}
    ioVersNum:   SignedByte; {version number}
    ioPermssn:   SignedByte; {read/write permission}
    ioMisc:      Ptr;        {miscellaneous}
    ioBuffer:    Ptr;        {data buffer}
    ioReqCount:  LongInt;    {requested number of bytes}
    ioActCount:  LongInt;    {actual number of bytes}
    ioPosMode:   INTEGER;    {newline character and type of }
                             { positioning operation}
    ioPosOffset: LongInt);   {size of positioning offset}
  fileParam:
   (ioFRefNum:    INTEGER;    {path reference number}
    ioFVersNum:   SignedByte; {version number}
    filler1:      SignedByte; {not used}
    ioFDirIndex:  INTEGER;    {file number}
    ioFlAttrib:   SignedByte; {file attributes}
    ioFlVersNum:  SignedByte; {version number}
    ioFlFndrInfo: FInfo;      {information used by the Finder}
    ioFlNum:      LongInt;    {file number}
    ioFlStBlk:    INTEGER;    {first allocation block of data fork}
    ioFlLgLen:    LongInt;    {logical end-of-file of data fork}
    ioFlPyLen:    LongInt;    {physical end-of-file of data fork}
    ioFlRStBlk:   INTEGER;    {first allocation block of resource fork}
    ioFlRLgLen:   LongInt;    {logical end-of-file of resource fork}
    ioFlRPyLen:   LongInt;    {physical end-of-file of resource fork}
    ioFlCrDat:    LongInt;    {date and time of creation}
    ioFlMdDat:    LongInt);   {date and time of last modification}
  volumeParam:
   (filler2:     LongInt;  {not used}
    ioVolIndex:  INTEGER;  {volume index}
    ioVCrDate:   LongInt;  {date and time of initialization}
    ioVLsBkUp:   LongInt;  {date and time of last volume backup}
    ioVAtrb:     INTEGER;  {bit 15=1 if volume locked}
    ioVNmFls:    INTEGER;  {number of files in file directory}
    ioVDirSt:    INTEGER;  {first block of file directory}
    ioVBlLn:     INTEGER;  {number of blocks in file directory}
    ioVNmAlBlks: INTEGER;  {number of allocation blocks on volume}
    ioVAlBlkSiz: LongInt;  {number of bytes per allocation block}
    ioVClpSiz:   LongInt;  {number of bytes to allocate}
    ioAlBlSt:    INTEGER;  {first block in volume block map}
    ioVNxtFNum:  LongInt;  {next free file number}
    ioVFrBlk:    INTEGER); {number of free allocation blocks}
  cntrlParam:
    . . . {used by Device Manager}
END;

     VCB = RECORD
             qLink:       QElemPtr;   {next queue entry}
             qType:       INTEGER;    {not used}
             vcbFlags:    INTEGER;    {bit 15=1 if dirty}
             vcbSigWord:  INTEGER;    {always $D2D7}
             vcbCrDate:   LongInt;    {date volume was initialized}
             vcbLsBkUp:   LongInt;    {date of last backup}
             vcbAtrb:     INTEGER;    {volume attributes}
             vcbNmFls:    INTEGER;    {number of files in directory}
             vcbDirSt:    INTEGER;    {directory's first block}
             vcbBlLn:     INTEGER;    {length of file directory}
             vcbNmBlks:   INTEGER;    {number of allocation blocks}
             vcbAlBlkSiz: LongInt;    {size of allocation blocks}
             vcbClpSiz:   LongInt;    {number of bytes to allocate}
             vcbAlBlSt:   INTEGER;    {first block in block map}
             vcbNxtFNum:  LongInt;    {next unused file number}
             vcbFreeBks:  INTEGER;    {number of unused blocks}
             vcbVN:       STRING[27]; {volume name}
             vcbDrvNum:   INTEGER;    {drive number}
             vcbDRefNum:  INTEGER;    {driver reference number}
             vcbFSID:     INTEGER;    {file system identifier}
             vcbVRefNum:  INTEGER;    {volume reference number}
             vcbMAdr:     Ptr;        {location of block map}
             vcbBufAdr:   Ptr;        {location of volume buffer}
             vcbMLen:     INTEGER;    {number of bytes in block map}
             vcbDirIndex: INTEGER;    {used internally}
             vcbDirBlk:   INTEGER     {used internally}
           END;

     DrvQEl = RECORD
                qLink:     QElemPtr; {next queue entry}
                qType:     INTEGER;  {not used}
                dQDrive:   INTEGER;  {drive number}
                dQRefNum:  INTEGER;  {driver reference number}
                dQFSID:    INTEGER;  {file-system identifier}
                dQDrvSize: INTEGER   {number of logical blocks}
              END;
```

High-Level Routines [Pascal only]

Accessing Volumes

```
FUNCTION GetVInfo   (drvNum: INTEGER; volName: StringPtr; VAR
                     vRefNum: INTEGER; VAR freeBytes: LongInt) :
                     OSErr;
FUNCTION GetVol     (volName: StringPtr; VAR vRefNum: INTEGER) :
                     OSErr;
```

```
FUNCTION SetVol     (volName: StringPtr; vRefNum: INTEGER) : OSErr;
FUNCTION FlushVol   (volName: StringPtr; vRefNum: INTEGER) : OSErr;
FUNCTION UnmountVol (volName: StringPtr; vRefNum: INTEGER) : OSErr;
FUNCTION Eject      (volName: StringPtr; vRefNum: INTEGER) : OSErr;
```

Changing File Contents

```
FUNCTION Create   (fileName: Str255; vRefNum: INTEGER; creator:
                   OSType; fileType: OSType) : OSErr;
FUNCTION FSOpen   (fileName: Str255; vRefNum: INTEGER; VAR
                   refNum: INTEGER) :  OSErr;
FUNCTION FSRead   (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr)
                   : OSErr;
FUNCTION FSWrite  (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr)
                   : OSErr;
FUNCTION GetFPos  (refNum: INTEGER; VAR filePos: LongInt) : OSErr;
FUNCTION SetFPos  (refNum: INTEGER; posMode: INTEGER; posOff: LongInt)
                   : OSErr;
FUNCTION GetEOF   (refNum: INTEGER; VAR logEOF: LongInt) : OSErr;
FUNCTION SetEOF   (refNum: INTEGER; logEOF: LongInt) : OSErr;
FUNCTION Allocate (refNum: INTEGER; VAR count: LongInt) : OSErr;
FUNCTION FSClose  (refNum: INTEGER) : OSErr;
```

Changing Information About Files

```
FUNCTION GetFInfo (fileName: Str255; vRefNum: INTEGER; VAR
                   fndrInfo: FInfo) : OSErr;
FUNCTION SetFInfo (fileName: Str255; vRefNum: INTEGER; fndrInfo:
                   FInfo) : OSErr;
FUNCTION SetFLock (fileName: Str255; vRefNum: INTEGER) : OSErr;
FUNCTION RstFLock (fileName: Str255; vRefNum: INTEGER) : OSErr;
FUNCTION Rename   (oldName: Str255; vRefNum: INTEGER; newName:
                   Str255) : OSErr;
FUNCTION FSDelete (fileName: Str255; vRefNum: INTEGER) : OSErr;
```

Low-Level Routines

Initializing the File I/O Queue

```
PROCEDURE InitQueue;
```

Accessing Volumes

```
FUNCTION PBMountVol   (paramBlock: ParmBlkPtr) : OSErr;
FUNCTION PBGetVolInfo (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBGetVol     (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBSetVol     (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBFlshVol    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBUnmountVol (paramBlock: ParmBlkPtr) : OSErr;
FUNCTION PBOffLine    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBEject      (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Changing File Contents

```
FUNCTION PBCreate    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBOpen      (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBOpenRF    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBRead      (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBWrite     (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBGetFPos   (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBSetFPos   (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBGetEOF    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBSetEOF    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBAllocate  (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBFlshFile  (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBClose     (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Changing Information About Files

```
FUNCTION PBGetFInfo (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBSetFInfo (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBSetFLock (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBRstFLock (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBSetFVers (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBRename   (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBDelete   (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Accessing Queues [Pascal only]

```
FUNCTION GetFSQHdr  : QHdrPtr;
FUNCTION GetVCBQHdr : QHdrPtr;
FUNCTION GetDrvQHdr : QHdrPtr;
```

Assembly-Language Information

Constants

```
; Flags in file information used by the Finder

fsQType        .EQU     5    ;I/O request queue entry type
fHasBundle     .EQU     5    ;set if file has a bundle
fInvisible     .EQU     6    ;set if file's icon is invisible

; Flag set when queue is in use

qInUse         .EQU     7    ;set if queue is in use

; Flags for testing trap words

asnycTrpBit    .EQU    10    ;set in trap word for an asynchronous call
noQueueBit     .EQU     9    ;set in trap word for immediate execution
```

Structure of File Information Used by the Finder

fdType	File type
fdCreator	File's creator
fdFlags	Flags
fdLocation	File's location
fdFldr	File's window

Standard Parameter Block Data Structure

qLink	Next queue entry
qType	Queue type
ioTrap	Routine trap
ioCmdAddr	Routine address
ioCompletion	Completion routine
ioResult	Result code
ioFileName	File name (and possibly volume name)
ioVNPtr	Volume name
ioVRefNum	Volume reference number
ioDrvNum	Drive number

I/O Parameter Block Data Structure

ioRefNum	Path reference number
ioFileType	Version number
ioPermssn	Read/write permission
ioNewName	New file or volume name for Rename
ioLEOF	Logical end-of-file for SetEOF
ioOwnBuf	Access path buffer
ioNewType	New version number for SetFilType
ioBuffer	Data buffer
ioReqCount	Requested number of bytes
ioActCount	Actual number of bytes
ioPosMode	Newline character and type of positioning operation
ioPosOffset	Size of positioning offset

File Information Parameter Block Data Structure

ioRefNum	Path reference number
ioFileType	Version number
ioFDirIndex	File number
ioFlAttrib	File attributes
ioFFlType	Version number
ioFlUsrWds	Information used by the Finder
ioFFlNum	File number
ioFlStBlk	First allocation block of data fork
ioFlLgLen	Logical end-of-file of data fork
ioFlPyLen	Physical end-of-file of data fork
ioFlRStBlk	First allocation block of resource fork
ioFlRLgLen	Logical end-of-file of resource fork
ioFlRPyLen	Physical end-of-file of resource fork
ioFlCrDat	Date and time file was created
ioFlMdDat	Date and time file was last modified

Volume Information Parameter Block Data Structure

ioVolIndex	Volume index number
ioVCrDate	Date and time volume was initialized
ioVLsBkUp	Date and time of last volume backup
ioVAtrb	Bit 15=1 if volume is locked
ioVNmFls	Number of files in file directory
ioVDirSt	First block of file directory
ioVBlLn	Number of blocks in file directory
ioVNmAlBlks	Number of allocation blocks on volume
ioVAlBlkSiz	Number of bytes per allocation block
ioVClpSiz	Number of bytes to allocate
ioAlBlSt	First block in volume block map
ioVNxtFNum	Next free file number
ioVFrBlk	Number of free allocation blocks

Volume Information Data Structure

drSigWord	Always $D2D7
drCrDate	Date and time of initialization
drLsBkUp	Date and time of last backup
drAtrb	Volume attributes
drNmFls	Number of files in file directory
drDirSt	First logical block of file directory
drBlLen	Number of logical blocks in file directory
drNmAlBlks	Number of allocation blocks on volume
drAlBlkSiz	Size of allocation blocks
drClpSiz	Number of bytes to allocate
drAlBlSt	Logical block number of first allocation block
drNxtFNum	Next unused file number
drFreeBks	Number of unused allocation blocks
drVN	Length and characters of volume name

File Directory Entry Data Structure

flFlags	Bit 7=1 if entry used; bit Ø=1 if file locked
flTyp	Version number
flUsrWds	Information used by the Finder
flFlNum	File number
flStBlk	First allocation block of data fork
flLgLen	Data fork's logical end-of-file
flPyLen	Data fork's physical end-of-file
flRStBlk	First allocation block of resource fork
flRLgLen	Resource fork's logical end-of-file
flRPyLen	Resource fork's physical end-of-file
flCrDat	Date and time file was created
flMdDat	Date and time file was last modified
flName	Length and characters of file name

Queue Header Data Structure

qFlags	Queue flags
qHead	Pointer to first queue entry
qTail	Pointer to last queue entry

Volume Control Block Data Structure

qLink	Next queue entry
qType	Not used
vcbFlags	Bit 15=1 if volume control block is dirty
vcbSigWord	Always $D2D7
vcbCrDate	Date and time volume was initialized
vcbLsBkUp	Date and time last backup copy was made
vcbAtrb	Volume attributes
vcbNmFls	Number of files in directory
vcbDirSt	First logical block of file directory

vcbBlLn	Length of file directory
vcbNmBlks	Number of allocation blocks on volume
vcbAlBlkSiz	Size of allocation blocks
vcbClpSiz	Number of bytes to allocate
vcbAlBlSt	First logical block in block map
vcbNxtFNum	Next unused file number
vcbFreeBks	Number of unused allocation blocks
vcbVN	Length and characters of volume name
vcbDrvNum	Drive number of drive in which volume is mounted
vcbDRefNum	Driver reference number of driver for drive in which volume is mounted
vcbFSID	ID for file system handling volume
vcbVRefNum	Volume reference number
vcbMAdr	Memory location of volume block map
vcbBufAdr	Memory location of volume buffer
vcbMLen	Number of bytes in volume block map
vcbDirIndex	For internal File Manager use
vcbDirBlk	For internal File Manager use

File Control Block Data Structure

fcbFlNum	File number
fcbMdRByt	Flags
fcbTypByt	Version number
fcbSBlk	First allocation block of file
fcbEOF	Logical end-of-file
fcbPLen	Physical end-of-file
fcbCrPs	Mark
fcbVPtr	Location of volume control block
fcbBfAdr	Location of access path buffer
fcbFlPos	For internal use of File Manager

File Control Block Data Structure

qLink	Next queue entry
qType	Always drvType
dQDrive	Drive number
dQRefNum	Driver reference number
dQFSID	File system ID
dQDrvSize	Number of logical blocks

Macro Names

Routine name	Macro name
InitQueue	_InitQueue
PBMountVol	_MountVol
PBGetVolInfo	_GetVolInfo
PBGetVol	_GetVol
PBSetVol	_SetVol
PBFlshVol	_FlushVol
PBUnmountVol	_UnmountVol

PBOffLine	_OffLine
PBEject	_Eject
PBCreate	_Create
PBOpen	_Open
PBOpenRF	_OpenRF
PBRead	_Read
PBWrite	_Write
PBGetFPos	_GetFPos
PBSetFPos	_SetFPos
PBGetEOF	_GetEOF
PBSetEOF	_SetEOF
PBAllocate	_Allocate
PBFlshFile	_FlushFile
PBClose	_Close
PBGetFInfo	_GetFileInfo
PBSetFInfo	_SetFileInfo
PBSetFLock	_SetFilLock
PBRstFLock	_RstFilLock
PBSetFVers	_SetFilType
PBRename	_Rename
PBDelete	_Delete

Variables

Name	Size	Contents
fsQHdr	4 bytes	File I/O queue
vcbQHdr	4 bytes	Volume-control-block queue
defVCBPtr	4 bytes	Pointer to default volume control block
fcbSPtr	4 bytes	Pointer to file-control-block buffer
tagData + 2	4 bytes	Location of file tags
drvQHdr	4 bytes	Drive queue
toExtFS	4 bytes	Pointer to external file system

Result Codes

These values are available as predefined constants in both Pascal and assembly language.

Name	Value	Meaning
badMDBErr	-6Ø	Master directory block is bad; must reinitialize volume
bdNamErr	-37	Bad file name or volume name (perhaps zero-length)
dirFulErr	-33	File directory full
dskFulErr	-34	All allocation blocks on the volume are full
dupFNErr	-48	A file with the specified name already exists
eofErr	-39	Logical end-of-file reached during read operation
extFSErr	-58	External file system; file-system identifier is nonzero, or path reference number is greater than 1Ø24

fBsyErr	-47	One or more files are open
fLckdErr	-45	File locked
fnfErr	-43	File not found
fnOpnErr	-38	File not open
fsRnErr	-59	Problem during Rename
ioErr	-36	Disk I/O error
mFulErr	-41	System heap is full
noErr	Ø	No error
nsDrvErr	-56	Specified drive number doesn't match any number in the drive queue
noMacDskErr	-57	Volume lacks Macintosh-format directory
nsvErr	-35	Specified volume doesn't exist
opWrErr	-49	The read/write permission of only one access path to a file can allow writing
paramErr	-5Ø	Parameters don't specify an existing volume, and there's no default volume
permErr	-54	Read/write permission doesn't allow writing
posErr	-4Ø	Attempted to position before start of file
rfNumErr	-51	Reference number specifies nonexistent access path
tmfoErr	-42	Only 12 files can be open simultaneously
volOffLinErr	-53	Volume not on-line
volOnLinErr	-55	Volume specified is already mounted and on-line
vLckdErr	-46	Volume is locked by a software flag
wrPermErr	-61	Read/write permission or open permission doesn't allow writing
wPrErr	-44	Volume is locked by a hardware setting

GLOSSARY

access path: A description of the route that the File Manager follows to access a file; created when a file is opened.

access path buffer: Memory used by the File Manager to transfer data between an application and a file.

allocation block: Volume space composed of an integral number of logical blocks.

asynchronous execution: During asynchronous execution of a File Manager routine, the calling application is free to perform other tasks.

block map: Same as volume allocation block map.

closed file: A file without an access path. Closed files cannot be read from or written to.

completion routine: Any application-defined code to be executed when an asynchronous call to a File Manager routine is completed.

data buffer: Heap space containing information to be written to a file or driver from an application, or read from a file or driver to an application.

data fork: The part of a file that contains data accessed via the File Manager.

default volume: A volume that will receive I/O during a File Manager routine call, whenever no other volume is specified.

drive number: A number used to identify a disk drive. The internal drive is number 1, and the external drive is number 2.

drive queue: A list of disk drives connected to the Macintosh.

end-of-file: See logical end-of-file or physical end-of-file.

file: A named, ordered sequence of bytes; a principal means by which data is stored and transmitted on the Macintosh.

file control block: 3Ø bytes of system heap space in a file-control-block buffer containing information about an access path.

file-control-block buffer: A 362-byte nonrelocatable block containing one file control block for each access path.

file directory: The part of a volume that contains descriptions and locations of all the files on the volume.

file I/O queue: A queue containing parameter blocks for all I/O requests to the File Manager.

file name: A sequence of up to 255 characters that identifies a file.

file number: A unique number assigned to a file, which the File Manager uses to distinguish it from other files on the volume. A file number specifies the file's entry in a file directory.

file tags: Information associated with each logical block, designed to allow reconstruction of files on a volume whose directory or other file-access information has been destroyed.

fork: One of the two parts of a file; see data fork and resource fork.

I/O request: A request for input from or output to a file or device driver; caused by calling a File Manager or Device Manager routine asynchronously.

locked file: A file whose data cannot be changed.

locked volume: A volume whose data cannot be changed. Volumes can be locked by either a software flag or a hardware setting.

logical block: Volume space composed of 512 consecutive bytes of standard information and an additional number of bytes of disk-driver specific information.

logical end-of-file: The position of one byte past the last byte in a file; equal to the actual number of bytes in the file.

mark: The position of the next byte in a file that will be read or written.

master directory block: Part of the data structure of a volume; contains the volume information and the first 448 bytes of the block map.

mounted volume: A volume that previously was inserted into a disk drive and had descriptive information read from it by the File Manager.

newline character: Any ASCII character, but usually Return (ASCII code $ØD), that indicates the end of a sequence of bytes.

newline mode: A mode of reading data where the end of the data is indicated by a newline character (and not by a specific byte count).

off-line volume: A mounted volume with all but 94 bytes of its descriptive information released.

on-line volume: A mounted volume with its volume buffer and descriptive information contained in memory.

open file: A file with an access path. Open files can be read from and written to.

open permission: Information about a file that indicates whether the file can be read from, written to, or both.

parameter block: Memory space used to transfer information between applications and the File Manager.

path reference number: A number that uniquely identifies an individual access path; assigned when the access path is created.

physical end-of-file: The position of one byte past the last allocation block of a file; equal to 1 more than the maximum number of bytes the file can contain.

read/write permission: Information associated with an access path that indicates whether the file can be read from, written to, both read from and written to, or whatever the file's open permission allows.

resource fork: The part of a file that contains the resources used by an application (such as menus, fonts, and icons) and also the application code itself; usually accessed via the Resource Manager.

synchronous execution: During synchronous execution of a File Manager routine, the calling application must wait until the routine is completed, and isn't free to perform any other task.

unmounted volume: A volume that hasn't been inserted into a disk drive and had descriptive information read from it, or a volume that previously was mounted and has since had the memory used by it released.

version number: A number from Ø to 255 used to distinguish between files with the same name.

volume: A piece of storage medium formatted to contain files; usually a disk or part of a disk. The 3 1/2-inch Macintosh disks are one volume.

volume allocation block map: A list of 12-bit entries, one for each allocation block, that indicate whether the block is currently allocated to a file, whether it's free for use, or which block is next in the file. Block maps exist both on volumes and in memory.

volume attributes: Information contained on volumes and in memory indicating whether the volume is locked, has one or more files open (in memory only), and whether the volume control block matches the volume information (in memory only).

volume buffer: Memory used initially to load the master directory block, and used thereafter for reading from files that are opened without an access path buffer.

volume control block: A 9Ø-byte nonrelocatable block that contains volume-specific information, including the first 64 bytes of the master directory block.

volume-control-block queue: A list of the volume control blocks for all mounted volumes.

volume index: A number identifying a mounted volume listed in the volume-control-block queue. The first volume in the queue has an index of 1, and so on.

volume information: Volume-specific information contained on a volume; includes the volume name, number of files on the volume, and so on.

volume name: A sequence of up to 27 printing characters that identifies a volume; always followed by a colon (:) to distinguish it from a file name.

volume reference number: A unique number assigned to a volume as it's mounted, used to refer to the volume.

Printing From Macintosh Applications /PRINTING/PRINT

See Also: The Resource Manager: A Programmer's Guide
QuickDraw: A Programmer's Guide
The Font Manager: A Programmer's Guide
The Dialog Manager: A Programmer's Guide
The Structure of a Macintosh Application
Programming Macintosh Applications in Assembly Language

Modification History: First Draft S. Chernicoff & B. Hacker 6/11/84

ABSTRACT

Macintosh applications can print information on any variety of printer the user has connected to the Macintosh by calling Printing Manager routines. Advanced programmers can also call the Printer Driver to implement alternate, low-level printing techniques. This manual describes the Printing Manager and Printer Driver.

TABLE OF CONTENTS

ABOUT THIS MANUAL

Macintosh applications can print information on any variety of printer the user has connected to the Macintosh by calling the Printing Manager routines in the User Interface Toolbox. Advanced programmers can also call the Printer Driver to implement alternate, low-level printing techniques. This manual describes the Printing Manager and Printer Driver. *** It will eventually become part of the comprehensive Inside Macintosh manual. ***

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 described in the Resource Manager manual
- the use of QuickDraw, as described in the QuickDraw manual, particularly bit images, rectangles, bitMaps, and pictures
- the use of fonts, as described in the Font Manager manual
- the basic concepts of dialogs, as described in the Dialog Manager manual
- files and volumes, as described in the File Manager manual
- device drivers, as described in the Device Manager manual, *** doesn't yet exist *** if you're interested in writing your own Printer Driver

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 Printing Manager and what you can do with it. It then discusses the basics about printing: the various methods of printing available; the relationship between printing and the Finder; and the Printing Manager's use of dialogs and data structures, the most important of which is the print record.

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

Following these descriptions are sections that won't interest all readers. Special information is given about the Printer Driver and the format of resource files used when printing, for programmers interested in writing their own Printer Driver.

Finally, there's a summary of the Printing Manager for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE PRINTING MANAGER

The Printing Manager is the part of the Macintosh User Interface Toolbox that's used to print text or graphics on a printer. It's not contained in the Macintosh ROM; it must be read from a resource file before it can be used. The Printing Manager provides your application with:

- two standard printing methods, and the ability to define two more

- a standard dialog for the user to specify the paper size and page orientation they're using, so you can easily implement a Page Setup command in your File menu

- a standard dialog for the user to specify the method of printing, which pages to print, and so on, so you can easily implement a Print command in your File menu

- the ability to perform background processing while the Printing Manager is printing

- a way to abort printing when the user types Command-period

The Printing Manager is designed such that an application need never be concerned with what kind of printer the user has connected to the Macintosh; an application uses the same routine calls to print with all varieties of printers.

This printer independence is possible because the Printing Manager uses separate, printer-specific code to implement its routines for each different variety of printer. While the code for some Printing Manager routines (such as those that begin and end printing sessions), is contained wholly within the Printing Manager itself, the code for other routines (such as those that do the actual printing) depends on the printer being used and is contained in a separate printer resource file on the user's disk. The Printing Manager dispatches calls to these routines, first loading the code into memory if necessary.

Although the actual routines of the Printing Manager differ for each variety of printer, your application uses the same Printing Manager calls to print on all varieties of printers. The user "installs" a new printer by giving the Printing Manager a new printer resource file to work with (Figure 1). Printer installation is transparent to you application, and you needn't be concerned with it.

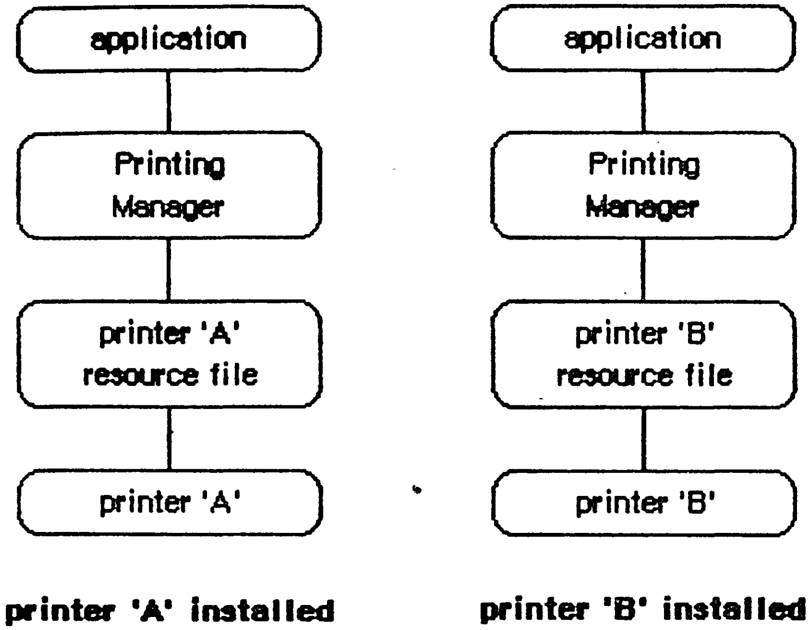


Figure 1. Printer Installation

Each printer resource file also contains a device driver that communicates between the Printing Manager and the printer. Because the actual routines of the device driver differ for each variety of printer, there exists a different device driver for each printer. The Printing Manager routines used to call a printer's device driver are the same, regardless of printer variety; this manual will refer to the device driver of the currently installed printer as the Printer Driver.

You define the image to be printed by using a printing port, a special QuickDraw grafPort customized for printing:

```
     TYPE TPPrPort = ^TPrPort;
          TPrPort  = RECORD
                       gPort:  GrafPort; {grafPort to draw in}
                       gProcs: QDProcs;  {pointers to drawing routines}
                       {more fields for internal use only}
                     END;
```

The Printing Manager gives you a printing port when you prepare to print a document. You print text and graphics by drawing into this port with QuickDraw, just as if you were drawing on the screen. The Printing Manager installs its own versions of QuickDraw's low-level drawing routines in the printing port, causing your higher-level QuickDraw calls to drive the printer instead of drawing on the screen. GProcs contains pointers to these low-level drawing routines.

(note)

> To convert a pointer to a printing port into an equivalent grafPtr for use with QuickDraw, you can use the following variant record type:

```
TYPE TPPort = PACKED RECORD
                 CASE INTEGER OF
                   Ø: (pGPort:  GrafPtr);
                   1: (pPrPort: TPPrPort)
              END;
```

METHODS OF PRINTING

The Printing Manager supports two different methods of printing documents: draft and spool. In draft printing, your QuickDraw calls are converted directly into command codes the printer understands, which are then immediately used to drive the printer. Each element of the image is printed as soon as you request it; as you move around to various coordinates within the grafPort, the print head moves to the corresponding positions on the printed page. Draft printing uses the printer's native font and graphics capabilities and probably won't produce an image matching the one on the screen. This method of printing is more direct than spool printing, but it can also be cumbersome, especially for graphics. Draft printing is most appropriate for making quick copies of text documents, which are printed straight down the page from top to bottom and left to right. Depending on the printer and what you're printing, draft printing may not even be possible; for instance, not all printers are capable of moving the paper backwards (toward the top of the page).

Spooling and spool printing are complementary halves of a two-stage process. First you cause the Printing Manager to write out (spool) a representation of your document's printed image to a disk file. This spool file is later read back in, each page is imaged (converted into an array of dots at the appropriate resolution), and the result is sent to the printer in a single pass from top to bottom. Spool printing uses QuickDraw and the Font Manager's graphics and font capabilities to produce an image closely matching the one on the screen.

(note)

> The internal format of spool files is private to the Printing Manager and may vary from one printer to another. This means that spool files destined for one printer can't necessarily be printed on another. In spool files for the Imagewriter printer, each page is stored in the form of a QuickDraw picture. It's envisioned that most other printers will use this same approach, but there may be exceptions.

Spooling and spool printing are two separate stages because spool printing a document takes a lot of space--typically from 2ØK to 4ØK for the printing code, buffers, and fonts, but spooling a document takes only about 3K. When spooling a document, large portions of your application's code and data may be needed in memory; when spool printing, most of your application's code and data are no longer needed. Normally you'll make your printing code a separate program segment, so you can swap the rest of your code and data out of memory during printing and swap it back in after you're finished.

If your application can't afford the space required by spool printing, it can just perform the spooling stage, and leave the spool file on the disk for the user to print later from the Finder (see next section). The maximum number of pages in a spool file is defined by the following constant *** it may increase *** :

```
CONST iPFMaxPgs = 128; {maximum number of pages in a spool file}
```

(note)

> Advanced programmers: In addition to draft printing and spooling, you can define as many as two more of your own methods of document printing for any given printer. (No such additional printing methods are currently defined for the Imagewriter.) There are also a number of low-level printing methods available, such as bitmap printing, text streaming, and screen printing. These methods are discussed in the section "Using a Printer Driver".

Imaging During Spool Printing

The bit image for a typical page is too big to fit in memory all at once. For instance, at the highest resolution of the Imagewriter printer (16Ø dots per inch horizontally by 144 vertically), an 8-by-1Ø 1/2-inch page image contains approximately a quarter megabyte of information, or twice the total memory capacity of the Macintosh. So instead of imaging and printing the entire page at once, the page has to be broken into bands small enough to fit in memory. During spool printing the Printing Manager actually images each band individually, adjusting the fields of the printing port to limit the actual drawing to the boundaries of the band. It then prints the resulting bit image before imaging the next band. A page can be broken into bands ("scanned") in any of four ways. Figure 2 shows the four possible scan directions of a printing port.

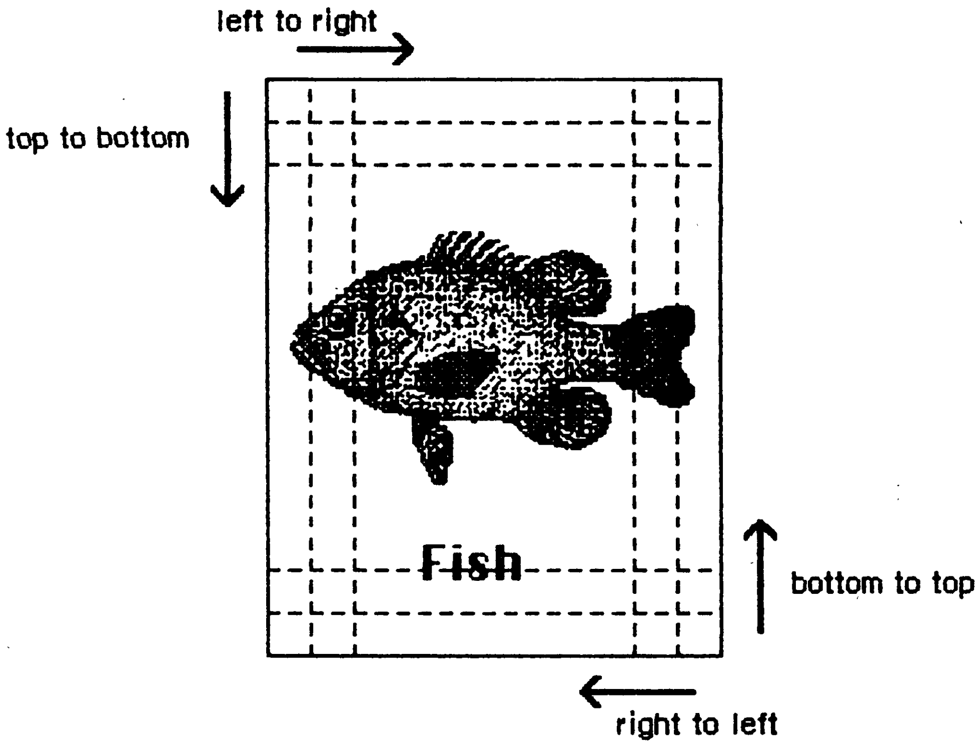


Figure 2. Scan Directions

The bands are always printed from top to bottom relative to the physical sheet of paper; the scan direction determines the correspondence between these printed bands and the dots of the image. If the long dimension of the paper runs vertically with respect to the image, the page is said to be in portrait orientation; if the long dimension runs horizontally, the page is in landscape orientation. In practice, portrait pages are normally scanned from top to bottom and landscape pages from left to right.

PRINTING FROM THE FINDER

The Macintosh user can choose to print from the Finder as well as from an application. Your application should support both alternatives.

To print a document from the Finder, the user selects the document's icon and chooses the Print command from the File menu. When the Print command is chosen, the Finder starts up the document's application, and passes information to the application indicating that the file is to be printed rather than opened. The application is then expected to print the document, preferably without doing its entire startup sequence. It may choose to do any of the following:

- Draft-print the document.
- Spool the document to a file and then print it immediately.
- Spool the document to a file and leave it for the user to print later via the Printer program (descibed below).

If your application writes spool files on a disk and then doesn't spool print them, it's up to the user to print them. The user simply selects the spool file's icon (Figure 3) and chooses the Print command from the File menu. When the Print command is chosen, the Finder starts up a special program called Printer, which spool prints spool files. It's provided as a utility for use with programs that don't do their own spool printing. Its main purpose is to read a spool file, image it, and print it.

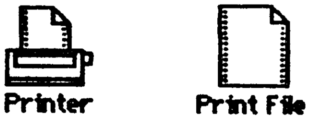


Figure 3. Icons for the Printer Program and Spool Files

Spool files can be identified by their file type and creator:

```
CONST lPfType = $5Ø46484C; {spool file type 'PFIL'}
      lPfSig  = $5Ø535953; {spool file creator 'PSYS'}
```

(note)

The details of the Finder interface are discussed in The Structure of a Macintosh Application.

*** This method of spool printing may be temporary. Currently, the easiest way for your application to do printing is to leave spool files on the disk and rely on the user to print them via Printer. Eventually Printer may be eliminated and one of the following solutions will be employed: The process will remain the same, and the code of Printer will be integrated into the Finder; or your application will be required to do spool printing itself. ***

PRINT RECORDS AND DIALOGS

For every printing operation, your application needs to determine the following:

- the resolution and other characteristics of the printer being used
- the dimensions of the printed image and of the physical sheet of paper
- the printing method to be used (draft or spool)
- the name of the spool file, if applicable
- which pages of the document to print
- how many copies to print
- an optional background procedure to be run during idle times in the printing process (discussed later)

This information is contained in a data structure called a print record. The Printing Manager fills in most of the print record for you. Some values depend on the variety of printer installed in the Printing Manager; others are set as a result of dialogs with the user.

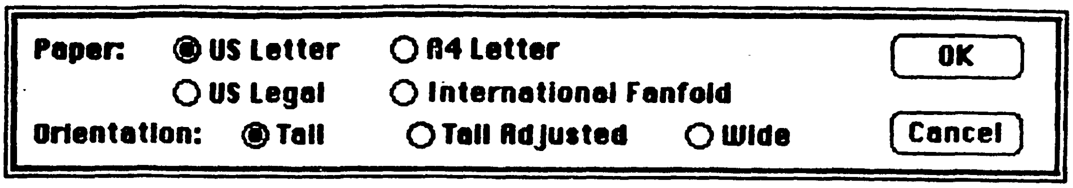
(note)

> Whenever you save a document, it's recommended that you write an appropriate print record in the document's file (see the "Printing Resources" section). This allows the document to "remember" its own printing parameters for use the next time it's printed.

(note)

> If you try to use a print record that's invalid for the current version of the Printing Manager or for the printer installed in the Printing Manager, the Printing Manager will correct the record by filling it with default values.

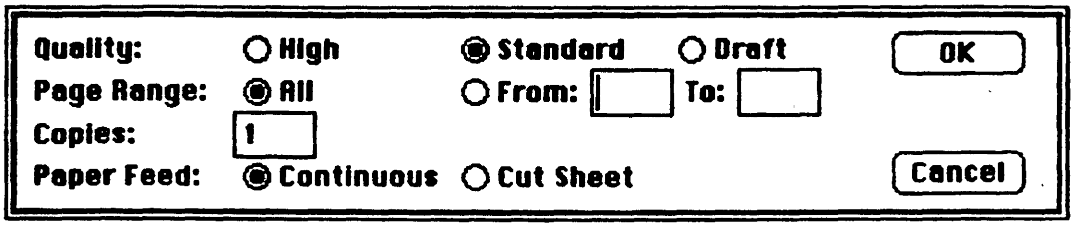
The information in the print record that can vary from one printing job to the next is obtained from the user by means of dialogs. The Printing Manager uses two standard dialogs for this purpose. The style dialog includes the paper size and page orientation (Figure 4). This dialog is conventionally associated with a Page Setup command in the application.



Paper: ● US Letter ○ A4 Letter
○ US Legal ○ International Fanfold
Orientation: ● Tall ○ Tall Adjusted ○ Wide

Figure 4. The Standard Style Dialog

The job dialog, normally associated with the application's Print command, requests information on how to print the document **this time**, such as the method of printing (draft or spool), the print quality (for printers that offer a choice of resolutions), the type of paper feed (such as fanfold or cut-sheet), the range of pages to be printed, and the number of copies (Figure 5).



Quality: ○ High ● Standard ○ Draft
Page Range: ● All ○ From: □ To: □
Copies: 1
Paper Feed: ● Continuous ○ Cut Sheet

Figure 5. The Standard Job Dialog

Print records are referred to by handles. Their structure is as follows:

```
TYPE THPrint = ^TPPrint;
     TPPrint = ^TPrint;
     TPrint  = RECORD
                  iPrVersion: INTEGER;  {Printing Manager version}
                  prInfo:     TPrInfo;  {printer information}
                  rPaper:     Rect;     {paper rectangle}
                  prStl:      TPrStl;   {style information}
                  prInfoPT:   TPrInfo;  {copy of prInfo}
                  prXInfo:    TPrXInfo; {band information}
                  prJob:      TPrJob;   {job information}
                  printX:     ARRAY [1..19] OF INTEGER
                                        {used internally}
               END;
```

IPrVersion identifies the version of the Printing Manager that initialized this print record.

Most of the other fields of the print record are "subrecords" containing various parts of the overall printing information; these are discussed in separate sections below.

Assembly-language note: The global constant iPrintSize equals the length in bytes of a print record.

The Printer Information Subrecord

The printer information subrecord (field prInfo of the print record) describes the characteristics of the particular printer you're using. Its contents are set by the Printing Manager when it initializes the print record. All applications will need to refer to the information it contains. (The prInfoPT field of the print record is a copy of the prInfo field and is used internally by the Printing Manager during printing.)

The printer information subrecord is defined as follows:

```
TYPE TPrInfo = RECORD
                 iDev:  INTEGER; {driver information}
                 iVRes: INTEGER; {printer vertical resolution}
                 iHRes: INTEGER; {printer horizontal resolution}
                 rPage: Rect     {page rectangle}
               END;
```

The iDev field contains information used by QuickDraw and the Font Manager for selecting fonts for the printer. The high-order byte is the reference number of the Printer Driver, -3. The low-order byte contains device-specific information on how the printer is being used. For example, for the Imagewriter printer, bit Ø specifies high (1) or low (Ø) resolution and bit 1 specifies portrait (1) or landscape (Ø) orientation.

(note)

If you store this word into the device field of a grafPort, you can use the QuickDraw routines CharWidth, StringWidth, TextWidth, and GetFontInfo to ask for information about a font drawn on that device.

IVRes and iHRes give the vertical and horizontal resolution of the printer, in dots per inch.

RPage is the page rectangle, representing the boundaries of the printable page. Its top left corner always has coordinates (Ø,Ø); the coordinates of the bottom right corner give the maximum page height and width attainable on the given printer, in dots. Typically these are slightly less than the physical dimensions of the paper, because of the printer's mechanical limitations.

The results of the style dialog conducted with the user determine the values of the iVRes, iHRes, and rPage fields. For example, with the

Imagewriter printer, the style dialog's three orientation buttons yield the following:

Button	Orientation	IVRes	IHRes
Tall	Portrait	8Ø	72
Tall adjusted	Portrait	72	72
Wide	Landscape	72	72

The physical paper size is given by the rPaper field of the print record. This paper rectangle is outside of the page rectangle: it defines the physical boundaries of the paper in the same coordinate system as rPage (see Figure 6). Thus the top left coordinates of the paper rectangle are typically negative and its bottom right coordinates are greater than those of the page rectangle.

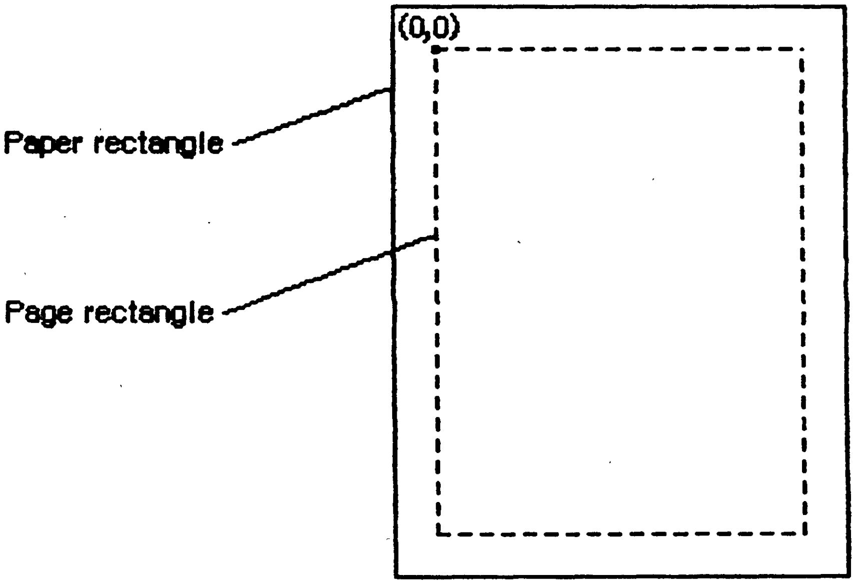


Figure 6. Page and Paper Rectangles

The Style Subrecord

The style subrecord (field prStl of the print record) describes the type and size of paper used in the printer. The contents of the style subrecord are normally set by the Printing Manager after dialogs with the user, and only advanced programmers need be concerned with them.

The style subrecord is defined as follows:

```
      TYPE TPrStl = RECORD
                       wDev:    TWord;      {used internally}
                       iPageV:  INTEGER;    {paper height}
                       iPageH:  INTEGER;    {paper width}
                       bPort:   SignedByte; {printer or modem port}
                       feed:    TFeed       {paper type}
                    END;
```

IPageV and iPageH give the physical dimensions of the paper, in 120ths of an inch. The user can set them by choosing a standard paper size (such as U.S. Letter, U.S. Legal, or European A4) from the style dialog. The number of units per inch is defined by the following constant:

```
      CONST iPrPgFract = 120; {units per inch of paper dimension}
```

BPort designates which port on the back of the Macintosh the printer is connected to: 0 for the printer port, 1 for the modem port. *** Currently the Printing Manager ignores this value, and instead uses the global variable sPPrint. ***

Feed identifies the type of paper feed being used:

```
      TYPE TFeed = (feedCut,      {hand-fed, individually cut sheets}
                    feedFanfold,  {continuous-feed fanfold paper}
                    feedMechCut,  {mechanically fed cut sheets}
                    feedOther);   {other types of paper}
```

The user sets this field by choosing Continuous or Cut Sheet from the job dialog. When Cut Sheet is chosen, the printer will pause at the end of each page and a dialog box will prompt the user to insert the next sheet.

The Job Subrecord

The job subrecord (field prJob of the print record) contains information about a particular printing job. Its contents are normally set by the Printing Manager as a result of a job dialog with the user.

The job subrecord is defined as follows:

```
TYPE TPrJob = RECORD
                iFstPage:  INTEGER;    {first page to print}
                iLstPage:  INTEGER;    {last page to print}
                iCopies:   INTEGER;    {number of copies}
                bJDocLoop: SignedByte; {printing method}
                fFromUsr:  BOOLEAN;    {TRUE if called from application}
                pIdleProc: ProcPtr;    {background procedure}
                pFileName: TPStr80;    {spool file name}
                iFileVol:  INTEGER;    {volume reference number}
                bFileVers: SignedByte; {version number of spool file}
                bJobX:     SignedByte  {not used}
              END;
```

```
     TPStr8Ø = ^TStr8Ø;
     TStr8Ø  = STRING[8Ø];
```

Most programmers need only be concerned with the bJDocLoop, pFileName, and pIdleProc fields. BJDocLoop represents the method of printing to use. The user sets this field by choosing High, Standard, or Draft from the job dialog. BJDocLoop should be one of the following predefined constants:

```
       CONST bDraftLoop = Ø; {draft printing}
             bSpoolLoop = 1; {spooling}
             bUser1Loop = 2; {printer-specific, method 1}
             bUser2Loop = 3; {printer-specific, method 2}
```

If you're spool printing, it's a good idea to give each file you spool to the disk a different name, in the pFileName field, so that it doesn't overwrite any other spool files on the disk. PFileName is initialized to NIL, denoting the default file name found in the printer resource file. *** (Currently the default file name is 'Print File'.) ***

IFstPage and iLstPage designate the first and last pages to be printed. The Printing Manager knows nothing about any page numbering placed by an application within a document, and always considers the first printable page to be page 1. For example, if iFstPage is 2, the Printing Manager will print the second page in the document, regardless of how the page is actually numbered. If you're draft printing, you'll need to use the value of iCopies to determine the number of copies to print (the Printing Manager automatically handles multiple copies for spooling).

FFromUsr is TRUE when the Printing Manager is called from an application program, FALSE when it's called from the Printer program. PIdleProc is a pointer to the background procedure (explained below) for this printing operation. In a newly initialized print record this field is set to NIL, designating the default background procedure. This procedure just polls the keyboard and cancels further printing if the user types Command-period. You can install a background procedure of your own by storing directly into the pIdleProc field.

For spooling operations, iFileVol and bFileVers are the volume reference number and version number of the spool file. IFileVol and bFileVers are both initialized to Ø. You can override the default settings by storing directly into these fields.

The Band Information Subrecord

The band information subrecord (field prXInfo of the print record) contains information about the way a page will be imaged during spool printing. Its contents are set by the Printing Manager, and most programmers needn't be concerned with it.

The band information subrecord is defined as follows:

```
TYPE TPrXInfo = RECORD
                     iRowBytes: INTEGER;     {bytes per row}
                     iBandV:    INTEGER;     {vertical dots}
                     iBandH:    INTEGER;     {horizontal dots}
                     iDevBytes: INTEGER;     {size of bit image}
                     iBands:    INTEGER;     {bands per page}
                     bPatScale: SignedByte;  {used by QuickDraw}
                     bUlThick:  SignedByte;  {underline thickness}
                     bUlOffset: SignedByte;  {underline offset}
                     bUlShadow: SignedByte;  {underline descender}
                     scan:      TScan;       {scan direction}
                     bXInfoX:   SignedByte   {not used}
                   END;
```

IRowBytes is the number of bytes in each row of the band's bit image, iBandV and iVBandH are the dimensions of the band in dots, iDevBytes is the number of bytes of memory needed to hold the bit image, and iBands is the number of bands per page.

BPatScale is used by QuickDraw when it scales patterns to the resolution of the printer. BUlThick, bUlOffset, and bUlShadow are used for underlining text; they stand for the thickness of the underline, its offset below the base line, and the width of the break around descenders, all in dots. The scan field specifies the scan direction for banding as a value of type TScan:

```
TYPE TScan = (scanTB,  {scan top to bottom}
              scanBT,  {scan bottom to top}
              scanLR,  {scan bottom to top}
              scanRL); {scan right to left}
```

BACKGROUND PROCESSING

As mentioned above, the job subrecord includes a pointer, pIdleProc, to an optional background procedure to be run whenever the Printing Manager has directed output to the printer and is waiting for the printer to finish. The background procedure takes no parameters and returns no result; the Printing Manager simply runs it at every opportunity. There's no limit to the length of time that a background procedure can execute, but beyond a certain length of time printing will be slowed.

If you don't designate a background procedure, the Printing Manager will use one by default that just polls the keyboard and cancels further printing if the user types Command-period. In this case you should display an alert box to inform the user that the Command-period option is available. It's suggested, however, that instead of relying on this method, you supply your own background procedure to give the user a more convenient way to cancel printing. For instance, you might put up a dialog box with a Cancel button the user can click with the mouse; or, in a background procedure that runs your application, you might replace the Print command with Stop Print.

While printing from a spool file, the Printing Manager maintains a printer status record in which it reports on the progress of the printing operation:

```
TYPE TPrStatus = RECORD
                   iTotPages:  INTEGER;  {total number of pages}
                   iCurPage:   INTEGER;  {page being printed}
                   iTotCopies: INTEGER;  {number of copies}
                   iCurCopy:   INTEGER;  {copy being printed}
                   iTotBands:  INTEGER;  {bands per page}
                   iCurBand:   INTEGER;  {band being printed}
                   fPgDirty:   BOOLEAN;  {TRUE if started printing page}
                   fImaging:   BOOLEAN;  {TRUE if imaging}
                   hPrint:     THPrint;  {print record}
                   pPrPort:    TPPrPort; {printing port}
                   hPic:       PicHandle {used internally}
                 END;
```

FPgDirty is TRUE if anything has been printed yet on the current page, FALSE if not; fImaging is TRUE while a band is being imaged, FALSE while it's being printed. HPrint is a handle to the print record for this printing operation; pPrPort is a pointer to the printing port.

Your background procedure can use this information--for example, to display a progress report on the screen ("Now printing copy 3 of 5, page 7 of 12").

(note)

> The Printing Manager only calls your background procedure while it's printing. If you want your background procedure to execute during spooling, you'll have to call it yourself.

Advanced programmers can use background processing in a variety of useful ways. For example, with a background procedure that performs one pass through your main program loop, you can achieve the effect of concurrent printing. That is, your application can continue to run while the printing is taking place, although there may be some degradation in performance. The user is given the illusion that the printing is going on "in the background" behind the application. (In reality, of course, it's the application that's running in the background behind the printing task.)

(warning)

> You have to be careful in the way you write your background procedure, to avoid a number of subtle concurrency problems that may arise. For instance, if the background procedure uses QuickDraw, it must be sure to restore the printing port as the current port before returning. It's particularly important not to attempt any printing from within the background procedure: the Printing Manager is **not** reentrant! If you use a background procedure that runs your application concurrently with printing, it should disable all menu items having to do with printing, such as Page Setup and Print.

USING THE PRINTING MANAGER

This section discusses how the Printing 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.

To use the Printing Manager, you must have previously initialized QuickDraw, the Font Manager, the Window Manager, the Menu Manager, TextEdit, and the Dialog Manager. The first Printing Manager routine to call is PrOpen, which opens the printer resource file. The last routine to call is PrClose, which closes the Printer Driver and the printer resource file.

(note)

> PrOpen and PrClose are meant to be called once each, at the beginning and end of your application. However, if space is particularly critical, you may prefer to bracket every Printing Manager call with a PrOpen and a PrClose. This frees the space occupied by various Printing Manager data structures when they're not in use.

Before printing a document, you need a properly filled out print record. You can either use an existing print record (for instance, from a document) or initialize one to the current default settings by calling PrintDefault. If you use an existing print record, you should call PrValidate to make sure it's valid for the current version of the Printing Manager and for the currently installed printer.

When the user chooses the Page Setup commmand, call PrStlDialog to ask about the paper size and page orientation. From the printer information subrecord you can then determine where each page break occurs.

When the user chooses the Print commmand, call PrJobDialog to ask the user for specific information about that printing job. To apply the results of one job dialog to several documents (when printing from the Finder, for example), call PrJobMerge.

To draft print or spool a document, begin by calling PrOpenDoc, which returns a printing port customized for draft printing or spooling (depending on the bJDocLoop field of the job subrecord). You can then print or spool your document by "drawing" into this printing port with QuickDraw, using the values in the printer information subrecord to adjust for the parameters of the printer. Call PrOpenPage and PrClosePage at the beginning and end of each page, and PrCloseDoc at the end of the entire document. Each page is either printed immediately (draft printing) or written to the disk as part of a spool file (spooling).

To print a spool file, swap as much of your program out of memory as you can, and then call PrPicFile.

Call PrError to check for errors caused by a Printing Manager routine. To cancel a printing operation in progress, use PrSetError. Be sure to call PrCloseDoc or PrClosePage after you cancel printing in progress.

PRINTING MANAGER ROUTINES

This section describes the procedures and functions that make up the Printing Manager. They're presented in their Pascal form; for information on using them from assembly language, see Programming Macintosh Applications in Assembly Language.

Initialization and Termination

PROCEDURE PrOpen;

PrOpen prepares the Printing Manager for use. It opens the Printer Driver and the printer resource file. If either of these items is missing, or if the printer resource file is not properly formed, PrOpen will do nothing, and PrError will return a Resource Manager result code.

PROCEDURE PrClose;

PrClose releases the memory used by the Printing Manager. It closes the printer resource file, allowing the file's resource map to be removed from memory. It *** currently *** doesn't close the Printer Driver, however, since the driver may have been opened before the PrOpen call was issued.

Print Records and Dialogs

PROCEDURE PrintDefault (hPrint: THPrint);

PrintDefault fills the fields of a print record with the current default values stored in the printer resource file. HPrint is a handle to the record, which may be a new print record that you've just allocated or an existing one (from a document, for example).

FUNCTION PrValidate (hPrint: THPrint) : BOOLEAN;

PrValidate checks the contents of a print record for compatibility with the current version of the Printing Manager and with the installed printer. If the record is valid, the function returns FALSE (no change); if invalid, the record is adjusted to the current default values, taken from the printer resource file, and the function returns TRUE.

PrValidate also updates the print record to reflect the current settings in the style and job subrecords. These changes have no effect on the function's Boolean result.

FUNCTION PrStlDialog (hPrint: THPrint) : BOOLEAN;

PrStlDialog conducts a style dialog with the user to determine the paper size and paper orientation being used. The initial settings displayed in the dialog box are taken from the current values in the print record. If the user confirms the dialog, the results of the dialog are saved in the print record and the function returns TRUE; otherwise the print record is left unchanged and the function returns FALSE.

(note)

> If the print record was taken from a document, you should update its contents in the document's file if PrStlDialog returns TRUE. This makes the results of the style dialog "stick" to the document.

FUNCTION PrJobDialog (hPrint: THPrint) : BOOLEAN;

PrJobDialog conducts a job dialog with the user to determine the printing quality, number of pages to print, and so on. The initial settings displayed in the dialog box are taken from the current values in the print record. If the user confirms the dialog, both the print record and the printer resource file are updated (so that the user's choices "stick" to the printer) and the function returns TRUE; otherwise the print record and printer resource file are left unchanged and the function returns FALSE.

(note)

> If the job dialog is associated with your application's Print command, you should proceed with the requested printing operation if PrJobDialog returns TRUE. If the print record was taken from a document, you should update its contents in the document's file.

PROCEDURE PrJobMerge (hPrintSrc,hPrintDst: THPrint);

PrJobMerge copies the job subrecord from one print record (hPrintSrc) to another (hPrintDst) and updates the destination record's printer information, band information, and paper rectangle, based on information in the job subrecord. This allows the information in the job subrecord to be used for a group of related jobs.

Draft Printing and Spooling

```
FUNCTION PrOpenDoc (hPrint: THPrint; pPrPort: TPPrPort; pIOBuf: Ptr)
           : TPPrPort;
```

PrOpenDoc initializes a printing port for use in printing a document, makes it the current port, and returns a pointer to it. HPrint is a handle to the print record for this printing operation. The printing port is customized for draft printing or spooling, depending on the setting of the bJDocLoop field in the job subrecord. For spooling, the spool file's name, volume reference number, and version number are taken from the job subrecord.

PPrPort is a pointer to the storage to be used for the printing port. If this parameter is NIL, PrOpenDoc will allocate a new printing port for you. Similarly, pIOBuf points to an area of memory to be used as an input/output buffer; if it's NIL, PrOpenDoc will use the volume buffer for the spool file's volume.

(note)

> The pPrPort and pIOBuf parameters are provided because both the printing port and the input/output buffer are nonrelocatable objects. To avoid cluttering the heap with such objects, you have the opportunity to allocate them yourself and pass them to PrOpenDoc. Most of the time you'll just set both of these parameters to NIL.

(note)

> Newly created printing ports use the system font (since they're grafPorts), but newly created windows use the application font. Be sure the font you use in the printing port is the same as the font in your application window if you want the text in both places to match.

PROCEDURE PrOpenPage (pPrPort: TPPrPort; pPageFrame: TPRect);

PrOpenPage begins a new page in the document associated with the given printing port. The page is printed only if it falls within the page range designated in the job subrecord.

For spooling, the pPageFrame parameter points to a rectangle that will be used as the QuickDraw picture frame for this page:

```
       TYPE TPRect = ^Rect;
```

When the spool file is later printed, this rectangle will be scaled (via the QuickDraw DrawPicture procedure) to coincide with the page rectangle in the printer information subrecord. Unless you want the printout to be scaled, you should set pPageFrame to NIL--this uses the current page rectangle as the picture frame, and the page will be printed with no scaling.

PROCEDURE PrClosePage (pPrPort: TPPrPort);

PrClosePage finishes up the current page of the document associated with the given printing port. For draft printing, it ejects the page from the printer and, if necessary, alerts the user to insert another; for spooling, it closes the picture representing the current page.

PROCEDURE PrCloseDoc (pPrPort: TPPrPort);

PrCloseDoc finishes up the printing of the document associated with the given printing port. For draft printing, it issues a form feed and a reset command to the printer; for spooling, it closes the file if the spooling was successfully completed or deletes it the file if the spooling was unsuccessful.

Spool Printing

PROCEDURE PrPicFile (hPrint: THPrint; pPrPort: TPPrPort; pIOBuf: Ptr;
            pDevBuf: Ptr; VAR prStatus: TPrStatus);

PrPicFile images and prints a spool file. HPrint is a handle to the print record for this printing operation. The name, volume reference number, and version number of the spool file will be taken from the job subrecord of this print record. After printing is successfully completed, the Printing Manager deletes the spool file from the disk.

PPrPort is a pointer to the storage to be used for the printing port for this operation. If this parameter is NIL, PrPicFile will allocate its own printing port. Similarly, pIOBuf points to an area of memory to be used as an input/output buffer for reading the spool file; if it's NIL, PrPicFile will use the volume buffer for the spool file's volume. PDevBuf points to a similar buffer (the "band buffer") for holding the bit image to be printed; if NIL, PrPicFile will allocate its own buffer from the heap. As for PrOpenDoc, you'll normally want to set all of these storage parameters to NIL.

(note)

> If you provide your own storage for pDevBuf, it has to be big enough to hold the number of bytes indicated by the iDevBytes field of the TPrXInfo subrecord of the print record.

(warning)

> Be sure not to pass, in pPrPort, a pointer to the same printing port you received from PrOpenDoc, the one you originally used to spool the file. If that earlier port was allocated by PrOpenDoc itself (that is, if the pPrPort parameter to PrOpenDoc was NIL), then PrCloseDoc will have disposed of the port, making your pointer to it invalid. PrPicFile initializes a fresh printing port of its own; you just provide the storage (or let PrPicFile allocate it for itself). Of course, if you earlier provided your own storage to PrOpenDoc, there's no reason you can't use the same storage again for PrPicFile.

The prStatus parameter is a printer status record that PrPicFile will use to report on its progress. Your background procedure (if any) can use this record to monitor the state of the printing operation.

Handling Errors

FUNCTION PrError : INTEGER; [Pascal only]

PrError returns the result code returned by the last Printing Manager routine. The possible result codes are:

```
CONST noErr       = Ø;    {no error}
      iMemFullErr = -1Ø8; {not enough heap space}
```

and any Resource Manager result code. A result code of iMemFullErr means that the Memory Manager was unable to fulfill a memory allocation request by the Printing Manager.

PROCEDURE PrSetError (iErr: INTEGER); [Pascal only]

PrSetError stores the specified value into the global variable where the Printing Manager keeps its result code. The main *** (currently the only) *** use of this procedure is for canceling a printing operation in progress. To do this, write

```
       PrSetError(iPrAbort)
```

where iPrAbort is the following predefined constant:

```
       CONST iPrAbort = 128; {result code for halting printing}
```

Assembly-language note: You can achieve the same effect as PrSetError by storing directly into the location specified by printVars+iPrErr. *** Currently you shouldn't store into this location if it already contains an nonzero value. ***

Low-Level Driver Access

The routines in this section are used for communicating directly with the Printer Driver; the Printer Driver itself is described in the next section. You'll need to be familiar with the Device Manager to use the information given in this section.

```
PROCEDURE PrDrvrOpen;
```

PrDrvrOpen opens the Printer Driver.

```
PROCEDURE PrDrvrClose;
```

PrDrvrClose closes the Printer Driver.

```
PROCEDURE PrCtlCall (iWhichCtl: INTEGER; lParam1,lParam2,lParam3:
            LongInt);
```

PrCtlCall calls the Printer Driver's control routine. IWhichCtl designates the operation to be performed; the rest of the parameters depend on the operation.

```
FUNCTION PrDrvrDCE : Handle;
```

PrDrvrDCE returns a handle to the Printer Driver's device control entry.

```
FUNCTION PrDrvrVers : INTEGER;
```

PrDrvrVers returns the version number of the Printer Driver in the system resource file.

The version number of the Printing Manager is available as the predefined constant iPrRelease. You may want to compare the result of PrDrvrVers with iPrRelease to see if the Printer Driver in the resource file is the most recent version.

PROCEDURE PrNoPurge;

PrNoPurge prevents the Printer Driver from being purged from the heap.

PROCEDURE PrPurge;

PrPurge allows the Printer Driver to be purged from the heap.

THE PRINTER DRIVER

This section describes the Printer Driver, the device driver that communicates with a printer via the printer port or the modem port. Only programmers interested in low-level printing or writing their own device driver need read this. You'll need to be familiar with the Device Manager manual to use most of this information and the low-level routines described above.

The printer resource file for each variety of printer includes a device driver for that printer. When a particular printer is installed in the Printing Manager, the printer's device driver is copied from the printer resource file into the system resource file, making it the active Printer Driver.

The Printer Driver responds to the standard Device Manager calls OpenDriver, CloseDriver, Control, and Status. You can also communicate with it via the Printing Manager routines PrDrvrOpen, PrDrvrClose, and PrCtlCall. (The Status call is normally used only by the Font Manager.) Its driver name and driver reference number are available as the following predefined constants:

```
       CONST sPrDrvr    = '.Print'; {Printer Driver resource name}
             iPrDrvrRef = -3;       {Printer Driver reference number}
```

To open the Printer Driver, call PrDrvrOpen; it'll remain open until you call PrDrvrClose. Calling PrNoPurge will prevent the driver from being purged from the heap until you call PrPurge.

You can call the PrDrvrVers function to determine whether the printing resources stored in the system resource file are compatible with the version of the Printing Manager you're using.

To get a handle to the driver's device control entry, call PrDrvrDCE. By calling the driver's control routine with PrCtlCall, you can perform a number of low-level printing operations such as bitmap printing, screen printing, and direct streaming of text to the printer (described below). The first parameter to PrCtlCall, iWhichCtl, identifies the operation you want. The following values are predefined:

```
CONST iPrBitsCtl = 4; {bitMap printing}
      iPrIOCtl   = 5; {text streaming}
      iPrEvtCtl  = 6; {screen printing}
      iPrDevCtl  = 7; {device control}
      iFMgrCtl   = 8; {used by the Font Manager}
```

The remaining parameters of PrCtlCall--lParam1, lParam2, and lParam3--are three long integers whose meaning depends on the operation, as described below.

BitMap Printing

To send all or part of a bitMap directly to the printer, use PrCtlCall with iWhichCtl = iPrBitsCtl. Parameter lParam1 is a pointer to a QuickDraw bitMap; lParam2 is a pointer to the rectangle to be printed, in the coordinates of the printing port.

LParam3 is a printer-dependent parameter. On the Imagewriter it's used to control the printer's aspect ratio (the ratio of horizontal to vertical resolution). In low resolution, the Imagewriter normally prints 8Ø dots per inch horizontally by 72 vertically. This produces rectangular dots that are taller than they are wide. Since the Macintosh screen has square pixels (72 per inch both horizontally and vertically), images printed on the Imagewriter don't look exactly the same as they do on the screen.

To address this problem, the Imagewriter has a special square-dot mode that alters the speed of the print head to produce 72 dots per inch horizontally instead of 8Ø. Printing in this mode is slower than in the normal mode, but gives a more faithful reproduction of what the user sees on the screen. The user can choose which of the two modes to use by using the Printer program.

The value of the lParam3 parameter should be one of the following predefined constants:

```
CONST lScreenBits = Ø; {configurable}
      lPaintBits  = 1; {72 by 72 dots}
```

LScreenBits tells the Printer Driver to honor the user's selection between rectangular and square dots; lPaintBits overrides the user's choice and forces square dots.

Putting all this together, you can print the entire screen at the user's chosen aspect ratio with

```
PrCtlCall(iPrBitsCtl, ORD(@screenBits),
            ORD(@screenBits.bounds), lScreenBits)
```

To print the contents of a single window in square dots, use

```
      PrCtlCall(iPrBitsCtl, ORD(@theWindow^.portBits),
                ORD(@theWindow^.portRect), lPaintBits)
```

Text Streaming

Text streaming is useful for fast printing of text when speed is more important than fancy formatting or visual fidelity. It gives you full access to the printer's native text facilities, such as control or escape sequences for boldface, italic, underlining, or condensed or expanded type, but makes no use of QuickDraw's elaborate formatting capabilities.

(warning)

> Relying on specific printer capabilities and control sequences will make your application printer-dependent.

You can send a stream of text characters directly to the printer with iWhichCtl = iPrIOCtl. LParam1 is a pointer to the beginning of the text; lParam2 is the number of bytes to transfer (a long integer); lParam3 is a pointer to an optional background procedure, or NIL for none.

IPrDevCtl is used for various printer control operations. When streaming text to the printer, you can use iPrDevCtl to perform these general operations in a printer-independent way, letting the Printer Driver take care of the details for a specific printer. The lParam1 parameter specifies the operation you want:

```
      CONST lPrReset    = $ØØØ1ØØØØ;  {reset printer}
            lPrPageEnd  = $ØØØ2ØØØØ;  {start new page}
            lPrLineFeed = $ØØØ3ØØØØ;  {start new line}
```

Before starting to print a document with text streaming, use

```
      PrCtlCall(iPrDevCtl, lPrReset, Ø, Ø)
```

to reset the printer to its standard initial state. The parameters lParam2 and lParam3 are meaningless and should be set to Ø.

At the end of each printed line,

```
      PrCtlCall(iPrDevCtl, lPrLineFeed, Ø, Ø)
```

advances the paper one line and returns to the left margin. This achieves the effect of the standard "CRLF" (carriage-return-line-feed) sequence in a printer-independent way. It's strongly recommended that you use this method instead of sending carriage returns and line feeds directly to the printer. The lParam2 parameter tells how far to advance the paper; lParam3 is meaningless and should be set to Ø.
*** The exact use of lParam2 in this call hasn't yet been determined.

A value of Ø will probably denote the printer's standard line height, which is usually what you'll want. ***

At the end of each page,

```
PrCtlCall(iPrDevCtl, lPrPageEnd, Ø, Ø)
```

does whatever is appropriate for the given printer, such as sending a form feed character and advancing past the paper fold. It's recommended that you use this call instead of just sending a form feed yourself. LParam2 and lParam3 are meaningless and should be set to Ø.

Screen Printing

IPrEvtCtl does an immediate dump of all or part of the screen directly to the printer. LParaml is one of the following codes:

```
CONST iPrEvtAll = $ØØØ2FFFD;     {print whole screen}
      iPrEvtTop = $ØØØ1FFFD;     {print top (frontmost) window}
```

The other two parameters are meaningless and should be set to Ø. So, for example,

```
PrCtlCall(iPrEvtCtl, iPrEvtAll, Ø, Ø)
```

prints the entire screen at the user's chosen aspect ratio, and

```
PrCtlCall(iPrEvtCtl, iPrEvtTop, Ø, Ø)
```

prints just the frontmost window.

The Operating System Event Manager uses this call to do immediate screen printing when the user types a special key combination (Command-$ for the frontmost window, the same with Caps Lock for the full screen).

Font Manager Support

The Printer Driver provides one Status and one Control call for use by the Font Manager in selecting fonts for a given printer. Both are identified by the following csCode value

```
CONST iFMgrCtl = 8;
```

With the Status call, the Font Manager asks for the printer's font characterization table. After using the information in this table to select a font, it issues the Control call to give the Printer Driver a chance to modify the choice. This process is described further in the Font Manager manual.

PRINTING RESOURCES

For programmers who want to write their own device drivers for different printers or modify existing drivers, this section describes the two files that contain the resources needed to run the Printing Manager: the system resource file and the printer resource file (see Figure 7). Most of the data described in this section is accessible only to assembly-language programmers.

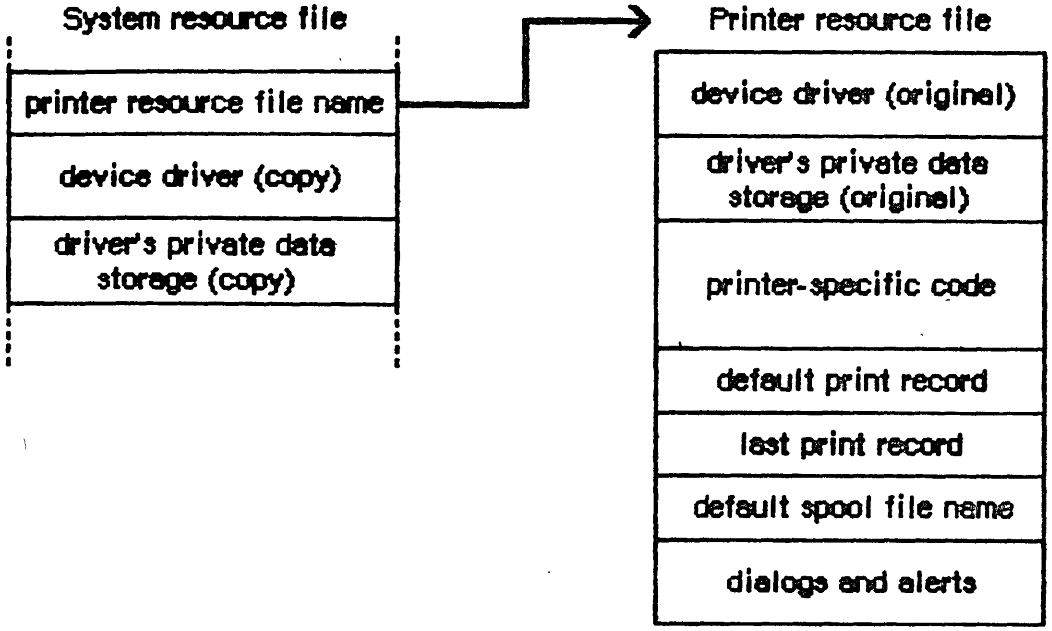


Figure 7. Printing Resources

The system resource file contains:

Resource	Resource type	Resource ID
Name of the current printer resource file	'STR '	$EØØØ
A copy of the device driver for the currently installed printer	'DRVR'	2
A copy of the driver's private data storage	'PREC'	2

The printer resource file contains the following information:

Resource	Resource type	Resource ID
The device driver for this printer	'DRVR'	$EØØØ
The driver's private storage	'PREC'	$EØØØ
Printer-specific code used to implement Printing Manager routines	'PDEF'	Ø through 6 (see below)
Default print record for use with this printer	'PREC'	Ø
Print record from the previous printing operation	'PREC'	1
Default spool file name	'STR '	$EØØ1
Style dialog	'DLOG'	$EØØØ
Job dialog	'DLOG'	$EØØ1
Installation dialog	'DLOG'	$EØØ2
Alerts	'ALRT'	(private)
Dialog and alert item lists	'DITL'	(private)

Notice that the Printer Driver and its private storage are kept in both the system and printer resource files. The copies in the system resource file are the ones actually used; those in the printer resource file are there just to be copied into the system resource file when a new printer is installed. Installing a new printer is done by copying the driver and its private storage from the printer resource file to the system resource file and placing the name of the printer resource file in the system resource file. (You can use this method to install a printer yourself, but normally it's done by the Printer program at the user's request.)

You can use the following predefined constants to identify the various resource types and IDs in the printer resource file (they'll be different in the system resource file):

```
CONST lPrintType = $50524543; {type ('PREC') for print records and }
                              { private storage}
      iPrintDef  = 0;         {ID for default print record}
      iPrintLst  = 1;         {ID for previous print record}
      iPrintDrvr = 2;         {ID for Printer Driver and its private }
                              { storage in system resource }
                              { file}
      iMyPrDrvr = $E000;      {ID for Printer Driver and its private }
                              { storage }

      iPStrRFil  = $E000;     {ID for printer resource file name}
      iPStrPFil  = $E001;     {ID for default spool file name}

      iPrStlDlg  = $E000;     {ID for style dialog}
      iPrJobDlg  = $E001;     {ID for job dialog}
```

The most important items in a printer resource file are the Printer Driver and the printer-specific code. The driver has the standard structure for device drivers, as described in the Device Manager manual, and implements the Control and Status calls as discussed above under "The Printer Driver".

The printer-specific code is kept in a series of separate overlays. They are all of resource type 'PDEF', and their resource IDs are available to assembly-language programmers as the following predefined constants:

```
iPrDraftID    .EQU   0 ;draft printing
iPrSpoolID    .EQU   1 ;spooling
iPrUser1ID    .EQU   2 ;printer-specific printing, method 1
iPrUser2ID    .EQU   3 ;printer-specific printing, method 2
iPrDlgsID     .EQU   4 ;print records and dialogs
iPrPicID      .EQU   5 ;spool printing
```

Overlays 0 and 1 do draft printing and spooling, respectively; overlays 2 and 3, if present, provide additional printing methods for a particular printer. All four overlays include the same routines, but implement them in different ways for the different printing methods. When one of the routines is called, the Printing Manager uses the bJDocLoop field in the job subrecord to decide which overlay to use. Each overlay begins with a list of offsets to the locations of the routines within that overlay.

```
lOpenDoc       .EQU       $000C0000       ;PrOpenDoc
lCloseDoc      .EQU       $00048004       ;PrCloseDoc
lOpenPage      .EQU       $00080008       ;PrOpenPage
lClosePage     .EQU       $0004000C       ;PrClosePage
```

This list is followed by the code of the routines themselves.

Overlay 4 contains the Printing Manager's routines for manipulating print records and dialogs:

```
        lDefault        .EQU        $ØØØ48ØØØ       ;PrintDefault
        lStlDialog      .EQU        $ØØØ48ØØ4       ;PrStlDialog
        lJobDialog      .EQU        $ØØØ48ØØ8       ;PrJobDialog
        lStlInit        .EQU        $ØØØ4ØØØC       ;PrStlInit
        lJobInit        .EQU        $ØØØ4ØØ1Ø       ;PrJobInit
        lDlgMain        .EQU        $ØØØ48Ø14       ;PrDlgMain
        lValidate       .EQU        $ØØØ48Ø18       ;PrValidate
        lJobMerge       .EQU        $ØØØ88Ø1C       ;PrJobMerge
```

*** PrStlInit, PrJobInit, and PrDlgMain are used in customizing the dialogs, and will be covered in a later draft of this manual. ***

Overlays 5 contains just the spool-printing routine PrPicFile (it's still preceded by an offset, however):

```
        lPrPicFile      .EQU        $ØØ148ØØØ       ;PrPicFile
```

SUMMARY OF THE PRINTING MANAGER

Constants

```
CONST { Result codes }

      iMemFullErr = -108; {not enough heap space}
      noErr       =    0; {no error}

      { Printing methods }

      bDraftLoop = 0;    {draft printing}
      bSpoolLoop = 1;    {spooling}
      bUser1Loop = 2;    {printer-specific, method 1}
      bUser2Loop = 3;    {printer-specific, method 2}

      { Printer Driver Control call parameters }

      iPrBitsCtl  = 4;          {bitMap printing}
      lScreenBits = 0;            {configurable}
      lPaintBits  = 1;            {72 by 72 dots}
      iPrIOCtl    = 5;          {text streaming}
      iPrEvtCtl   = 6;          {screen printing}
      iPrEvtAll   = $0002FFFD;    {print whole screen}
      iPrEvtTop   = $0001FFFD;    {print top (frontmost) window}
      iPrDevCtl   = 7;          {device control}
      lPrReset    = $00010000;    {reset printer}
      lPrPageEnd  = $00020000;    {start new page}
      lPrLineFeed = $00030000;    {start new line}
      iFMgrCtl    = 8;          {used by the Font Manager}

      { Miscellaneous }

      iPFMaxPgs  = 128;        {maximum number of pages in a spool file}
      iPrPgFract = 120;        {units per inch of paper dimension}
      iPrAbort   = 128;        {result code for halting printing}
      iPrRelease = 2;          {current version number of Printing }
                               { Manager}
      lPfType    = $5046484C; {spool file type 'PFIL'}
      lPfSig     = $50535953; {spool file creator 'PSYS'}

      { Printing resources }

      sPrDrvr    = '.Print';  {Printer Driver resource name}
      iPrDrvrRef = -3;        {Printer Driver reference number}
      lPrintType = $50524543; {type ('PREC') for print records }
                              { and private storage}
      iPrintDef  = 0;         {ID for default print record}
      iPrintLst  = 1;         {ID for previous print record}
      iPrintDrvr = 2;         {ID for Printer Driver and its }
                              { private storage in system }
```

```
                               { resource file}
     iMyPrDrvr = $EØØØ;        {ID for Printer Driver and its }
                               { private storage in printer }
                               { resource file}
     iPStrRFil  = $EØØØ;       {ID for printer resource file name}
     iPStrPFil  = $EØØ1;       {ID for default spool file name}
     iPrStlDlg  = $EØØØ;       {ID for style dialog}
     iPrJobDlg  = $EØØ1;       {ID for job dialog}
```

Data Types

```
TYPE TPStr8Ø = ^TStr8Ø;
     TStr8Ø  = STRING[8Ø];

     TPRect  = ^Rect;

     TPPrPort = ^TPrPort;
     TPrPort  = RECORD
                  gPort:  GrafPort; {grafPort to draw in}
                  gProcs: QDProcs;  {pointers to drawing routines}
                  {more fields for internal use only}
                END;

     TPPort = PACKED RECORD
                CASE INTEGER OF
                  Ø: (pGPort:  GrafPtr);
                  1: (pPrPort: TPPrPort)
              END;

     THPrint = ^TPPrint;
     TPPrint = ^TPrint;
     TPrint  = RECORD
                 iPrVersion: INTEGER;  {Printing Manager version}
                 prInfo:     TPrInfo;  {printer information}
                 rPaper:     Rect;     {paper rectangle}
                 prStl:      TPrStl;   {style information}
                 prInfoPT:   TPrInfo;  {copy of PrInfo}
                 prXInfo:    TPrXInfo; {band information}
                 prJob:      TPrJob;   {job information}
                 printX:     ARRAY [1..19] OF INTEGER
                                       {used internally}
               END;

     TPrInfo = RECORD
                 iDev:  INTEGER; {driver information}
                 iVRes: INTEGER; {printer vertical resolution}
                 iHRes: INTEGER; {printer horizontal resolution}
                 rPage: Rect     {page rectangle}
               END;
```

```
TPrStl = RECORD
           wDev:   TWord;      {used internally}
           iPageV: INTEGER;    {paper height}
           iPageH: INTEGER;    {paper width}
           bPort:  SignedByte; {printer or modem port}
           feed:   TFeed       {paper type}
         END;

TFeed = (feedCut,     {hand-fed, individually cut sheets}
         feedFanfold, {continuous-feed fanfold paper}
         feedMechCut, {mechanically fed cut sheets}
         feedOther);  {other types of paper}

TPrJob  = RECORD
            iFstPage:  INTEGER;    {first page to print}
            iLstPage:  INTEGER;    {last page to print}
            iCopies:   INTEGER;    {number of copies}
            bJDocLoop: SignedByte; {printing method}
            fFromUsr:  BOOLEAN;   {TRUE if called from application}
            pIdleProc: ProcPtr;    {background procedure}
            pFileName: TPStr8Ø;    {spool file name}
            iFileVol:  INTEGER;    {volume reference number}
            bFileVers: SignedByte; {version number of spool file}
            bJobX:     SignedByte  {not used}
          END;

TPrXInfo = RECORD
             iRowBytes: INTEGER;    {bytes per row}
             iBandV:    INTEGER;    {vertical dots}
             iBandH:    INTEGER;    {horizontal dots}
             iDevBytes: INTEGER;    {size of bit image}
             iBands:    INTEGER;    {bands per page}
             bPatScale: SignedByte; {used by QuickDraw}
             bUlThick:  SignedByte; {underline thickness}
             bUlOffset: SignedByte; {underline offset}
             bUlShadow: SignedByte; {underline descender}
             scan:      TScan;      {scan direction}
             bXInfoX:   SignedByte  {not used}
           END;

TScan = (scanTB,  {scan top to bottom}
         scanBT,  {scan bottom to top}
         scanLR,  {scan bottom to top}
         scanRL); {scan right to left}
```

```
     TPrStatus = RECORD
                    iTotPages:  INTEGER;  {total number of pages}
                    iCurPage:   INTEGER;  {page being printed}
                    iTotCopies: INTEGER;  {number of copies}
                    iCurCopy:   INTEGER;  {copy being printed}
                    iTotBands:  INTEGER;  {bands per page}
                    iCurBand:   INTEGER;  {band being printed}
                    fPgDirty:   BOOLEAN;  {TRUE if started printing page}
                    fImaging:   BOOLEAN;  {TRUE if imaging}
                    hPrint:     THPrint;  {print record}
                    pPrPort:    TPPrPort; {printing port}
                    hPic:       PicHandle {used internally}
                 END;
```

Routines

Initialization and Termination

```
PROCEDURE PrOpen;
PROCEDURE PrClose;
```

Print Records and Dialogs

```
PROCEDURE PrintDefault (hPrint: THPrint);
FUNCTION  PrValidate   (hPrint: THPrint) : BOOLEAN;
FUNCTION  PrStlDialog  (hPrint: THPrint) : BOOLEAN;
FUNCTION  PrJobDialog  (hPrint: THPrint) : BOOLEAN;
PROCEDURE PrJobMerge   (hPrintSrc,hPrintDst: THPrint);
```

Document Printing

```
FUNCTION  PrOpenDoc   (hPrint: THPrint; pPrPort: TPPrPort; pIOBuf: Ptr) :
                       TPPrPort;
PROCEDURE PrCloseDoc  (pPrPort: TPPrPort);
PROCEDURE PrOpenPage  (pPrPort: TPPrPort; pPageFrame: TPRect);
PROCEDURE PrClosePage (pPrPort: TPPrPort);
```

Spool Printing

```
PROCEDURE PrPicFile (hPrint: THPrint; pPrPort: TPPrPort; pIOBuf: Ptr;
                     pDevBuf: Ptr; VAR prStatus: TPrStatus);
```

Handling Errors [Pascal only]

```
FUNCTION  PrError :  INTEGER;
PROCEDURE PrSetError (iErr: INTEGER);
```

Low-Level Driver Access

```
PROCEDURE PrDrvrOpen;
PROCEDURE PrDrvrClose;
PROCEDURE PrCtlCall   (iWhichCtl: INTEGER; lParam1,lParam2,lParam3:
                       LongInt);
FUNCTION  PrDrvrDCE : Handle;
FUNCTION  PrDrvrVers : INTEGER;
PROCEDURE PrNoPurge;
PROCEDURE PrPurge;
```

Resource File Contents

System Resource File

Resource	Resource type	Resource ID
Name of the current printer resource file	'STR '	-8192
A copy of the device driver for the currently installed printer	'DRVR'	2
A copy of the driver's private data storage	'PREC'	2

Printer Resource File

Resource	Resource type	Resource ID
Original copy of the device driver for this printer	'DRVR'	-8192
Original copy of the driver's private storage	'PREC'	-8192
Printer-specific code used to implement Printing Manager routines	'PDEF'	Ø through 6
Default print record for use with this printer	'PREC'	Ø
Print record from the previous printing operation	'PREC'	1
Default spool file name	'STR '	-8191
Style dialog	'DLOG'	-8192
Job dialog	'DLOG'	-8191
Installation dialog	'DLOG'	-819Ø
Alert definitions	'ALRT'	(private)
Dialog and alert item lists	'DITL'	(private)

Assembly-Language Information

Constants

```
; Result codes

iMemFullErr    .EQU     -1Ø8     ;not enough heap space
noErr          .EQU        Ø     ;no error

; Printing methods

bDraftLoop     .EQU    Ø     ;draft printing
bSpoolLoop     .EQU    1     ;spooling
bUser1Loop     .EQU    2     ;printer-specific, method 1
bUser2Loop     .EQU    3     ;printer-specific, method 2

; Printer Driver Control call parameters

iPrBitsCtl      .EQU    4            ;bitMap printing
```

```
lScreenBits   .EQU    0            ;  configurable
lPaintBits    .EQU    1            ;  72 by 72 dots
iPrIOCtl      .EQU    5            ;text streaming
iPrEvtCtl     .EQU    6            ;screen printing
iPrEvtAll     .EQU    $00FFFFFD    ;  print whole screen
iPrEvtTop     .EQU    $00FEFFFD    ;  print top (frontmost) window
iPrDevCtl     .EQU    7            ;device control
lPrReset      .EQU    1            ;  reset printer
lPrPageEnd    .EQU    2            ;  start new page
lPrLineFeed   .EQU    3            ;  start new line
iFMgrCtl      .EQU    8            ;used by the Font Manager

; Miscellaneous

iPrintSize    .EQU   120       ;length of print record
iPrPortSize   .EQU   178       ;length of printing port
iPrStatSize   .EQU   26        ;length of printer status record
iPrAbort      .EQU   128       ;result code for halting printing
iPrRelease    .EQU   2         ;current version number of Printing
                               ; Manager
lPfType       .EQU   $5046484C ;file type ('PFIL') for spool files
lPfSig        .EQU   $50535953 ;signature ('PSYS') of Printer program

; Printing resources

iPrDrvrRef    .EQU   -3        ;Printer Driver reference number
lPrintType    .EQU   $50524543 ;type ('PREC') for print records
                               ; and private storage
iPrintDef     .EQU   0         ;ID for default print record
iPrintLst     .EQU   1         ;ID for previous print record
iPrDrvrID     .EQU   2         ;ID for Printer Driver and its
                               ; private storage in system
                               ; resource file
lPStrType     .EQU   $53545220 ;type 'STR ' for file name
                               ; resources
iPStrRFil     .EQU   $E000     ;ID for printer resource file
                               ; name
iPStrPFil     .EQU   $E001     ;ID for default spool file name
iPrStlDlg     .EQU   $E000     ;ID for style dialog
iPrJobDlg     .EQU   $E001     ;ID for job dialog

; Resource IDs for code overlays

iPrDraftID    .EQU        0     ;draft printing
iPrSpoolID    .EQU        1     ;spooling
iPrUser1ID    .EQU        2     ;printer-specific printing, method 1
iPrUser2ID    .EQU        3     ;printer-specific printing, method 2
iPrDlgsID     .EQU        4     ;print records and dialogs
iPrPicID      .EQU        5     ;spool printing
```

```
; Offsets to document printing code overlays

lOpenDoc      .EQU        $000C0000     ;PrOpenDoc
lCloseDoc     .EQU        $00048004     ;PrCloseDoc
lOpenPage     .EQU        $00080008     ;PrOpenPage
lClosePage    .EQU        $0004000C     ;PrClosePage

; Offsets to print record and dialog code overlays

lDefault      .EQU        $00048000     ;PrintDefault
lStlDialog    .EQU        $00048004     ;PrStlDialog
lJobDialog    .EQU        $00048008     ;PrJobDialog
lStlInit      .EQU        $0004000C     ;PrStlInit
lJobInit      .EQU        $00040010     ;PrJobInit
lDlgMain      .EQU        $00048014     ;PrDlgMain
lValidate     .EQU        $00048018     ;PrValidate
lJobMerge     .EQU        $0008801C     ;PrJobMerge

; Offset to spool printing code overlay

lPrPicFile    .EQU        $00148000     ;PrPicFile
```

Printing Port

gPort	GrafPort to draw in
gProcs	Pointers to drawing routines

Print Record

iPrVersion	Printing Manager version
prInfo	Printer information
rPaper	Paper rectangle
prStl	Style information
prJob	Job information

Printer Information Subrecord

iDev	Driver information
iVRes	Printer vertical resolution
iHRes	Printer horizontal resolution
rPage	Page rectangle

Style Subrecord

iPageV	Paper height
iPageH	Paper width
bPort	Printer or modem port
feed	Paper type

Job Subrecord

iFstPage	First page to print
iLstPage	Last page to print
iCopies	Number of copies
bJDocLoop	Printing method
fFromApp	Nonzero if called from application
pIdleProc	Pointer to background procedure
pFileName	Spool file name
iFileVol	Volume reference number
bFileVers	Version number spool file

Band Information Subrecord

iRowBytes	Bytes per row
iBandV	Vertical dots
iBandH	Horizontal dots
iDevBytes	Size of bit image
iBands	Bands per page
bPatScale	Used by QuickDraw
bUlThick	Underline thickness
bUlOffset	Underline offset
bUlShadow	Underline descender
scan	Scan direction

Printer Status Record

iTotPages	Total number of pages
iCurPage	Page being printed
iTotCopies	Number of copies
iCurCopy	Copy being printed
iTotBands	Bands per page
iCurBand	Band being printed
fPgDirty	Nonzero if started printing page
fImaging	Nonzero if imaging
hPrint	Print record
pPrPort	Printing port

Variables

Name	Size	Contents
printVars+iPrErr	2 bytes	Current result code

GLOSSARY

background procedure: A procedure passed to the Printing Manager to be run during idle times in the printing process.

band: One of the sections into which a page is divided for imaging and printing.

draft printing: Printing a document by using QuickDraw calls to drive the printer's character generator directly.

imaging: The process of converting an application's description of an image (such as a QuickDraw picture) into an actual array of bits to be displayed or printed.

job dialog: A dialog pertaining to one particular printing job; conventionally associated with the application's Print command.

landscape orientation: The positioning of a document in a printer with the long dimension of the paper running horizontally.

page rectangle: The rectangle marking the boundaries of a printed page image.

paper rectangle: The rectangle marking the boundaries of the physical sheet of paper on which a page is printed.

portrait orientation: The positioning of a document in a printer with the long dimension of the paper running vertically.

Printer: A special application program for printing spool files from a disk and configuring different printers.

Printer Driver: The device driver for the currently installed printer.

printer resource file: A file containing all the resources needed to run the Printing Manager with a particular printer.

printer status record: A record used by the Printing Manager to report on the progress of printing operations.

printing port: A special grafPort customized for printing instead of drawing on the screen.

print record: A record containing all the information needed by the Printing Manager to perform a particular printing job.

spool file: A disk file created as the result of spooling.

spooling: Writing a representation of a document's printed image to a disk file, rather than directly to the printer.

spool printing: Printing the image contained in a spool file.

style dialog: A dialog pertaining to the use of the printer for a particular document; conventionally associated with the application's Page Setup command.

MACINTOSH USER EDUCATION

The Device Manager: A Programmer's Guide /DMGR/DEVICE

See Also: The Macintosh User Interface Guidelines
The Memory Manager: A Programmer's Guide
The File Manager: A Programmer's Guide
The Desk Manager: A Programmer's Guide
The Vertical Retrace Manager: A Programmer's Guide
Inside Macintosh: A Road Map
Programming Macintosh Applications in Assembly Language

Modification History: First Draft (ROM 7) Bradley Hacker 6/15/84

ABSTRACT

This manual describes the Device Manager, the part of the Macintosh Operating System that controls the exchange of information between a Macintosh application and devices. It also discusses interrupts.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Device Manager, the part of the Macintosh Operating System that controls the exchange of information between a Macintosh application and devices. It also discusses interrupts. *** Eventually it will become part of the comprehensive Inside Macintosh manual. *** General information about using and writing device drivers can be found in this manual; specific information about the standard Macintosh drivers is contained in separate manuals.

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal. You should also be familiar with the basic concepts behind the Macintosh Operating System's Memory Manager.

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 Device Manager and what you can do with it. It then discusses some basic concepts behind the Device Manager: what devices and device drivers are and how they're used.

A section on using the Device Manager introduces its routines and tells how they fit into the flow of your application. This is followed by detailed descriptions of all the commonly used Device Manager routines, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that provide information for programmers who want to write their own drivers, including a discussion of interrupts and a sample device driver.

Finally, there's a summary of the Device Manager, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE DEVICE MANAGER

The Device Manager is the part of the Operating System that handles communication between applications and devices. A device is a part of the Macintosh, or a piece of external equipment, that can transfer information into or out of the Macintosh. Macintosh devices include disk drives, two serial communications ports, the sound generator, and printers. The video screen is **not** a device; drawing on the screen is handled by QuickDraw.

There are two kinds of devices: character devices and block devices. A character device reads or writes a stream of characters, one at a time: it can neither skip characters nor go back to a previous character. A character device is used to get information from or send information to the world outside of the Macintosh Operating System and memory: it can be an input device, an output device, or an input/output device. The serial ports and printers are all character devices.

A block device reads and writes blocks of 512 characters at a time; it can read or write any accessible block on demand. A block device is usually used to store and retrieve information; disk drives are block devices.

Applications communicate with devices through the Device Manager--either directly, or indirectly through another Operating System or Toolbox "Manager". For example, an application can communicate with a disk drive directly via the Device Manager, or indirectly via the File Manager (which calls the Device Manager). The Device Manager doesn't manipulate devices directly; it calls device drivers that do (Figure 1). Device drivers are programs that take data coming from the Device Manager and convert them into actions of devices, and convert device actions into data for the Device Manager to process.

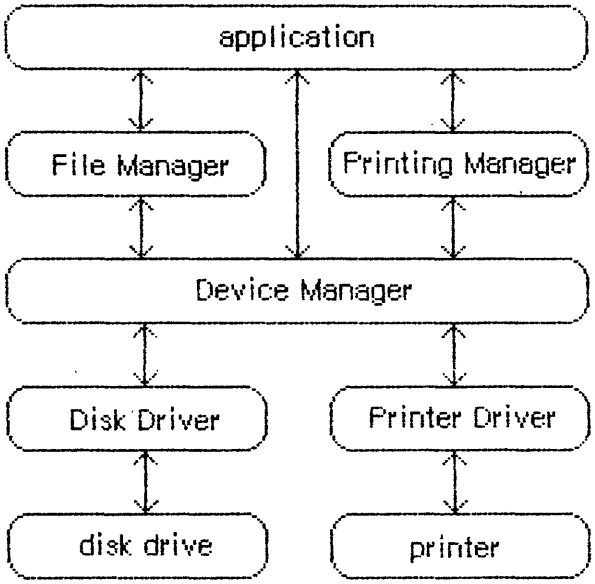


Figure 1. Communication with Devices

The Operating System includes three standard device drivers in ROM: the Disk Driver, the Sound Driver, and the ROM Serial Drivers. There are also a number of standard RAM drivers: the Printer Driver, the RAM Serial Drivers, and desk accessories. RAM drivers are resources, and are read from the system resource file as needed.

You can add other drivers independently or build on top of the existing drivers (for example, the Printer Driver is built on top of the Serial Driver); the section "Writing Your Own Device Drivers" describes how to do this. Desk accessories are a special type of device driver, and are manipulated via the specialized routines of the Desk Manager.

(warning)

> Information about desk accessories covered in the Desk Manager manual will not be repeated here. Some information in this manual may not apply to desk accessories.

A device driver can be either open or closed. The Sound Driver and Disk Driver are opened when the system starts up--the rest of the drivers are opened at the specific request of an application. After a driver has been opened, an application can read data from and write data to the driver. You can close device drivers that are no longer in use, and recover the memory used by them. Up to 32 device drivers may be open at any one time.

Before it's opened, you identify a device driver by its driver name; after it's opened, you identify it by its reference number. A driver name consists of a period (.) followed by any sequence of 1 to 254 printing characters. A RAM driver's name is the same as its resource name. You can use uppercase and lowercase letters when naming drivers, but the Device Manager ignores case when comparing names (it doesn't ignore diacritical marks).

(note)

> Although device driver names can be quite long, there's little reason for them to be more than a few characters in length.

The Device Manager assigns each open device driver a driver reference number, from -1 to -32, that's used instead of its driver name to refer to it.

Most communication between an application and an open device driver occurs by reading and writing data. Data read from a driver is placed in the application's data buffer, and data written to a driver is taken from the application's data bufffer. A data buffer is memory allocated by the application for communication with drivers.

In addition to data that's read from or written to device drivers, drivers may require or provide other information. Information transmitted to a driver by an application is called control information; information provided by a driver is called status information. Control information may select modes of operation, start or stop processes, enable buffers, choose protocols, and so on. Status information may indicate the current mode of operation, the readiness of the device, the occurrence of errors, and so on. Each device driver may respond to a number of different types of control information and may provide a number of different types of status information.

Each of the standard Macintosh drivers includes predefined calls for transmitting control information and receiving status information. Explanations of these calls can be found in the manuals describing the drivers.

USING THE DEVICE MANAGER

This section discusses how the Device Manager routines for calling device drivers fit into the general flow of an application program and gives an idea of what routines you'll need to use. The routines themselves are described in detail in the section "Device Manager Routines". The Device Manager routines for writing device drivers are described in the section "Writing Your Own Device Drivers"

You can call Device Manager routines via three different methods: high-level Pascal calls, low-level Pascal calls, and assembly language. The high-level Pascal calls are designed for Pascal programmers interested in using the Device Manager in a simple manner; they provide adequate device I/O and don't require much special knowledge to use. The low-level Pascal and assembly-language calls are designed for advanced Pascal programmers and assembly-language programmers interested in using the Device Manager to its fullest capacity; they require some special knowledge to be used most effectively.

(note)

> The names used to refer to routines here are actually assembly-language macro names for the low-level routines, but the Pascal routine names are very similar.

The Device Manager is automatically initialized each time the system is started up.

Before an application can exchange information with a device driver, it must open the driver. ROM drivers are opened when the system starts up; for RAM drivers, call Open. The Device Manager will return the driver reference number that you'll use every time you want to refer to that device driver.

An application can send data from its data buffer to an open driver with a Write call, and transfer data from an open driver to its data buffer with Read. An application passes control information to a device driver by calling Control, and receives status information from a driver by calling Status.

Whenever you want to stop a device driver from completing I/O initiated by a Read, Write, Control, or Status call, call KillIO. KillIO halts any current I/O and deletes any pending I/O. For example, you could use KillIO to implement a Cancel button that interrupts printing by your application.

When you're through using a driver, call Close. Close forces the device driver to complete any pending I/O, and then releases all the memory used by the driver.

DEVICE MANAGER ROUTINES

This section describes the Device Manager routines used to call drivers. It's divided into two parts. The first describes all the high-level Pascal routines of the Device Manager, and the second presents information about calling the low-level Pascal and assembly-language routines.

All the Device Manager routines in this section return a result code of type OSErr. Each routine description lists all of the applicable result codes, along with a short description of what the result code means. Lengthier explanations of all the result codes can be found in the summary at the end of this manual.

High-Level Device Manager Routines

The Pascal calls in this section cannot be invoked from assembly language; see the following section for equivalent calls.

(note)

As described in the File Manager manual, the FSRead and FSWrite routines are also used to read from and write to files.

FUNCTION OpenDriver (name: Str255; VAR refNum: INTEGER) : OSErr;

OpenDriver opens the device driver specified by name and returns its reference number in refNum.

Result codes		
	noErr	No error
	badUnitErr	Bad reference number
	dInstErr	Couldn't find driver in resource file
	openErr	Driver cannot perform the requested reading or writing
	unitEmptyErr	Bad reference number

FUNCTION CloseDriver (refNum: INTEGER) : OSErr;

CloseDriver closes the device driver having the reference number refNum. Any pending I/O is completed, and the memory used by the driver is released.

Result codes		
	noErr	No error
	badUnitErr	Bad reference number
	dRemoveErr	Tried to remove an open driver
	unitEmptyErr	Bad reference number

```
FUNCTION FSRead (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr) :
          OSErr;
```

FSRead attempts to read the number of bytes specified by the count parameter from the device driver having the reference number refNum, and transfer them to the data buffer pointed to by buffPtr. After the read operation is completed, the number of bytes actually read is returned in the count parameter.

<u>Result codes</u>	noErr	No error
	badUnitErr	Bad reference number
	notOpenErr	Driver isn't open
	unitEmptyErr	Bad reference number
	readErr	Driver can't respond to Read calls

```
FUNCTION FSWrite (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr) :
          OSErr;
```

FSWrite attempts to take the number of bytes specified by the count parameter from the buffer pointed to by buffPtr and write them to the open device driver having the reference number refNum. After the write operation is completed, the number of bytes actually written is returned in the count parameter.

<u>Result codes</u>	noErr	No error
	badUnitErr	Bad reference number
	notOpenErr	Driver isn't open
	unitEmptyErr	Bad reference number
	writErr	Driver can't respond to Write calls

```
FUNCTION Control (refNum: INTEGER; csCode: INTEGER; csParam: Ptr) :
          OSErr;
```

Control sends control information to the device driver having the reference number refNum. The type of information sent is specified by csCode, and the information itself is pointed to by csParam. The values passed in csCode and pointed to by csParam depend on the driver being called.

<u>Result codes</u>	noErr	No error
	badUnitErr	Bad reference number
	notOpenErr	Driver isn't open
	unitEmptyErr	Bad reference number
	controlErr	Driver can't respond to this Control call

```
FUNCTION Status (refNum: INTEGER; csCode: INTEGER; csParam: Ptr) :
            OSErr;
```

Status returns status information about the device driver having the reference number refNum. The type of information returned is specified by csCode, and the information itself is pointed to by csParam. The values passed in csCode and pointed to by csParam depend on the driver being called.

Result codes		
	noErr	No error
	badUnitErr	Bad reference number
	notOpenErr	Driver isn't open
	unitEmptyErr	Bad reference number
	statusErr	Driver can't respond to this Status call

```
FUNCTION KillIO (refNum: INTEGER) : OSErr;
```

KillIO terminates all current and pending I/O with the device driver having the reference number refNum.

Result codes		
	noErr	No error
	badUnitErr	Bad reference number
	unitEmptyErr	Bad reference number
	controlErr	Driver can't respond to KillIO calls

(note)

> KillIO is actually a special type of PBControl call, and all information about PBControl calls applies equally to KillIO.

Low-Level Device Manager Routines

This section contains special information for programmers using the low-level Pascal or assembly-language routines of the Device Manager, and then describes the routines in detail.

All low-level Device Manager routines can be executed either synchronously (meaning that the application cannot continue until the I/O is completed) or asynchronously (meaning that the application is free to perform other tasks while the I/O is being completed).

When you call a Device Manager routine asynchronously, an I/O request is placed in the driver's I/O queue, and control returns to the calling application--even before the actual I/O is completed. Requests are taken from the queue one at a time (in the same order that they were entered), and processed. Only one request per driver may be processed at any given time.

The calling application may specify a completion routine to be executed as soon as the I/O operation has been completed.

Routine parameters passed by an application to the Device Manager and returned by the Device Manager to an application are contained in a parameter block, which is memory space in the heap or stack. All low-level Pascal calls to the Device Manager are of the form

```
PBCallName (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

PBCallName is the name of the routine. ParamBlock points to the parameter block containing the parameters for the routine. If async is TRUE, the call is executed asynchronously; if FALSE, it's executed synchronously.

Assembly-language note: When you call a Device Manager routine, AØ must point to a parameter block containing the parameters for the routine. If you want the routine to be executed asynchronously, set bit 1Ø of the routine trap word. You can do this by supplying the word ASYNC as the second argument to the routine macro. For example:

```
_Read   ,ASYNC
```

You can set or test bit 1Ø of a trap word by using the global constant asynTrpBit.

If you want a routine to be executed immediately (bypassing the driver's I/O queue), set bit 9 of the routine trap word. This can be accomplished by supplying the word IMMED as the second argument to the routine macro. (The driver must be able to handle immediate calls for this to work.) For example

```
_Write   ,IMMED
```

You can set or test bit 9 of a trap word by using the global constant noQueueBit. You can specify either ASYNC or IMMED, but not both.

All routines return a result code in DØ.

Routine Parameters

The lengthy, variable-length data structure of a parameter block is given below. The Device Manager and File Manager use this same data structure, but only the parts relevant to the Device Manager are discussed here. Each kind of parameter block contains eight fields of standard information and two to nine fields of additional information:

```
TYPE ParamBlkType = (ioParam, fileParam, volumeParam, cntrlParam);

     ParamBlockRec = RECORD
                       qLink:        QElemPtr;  {next queue entry}
                       qType:        INTEGER;   {queue type}
                       ioTrap:       INTEGER;   {routine trap}
                       ioCmdAddr:    Ptr;       {routine address}
                       ioCompletion: ProcPtr;   {completion routine}
                       ioResult:     OSErr;     {result code}
                       ioNamePtr:    StringPtr; {driver name}
                       ioVRefNum:    INTEGER;   {used by Disk Driver}
                       CASE ParamBlkType OF
                         ioParam:
                         . . .  {I/O routine parameters}
                         fileParam:
                         . . .  {used by File Manager}
                         volumeParam:
                         . . .  {used by File Manager}
                         cntrlParam:
                         . . .  {Control and Status call parameters}
                     END;

     ParmBlkPtr = ^ParamBlockRec;
```

The first four fields in each parameter block are handled entirely by the Device Manager, and most programmers needn't be concerned with them; programmers who are interested in them should see the section "The Structure of a Device Driver".

IOCompletion contains the address of a completion routine to be executed at the end of an asynchronous call; it should be NIL for asynchronous calls with no completion routine, and is automatically set to NIL for all synchronous calls. For asynchronous calls, ioResult is positive while the routine is executing, and returns the result code.

IONamePtr is a pointer to the name of a driver and is used only for calls to the PBOpen routine. IOVRefNum is used by the Disk Driver to identify volumes.

An 8-field parameter block is adequate for opening a driver, but most of the Device Manager routines require longer parameter blocks, as described below.

I/O routines use seven additional fields:

```
      ioParam:
        (ioRefNum:    INTEGER;    {driver reference number}
         ioVersNum:   SignedByte; {not used}
         ioPermssn:   SignedByte; {read/write permission}
         ioMisc:      Ptr;        {not used}
         ioBuffer:    Ptr;        {data buffer}
         ioReqCount:  LongInt;    {requested number of bytes}
         ioActCount:  LongInt;    {actual number of bytes}
         ioPosMode:   INTEGER;    {type of positioning operation}
         ioPosOffset: LongInt);   {size of positioning offset}
```

IOPermssn requests permission to read from or write to a driver when the driver is opened, and must contain one of the following predefined constants:

```
      fsCurPerm   = Ø; {whatever is currently allowed}
      fsRdPerm    = 1; {request to read only}
      fsWrPerm    = 2; {request to write only}
      fsRdWrPerm  = 3; {request to read and write}
```

This request is compared with the capabilities of the driver (some drivers are read-only, some are write-only). If the driver is incapable of performing as requested, an error will be returned.

IOBuffer points to an application's data buffer into which data is written by Read calls and from which data is read by Write calls. IOReqCount specifies the requested number of bytes to be read or written. IOActCount contains the number of bytes actually read or written.

Advanced programmers: IOPosMode and ioPosOffset contain positioning information used for Read and Write calls by drivers of block devices. Bits Ø and 1 of ioPosMode indicate a byte position from the physical beginning of the block-formatted medium (such as a disk); it must contain one of the following predefined constants:

```
      fsAtMark     = Ø; {at current position of mark }
                        { (ioPosOffset ignored)}
      fsFromStart  = 1; {offset relative to beginning of file}
      fsFromLEOF   = 2; {offset relative to logical end-of-file}
      fsFromMark   = 3; {offset relative to current mark}
```

IOPosOffset specifies the byte offset beyond ioPosMode where the operation is to be performed. Control and Status calls use two additional fields:

```
      cntrlParam:
        (csCode:  INTEGER;                {type of Control or Status call}
         csParam: ARRAY[Ø..Ø] OF Byte); {control or status information}
```

CSCode contains a number identifying the type of call. This number may be interpreted differently by each driver. The csParam field contains the control or status information for the call; it's declared as a zero-length array because its exact contents will vary depending from one Control or Status call to the next.

(note)

> Programmers who want to use the low-level Control and Status calls will need to declare their own data type that mimics all fields of the ParamBlockRec except for csParam. For example, if you want to pass a long integer in csParam, declare the following:
>
> ```
> TYPE MyParamBlockRec = RECORD
>                          qLink:   QElemPtr;
>                          . . .
>                          csCode:  INTEGER;
>                          csParam: LongInt;
>                        END;
>
> VAR  MyPBR: MyParamBlockRec;
> ```
>
> Then pass @MyPBR (a pointer to your variable) to the low-level Control and Status routines.

Routine Descriptions

This section describes the procedures and functions. Each routine description includes the low-level Pascal form of the call and the routine's assembly-language macro. A list of the fields in the parameter block affected by the call is also given.

Assembly-language note: The field names given in these descriptions are those of the ParamBlockRec data type; see "Summary of the Device Manager" for the corresponding assembly-language equates.

The number next to each parameter name indicates the byte offset of the parameter from the start of the parameter block pointed to by AØ; only assembly-language programmers need be concerned with it. An arrow drawn next to each parameter name indicates whether it's an input, output, or input/output parameter:

Arrow	Meaning
-->	Parameter is passed to the routine
<--	Parameter is returned by the routine
<->	Parameter is passed to and returned by the routine

(note)

As described in the File Manager manual, the PBOpen and PBClose routines are also used to open and close files.

FUNCTION PBOpen (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Open

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
<--	24	ioRefNum	word
-->	27	ioPermssn	byte

Result codes

noErr	No error
badUnitErr	Bad reference number
dInstErr	Couldn't find driver in resource file
openErr	Driver cannot perform the requested reading or writing
unitEmptyErr	Bad reference number

PBOpen opens the device driver specified by ioNamePtr and returns its reference number in ioRefNum. IOPermssn specifies the requested read/write permission.

FUNCTION PBClose (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Close

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word

Result codes

noErr	No error
badUnitErr	Bad reference number
dRemoveErr	Tried to remove an open driver
unitEmptyErr	Bad reference number

PBClose closes the device driver having the reference number ioRefNum. Any pending I/O is completed, and the memory used by the driver is released.

FUNCTION PBRead (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Read

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word
-->	32	ioBuffer	pointer
-->	36	ioReqCount	long word
<--	4∅	ioActCount	long word
-->	44	ioPosMode	word
<->	46	ioPosOffset	long word

Result codes

noErr	No error
badUnitErr	Bad reference number
notOpenErr	Driver isn't open
unitEmptyErr	Bad reference number
readErr	Driver can't respond to Read calls

PBRead attempts to read ioReqCount bytes from the device driver having the reference number ioRefNum, and transfer them to the data buffer pointed to by ioBuffer. After the read operation is completed, the number of bytes actually read is returned in ioActCount.

Advanced programmers: If the driver is reading from a block device, the byte offset from the position indicated by ioPosMode, where the read should actually begin, is given by ioPosOffset.

```
FUNCTION PBWrite (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _Write

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word
-->	32	ioBuffer	pointer
-->	36	ioReqCount	long word
<--	4Ø	ioActCount	long word
-->	44	ioPosMode	word
-->	46	ioPosOffset	long word

Result codes

noErr	No error
badUnitErr	Bad reference number
notOpenErr	Driver isn't open
unitEmptyErr	Bad reference number
writErr	Driver can't respond to Write calls

PBWrite attempts to take ioReqCount bytes from the buffer pointed to by ioBuffer and write them to the device driver having the reference number ioRefNum. After the write operation is completed, the number of bytes actually written is returned in ioActCount.

Advanced programmers: If the driver is writing to a block device, ioPosMode indicates whether the write should begin relative to the beginning of the device or the current position. The byte offset from the position indicated by ioPosMode, where the write should actually begin, is given by ioPosOffset.

FUNCTION PBControl (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Control

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word
-->	26	csCode	word
-->	28	csParam	record

Result codes

noErr	No error
badUnitErr	Bad reference number
notOpenErr	Driver isn't open
unitEmptyErr	Bad reference number
controlErr	Driver can't respond to this Control call

PBControl sends control information to the device driver having the reference number ioRefNum. The type of information sent is specified by csCode, and the information itself begins at csParam. The values passed in csCode and csParam depend on the driver being called.

FUNCTION PBStatus (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Status

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word
-->	26	csCode	word
-->	28	csParam	record

Result codes

noErr	No error
badUnitErr	Bad reference number
notOpenErr	Driver isn't open
unitEmptyErr	Bad reference number
statusErr	Driver can't respond to this Status call

PBStatus returns status information about the device driver having the reference number ioRefNum. The type of information returned is specified by csCode, and the information itself begins at csParam. The values passed in csCode and csParam depend on the driver being called.

FUNCTION PBKillIO (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _KillIO

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word

Result codes

noErr	No error
badUnitErr	Bad reference number
unitEmptyErr	Bad reference number
controlErr	Driver can't respond to KillIO calls

KillIO stops any current I/O request being processed, and removes all pending I/O requests from the I/O queue of the device driver having the reference number ioRefNum. The completion routine of each pending I/O request is called, with ioResult equal to the following result code:

```
CONST abortErr = -27;
```

(note)

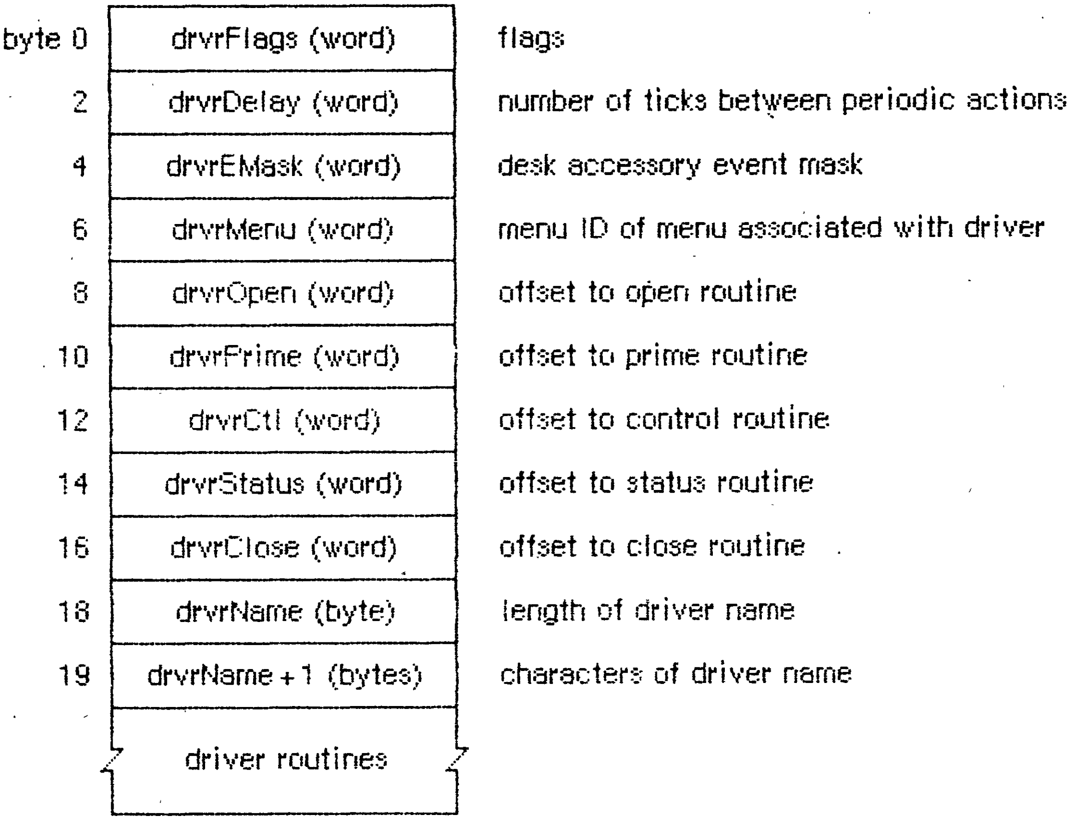
KillIO is actually a special type of Control call, and all information about Control calls applies equally to KillIO.

THE STRUCTURE OF A DEVICE DRIVER

This section describes the structure of device drivers for programmers interested in writing their own driver or manipulating existing drivers. Most of the information presented here is accessible only through assembly language.

RAM drivers are stored in resource files. The resource type for drivers is 'DRVR'. The resource name is the driver name. The resource ID for a driver is its unit number (explained below) and will be between Ø and 31 inclusive. Don't use the unit number of an existing driver unless you want the existing driver to be replaced.

As illustrated in Figure 2, a driver begins with a few words of flags and other data, followed by offsets to the routines that do the work of the driver, an optional title, and finally the routines themselves.



byte 0	drvrFlags (word)	flags
2	drvrDelay (word)	number of ticks between periodic actions
4	drvrEMask (word)	desk accessory event mask
6	drvrMenu (word)	menu ID of menu associated with driver
8	drvrOpen (word)	offset to open routine
10	drvrPrime (word)	offset to prime routine
12	drvrCtl (word)	offset to control routine
14	drvrStatus (word)	offset to status routine
16	drvrClose (word)	offset to close routine
18	drvrName (byte)	length of driver name
19	drvrName+1 (bytes)	characters of driver name
	driver routines	

Figure 2. Driver Structure

Every driver contains a routine to handle Open and Close calls, and may contain routines to handle Read, Write, Control, Status, and KillIO calls. The driver routines that handle Device Manager calls are as follows:

Device Manager call	Driver routine
Open	Open
Read	Prime
Write	Prime
Control	Control
KillIO	Control
Status	Status
Close	Close

For example, when a KillIO call is made to a driver, the driver's control routine must implement the call. Each bit of the **high-order** bytes of the drvrFlags word contains a flag:

```
dReadEnable     .EQU    Ø    ;set if driver can respond to Read calls
dWritEnable     .EQU    1    ;set if driver can respond to Write calls
dCtlEnable      .EQU    2    ;set if driver can respond to Control calls
dStatEnable     .EQU    3    ;set if driver can respond to Status calls
dNeedGoodBye    .EQU    4    ;set if driver needs to be called before the
                             ; application heap is reinitialized
dNeedTime       .EQU    5    ;set if driver needs time for performing a
                             ; periodic action
dNeedLock       .EQU    6    ;set if driver will be locked in memory as
                             ; soon as it's opened (always set for
                             ; ROM drivers)
```

Bits 8 through 11 indicate which Device Manager calls the driver's routines can respond to.

Unlocked RAM drivers that exist on the application heap will be lost every time the heap is reinitialized (when an application starts up, for example). If dNeedGoodBye is set, the control routine of the device driver will be called before the heap is reinitialized, and the driver can perform any "clean-up" actions it needs to. The driver's control routine identifies this "good-bye" call by checking the csCode parameter--it will be -1.

Device drivers may need to perform predefined actions periodically. For example, a network driver may want to poll its input buffer every ten seconds to see if it has received any messages. If the dNeedTime flag is set, the driver **does** need to perform a periodic action, and the drvrDelay word contains 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. Whether the action actually occurs this frequently depends on how often you call the Desk Manager routine SystemTask. SystemTask calls the driver's control routine (if the time indicated by drvrDelay has elapsed), and the control routine must perform whatever predefined action is desired. The driver's control routine identifies the SystemTask call by checking the csCode parameter--it will be the global constant accRun.

(note)

> Some drivers may not want to rely on the application to call SystemTask, and should install their own task in the vertical retrace queue to accomplish the desired action (see the Vertical Retrace Manager manual).

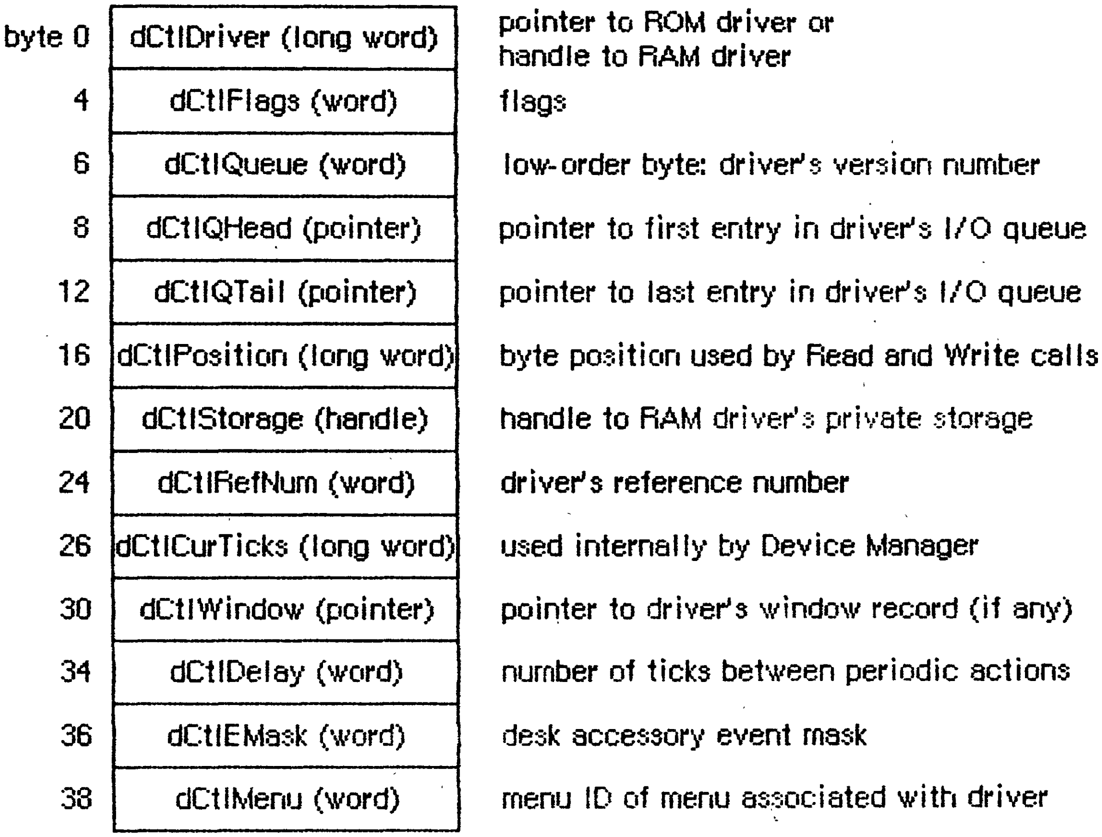
DrvrEMask and drvrMenu are used only for desk accessories and are discussed in the Desk Manager manual.

Following drvrMenu are the offsets to the driver routines, a title for the driver (preceded by its length in bytes), and the routines that do the work of the driver.

A Device Control Entry

The first time a driver is opened, information about it is read into a structure in memory called a device control entry. A device control entry tells the Device Manager the location of the driver's routines, the location of the driver's I/O queue, and other information. A device control entry is a 40-byte relocatable block located on the system heap. It's locked while the driver is open, and unlocked while the driver is closed.

The structure of a device control entry is illustrated in Figure 3. Notice that some of the data is taken from the first four words of the driver. Most of the data in the device control entry is stored and accessed only by the Device Manager, but in some cases the driver itself must store into it.



byte 0	dCtlDriver (long word)	pointer to ROM driver or handle to RAM driver
4	dCtlFlags (word)	flags
6	dCtlQueue (word)	low-order byte: driver's version number
8	dCtlQHead (pointer)	pointer to first entry in driver's I/O queue
12	dCtlQTail (pointer)	pointer to last entry in driver's I/O queue
16	dCtlPosition (long word)	byte position used by Read and Write calls
20	dCtlStorage (handle)	handle to RAM driver's private storage
24	dCtlRefNum (word)	driver's reference number
26	dCtlCurTicks (long word)	used internally by Device Manager
30	dCtlWindow (pointer)	pointer to driver's window record (if any)
34	dCtlDelay (word)	number of ticks between periodic actions
36	dCtlEMask (word)	desk accessory event mask
38	dCtlMenu (word)	menu ID of menu associated with driver

Figure 3. Device Control Entry

The low-order byte of the dCtlFlags word contains the following flags:

```
dOpened         .EQU    5    ;set if driver is open
dRAMBased       .EQU    6    ;set if driver is RAM-based
drvrActive      .EQU    7    ;set if driver is currently executing
```

The high-order byte contains information copied from the drvrFlags word of the driver:

```
dReadEnable     .EQU   Ø   ;set if driver can respond to Read calls
dWritEnable     .EQU   1   ;set if driver can respond to Write calls
dCtlEnable      .EQU   2   ;set if driver can respond to Control calls
dStatEnable     .EQU   3   ;set if driver can respond to Status calls
dNeedGoodBye    .EQU   4   ;set if driver needs to be called before the
                           ; application heap is reinitialized
dNeedTime       .EQU   5   ;set if driver needs time for performing a
                           ; periodic action
dNeedLock       .EQU   6   ;set if driver will be locked in memory as
                           ; soon as it's opened (always set for
                           ; ROM drivers)
```

DCtlPosition is used only by drivers of block devices, and indicates the current source or destination position of a Read or Write call. The position is given as a number of bytes beyond the physical beginning of the medium used by the device. For example, if one logical block of data has just been read from a 3 1/2-inch disk via the Disk Driver, dCtlPosition will be 512.

ROM drivers generally use locations in low memory for their local storage. RAM drivers may reserve memory within their code space, or allocate a relocatable block and keep a handle to it in dCtlStorage (if the block resides in the application heap, its handle will be set to NIL when the heap is reinitialized).

The Unit Table

The location of each device control entry is maintained in a list called the unit table. The unit table is a 128-byte nonrelocatable block containing 32 4-byte entries. Each entry has a number, from Ø to 31, called the unit number, and contains a handle to the device control entry for a driver. The unit number can be used as an index into the unit table to locate the handle to a specific driver's device control entry; it's equal to

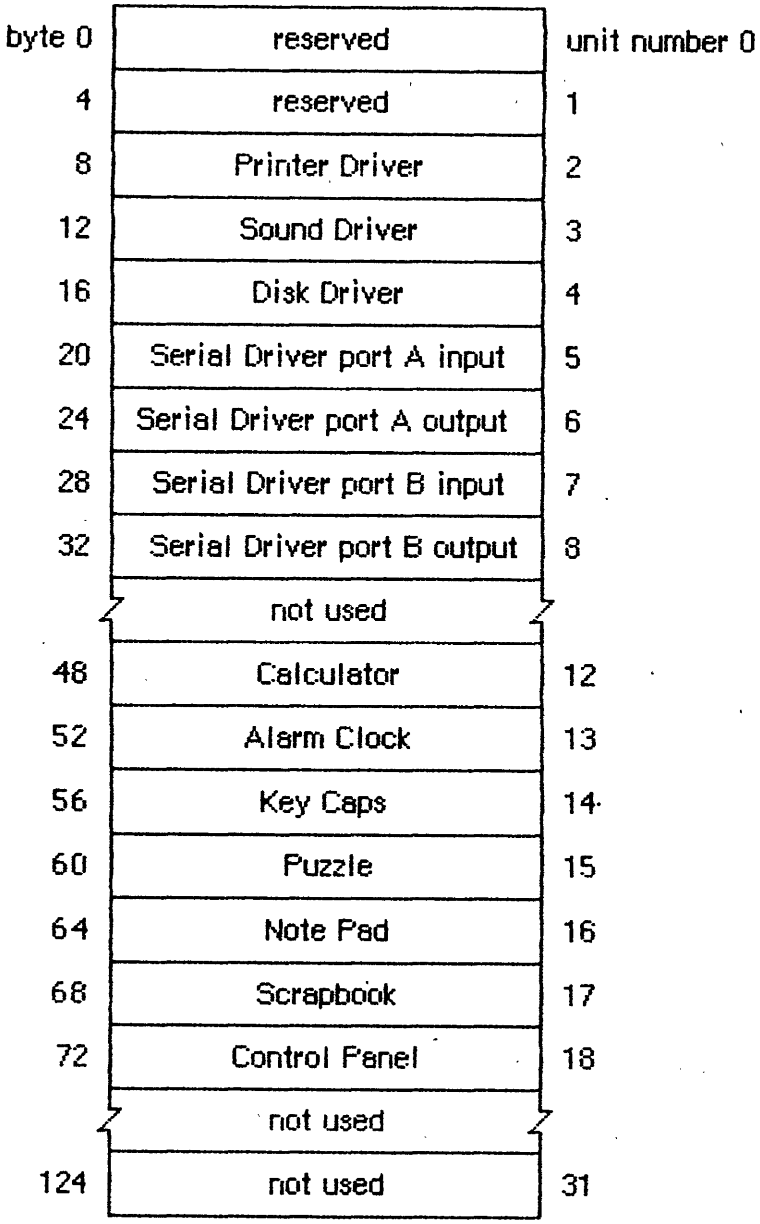
$$-1 * (refNum + 1)$$

where refNum is the driver's reference number. For example, the Sound Driver's reference number is -4 and its unit number is 3.

Figure 4 shows the layout of the unit table just after the system starts up.

(note)

> Any new drivers contained in resource files should have resource IDs that don't conflict with the unit numbers of existing drivers--unless you want an existing driver to be replaced.



byte		unit number
byte 0	reserved	unit number 0
4	reserved	1
8	Printer Driver	2
12	Sound Driver	3
16	Disk Driver	4
20	Serial Driver port A input	5
24	Serial Driver port A output	6
28	Serial Driver port B input	7
32	Serial Driver port B output	8
	not used	
48	Calculator	12
52	Alarm Clock	13
56	Key Caps	14
60	Puzzle	15
64	Note Pad	16
68	Scrapbook	17
72	Control Panel	18
	not used	
124	not used	31

Figure 4. The Unit Table

Assembly-language note: The global variable uTableBase points to the unit table.

Each device driver contains an I/O queue with a list of I/O requests to be completed by the driver. A driver I/O queue is a standard Operating

System queue (described in the Operating System Utilities manual *** doesn't yet exist; for now, see the appendix of the File Manager manual ***). The queue is located in the device control entry for the driver (Figure 5).

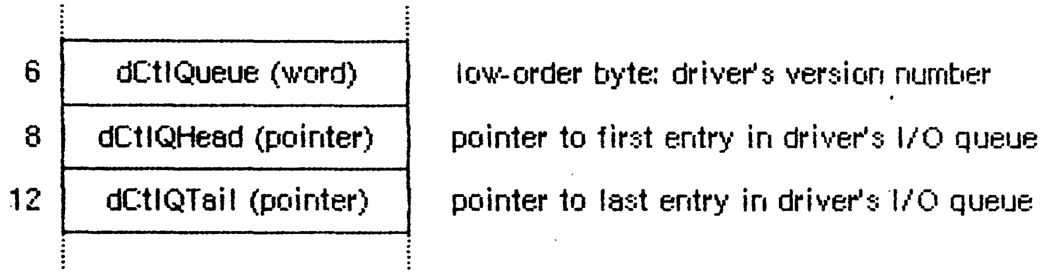

6
dCtlQueue (word)
low-order byte: driver's version number
8
dCtlQHead (pointer)
pointer to first entry in driver's I/O queue
12
dCtlQTail (pointer)
pointer to last entry in driver's I/O queue

Figure 5. Driver I/O Queue Structure

The three fields shown in Figure 5 are analogous to the QHdr data type of a standard Operating System queue.

Each driver I/O queue uses entries of type ioQType. Each entry in the queue consists of a parameter block for the routine that was called. The structure of this block is shown in part below:

```
TYPE ParamBlockRec = RECORD
                       qLink:      QElemPtr; {next queue entry}
                       qType:      INTEGER;  {queue type}
                       ioTrap:     INTEGER;  {routine trap}
                       ioCmdAddr:  Ptr    ;  {routine address}
                         . . .               {rest of block}
                     END;
```

QLink points to the next entry in the queue, and qType indicates the queue type, which must always be ORD(ioQType). IOTrap and ioCmdAddr contain the trap and address of the Device Manager routine that was called. You can use the following global constants to identify Device Manager traps, by comparing the global constant with the low-order byte of the trap:

```
aRdCmd      .EQU    2    ;Read call (trap $AØØ2)
aWrCmd      .EQU    3    ;Write call (trap $AØØ3)
aCtlCmd     .EQU    4    ;Control call (trap $AØØ4)
aStsCmd     .EQU    5    ;Status call (trap $AØØ5)
```

You can get a pointer to a driver's I/O queue by calling the Device Manager function GetDCtlQHdr.

FUNCTION GetDCtlQHdr (refNum: INTEGER) : QHdrPtr;   [Pascal only]

GetDCtlQHdr returns a pointer to the I/O queue of the device driver having the reference number refNum.

Assembly-language note: To access the contents of a driver's I/O queue from assembly language, you can use offsets from the address of the global variable dCtlQueue.

WRITING YOUR OWN DEVICE DRIVERS

This section describes what you'll need to do to write your own device driver. If you aren't interested in writing your own driver, skip ahead to the summary.

Drivers are usually written in assembly language. The structure of your driver must match that shown in the previous section. The routines that do the work of the driver should be written to operate the device in whatever way you require. Your driver must contain routines to handle Open and Close calls, and may choose to handle Read, Write, Control, Status, and KillIO calls as well.

When the Device Manager executes a driver routine to handle an application call, it passes a pointer to the call's parameter block in A∅ and a pointer to the driver's device control entry in A1. From this information, the driver can determine exactly what operations are required to fulfill the call's requests, and do them.

Open and close routines must execute synchronously. They needn't preserve any registers that they use. Open and close routines should place a result code in D∅ and return via an RTS instruction. *** Currently the Device Manager sets D∅ to zero upon return from an Open call. ***

The open routine must allocate any private storage required by the driver, store a handle to it in the device control entry (in the dCtlStorage field), initialize any local variables, and then be ready to receive a Read, Write, Status, Control, or KillIO call. It might also install interrupt handlers, change interrupt vectors, and store a pointer to the device control entry somewhere in its local storage for its interrupt handlers to use. The close routine must reverse the effects of the open routine, by releasing all used memory, removing interrupt handlers, and replacing changed interrupt vectors. If anything about the operational state of the driver should be saved until the next time the driver is opened, it should be kept in the relocatable block of memory pointed to by dCtlStorage.

Prime, control, and status routines must be able to respond to queued calls and asynchronous calls, and should be interrupt-driven. Asynchronous portions of the routines can use registers A∅ to A3 and D∅ to D3, but must preserve any other registers used; synchronous portions can use all registers. Prime, control, and status routines should return a result code in DØ. They must return via an RTS if called immediately (with IMMED as the second argument to the routine macro) or via an RTS if the device couldn't complete the I/O request right away, or via a JMP to the IODone routine (explained below) if the device completed the request.

(warning)

> If they can be called as the result of an interrupt, the prime, control, and status routines should never call Memory Manager routines that cause heap compactions.

The prime routine must implement all Read and Write calls made to the driver. It can distinguish between Read and Write calls by checking the value of the ioTrap field. You may want to use the Fetch and Stash routines described below to read and write characters. If the driver is for a block device, it should update the dCtlPosition field of the device control entry after each read or write. The control routine must accept the control information passed to it, and manipulate the device as requested. The status routine must return requested status information. Since both the control and status routines may be subjected to Control and Status calls sending and requesting a variety of information, they must be prepared to respond correctly to all types. The control routine must handle KillIO calls; the driver identifies KillIO calls by checking the csCode parameter--it will be the global constant killCode.

(warning)

> KillIO calls must return via an RTS, and shouldn't jump (via JMP) to the IODone routine.

Routines for Writing Drivers

The Device Manager includes three routines, Fetch, Stash, and IODone, that provide low-level support for driver routines. Include them in the code of your device driver if they're useful to you. Fetch, Stash, and IODone are invoked via "jump vectors" (jFetch, jStash, and jIODone) rather than macros (in the interest of speed). You use a jump vector by moving its address onto the stack:

```
        MOVE.L    jIODone,-(SP)
        RTS
```

Fetch and Stash don't return a result code, since the only result possible is dSIOCoreErr, which invokes the System Error Handler. IODone can return a result code.

Fetch Function

Jump vector   jFetch

On entry   A1: pointer to device control entry

On exit   DØ: character fetched; bit 15=1 if it's the last character in the data buffer

Fetch gets the next character from the data buffer pointed to by ioBuffer and places it in DØ. IOActCount is incremented by 1. If ioActCount equals ioReqCount, bit 15 of DØ is set. After receiving the last byte requested, the driver should call IODone.

Stash Function

Jump vector   jStash

On entry   A1: pointer to device control entry
DØ: character to stash

On exit   DØ: bit 15=1 if it's the last character requested

Stash places the character in DØ into the data buffer pointed to by ioBuffer, and increments ioActCount by 1. If ioActCount equals ioReqCount, bit 15 of DØ is set. After stashing the last byte requested, the driver should call IODone.

IODone Function

Jump vector	jIODone	
On entry	A1: pointer to device control entry	
On exit	DØ: result code	
Result codes	noErr	No error
	unitEmptyErr	Reference number specifies NIL handle in unit table

IODone removes the current I/O request from the driver's I/O queue, marks the driver inactive, unlocks the driver and its device control entry (if it's allowed to by the dNeedLock bit of the dCtlFlags word), and executes the completion routine (if there is one). Then it begins executing the next I/O request in the I/O queue.

A Sample Driver

Here's the skeleton of the Disk Driver, as an example of how a driver should be constructed.

```
; Driver header

DiskDrvr
                .WORD   $4FØØ                   ;RAM driver, read, write,
                                                ; control, status, needs
                                                ; lock
                .WORD   Ø,Ø                     ;no delay or event mask
                .WORD   Ø                       ;no menu

; Offsets to driver routines

                .WORD   DiskOpen-DiskDrvr       ;open
                .WORD   DiskPrime-DiskDrvr      ;prime
                .WORD   DiskControl-DiskDrvr    ;control
                .WORD   DiskStatus-DiskDrvr     ;status
                .WORD   allDone-DiskDrvr        ;close (just RTS)
                .BYTE   5                       ;length of name
                .ASCII  '.Disk'                 ;driver name

; Local variables and constants

; Driver routines

; Open routine

DiskOpen        MOVEQ   #<DiskVarLth/2>,DØ      ;get memory for variables
                . . .                           ;allocate variables
                . . .                           ;initialize drive queue
                . . .                           ;install a vertical-
```

```
                                             ; retrace task
DiskRTS         RTS

DiskDone        MOVE.W   DØ,DskErr           ;return result code
                MOVE.L   DiskVars,A1         ;get pointer to locals
                CLR.B    Active(A1)          ;driver isn't active
                MOVE.L   DiskUnitPtr(A1),A1  ;return pointer to DCE
                MOVE.L   jIODone,-(SP)       ;go to IODone
                RTS

; Prime routine

DiskPrime       ORI      #$Ø1ØØ, SR         ;exclude vertical-retrace
                                             ; interrupts
                . . .
                RTS

; Control routine
;               AØ (input): pointer to Control call's parameter block
;               csCode = killCode for KillIO, ejectCode for Eject

DiskControl     MOVE.W     csCode(AØ),DØ       ;get the control code
                SUBQ.W     #killCode,DØ        ;is it KillIO?
                BNE.S      @Ø                  ;branch if not
                MOVE       SR,-(SP)
                ORI        #$Ø1ØØ,SR           ;no VIA interrupts
                BSR        PowerDown           ;start power down,
                                               ; get VIA address
                MOVE.B     #$2Ø,VIER(A2)       ;remove any pending
                                               ; timer interrupts
                RTE                            ;special for KillIO

@Ø              SUBQ.W     #<ejectCode-killCode>,DØ   ;Eject?
                BEQ.S      @1                  ;branch if so
                MOVEQ      #controlErr,DØ      ;can't handle csCode
                BRA.S      DiskDone            ;exit

@1              BSR.S      CkDrvNum            ;set drive to eject
                . . .
                BRA.S      DiskDone            ;exit

; Status routine
;               AØ (input): pointer to Status call's parameter block
;               csCode = 8 for drive status

DiskStatus      MOVE.Q     #statusErr,DØ       ;assume status error

                CMP.W      #drvrStsCode,csCode(AØ) ;drive status call?
                BNE.S      DiskDone            ;exit for other calls

                BSR.S      CkDrvNum
                BNE.S      DiskDone            ;exit on error

                .END
```

Interrupts

This section discusses interrupts: how the Macintosh uses them, and how you can use them if you're writing your own device driver. Only programmers who want to write their own interrupt-driven device drivers need read this section. Programmers who want to build their own driver on top of a built-in Macintosh driver may be interested in some of the information presented here.

An interrupt is a form of exception: an error or abnormal condition detected by the processor in the course of program execution. Specifically, an interrupt is an exception that's signaled to the processor by a device, as distinct from a trap, which arises directly from the execution of an instruction. Interrupts are used by devices to notify the processor of a change in condition of the device, such as the completion of an I/O request. An interrupt causes the processor to suspend normal execution, save the address of the next instruction and the processor's internal status on the stack, and execute an interrupt handler.

The MC68ØØØ recognizes seven different levels of interrupt, each with its own interrupt handler. The addresses of the various handlers, called interrupt vectors, are kept in a vector table in the system communication area. Each level of interrupt has its own vector located in the vector table. When an interrupt occurs, the processor fetches the proper vector from the table, uses it to locate the interrupt handler for that level of interrupt, and jumps to the handler. On completion, the handler exits with an RTE instruction, which restores the internal state of the processor from the stack and resumes normal execution from the point of suspension.

There are three devices that can create interrupts: the 6522 Versatile Interface Adapter (VIA), the 853Ø Serial Communications Controller, and the debugging switch. They send a 3-bit number, from Ø to 7, called the interrupt priority level, to the processor. The interrupt level indicates which device is interrupting, and indicates which interrupt handler should be executed:

Level	Interrupting device
Ø	None
1	VIA
2	SCC
3	VIA and SCC
4-7	Debugging button

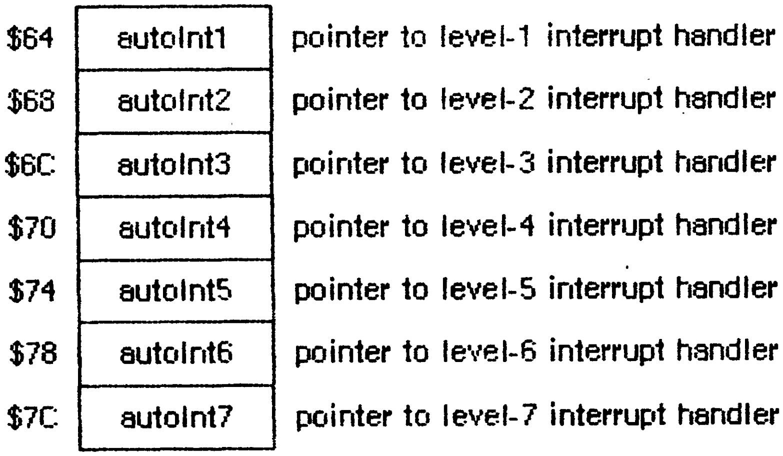
A level-3 interrupt occurs when both the VIA and SCC interrupt at the same instant; the interrupt handler for a level-3 interrupt is simply an RTE instruction. Debugging interrupts shouldn't occur during the normal execution of an application.

The interrupt priority level is compared with the processor priority in bits 8, 9, and 1Ø of the status register. If the interrupt priority level is greater than the processor priority, the MC68ØØØ acknowledges the interrupt and initiates interrupt processing. The processor priority determines which interrupting devices are ignored, and which are serviced:

Level	Services
Ø	All interrupts
1	VIA and debugging interrupts only
2	SCC and debugging interrupts only
3-6	Debugging interrupts only
7	No interrupts

When an interrupt is acknowledged, the processor priority is set to the interrupt priority level, to prevent additional interrupts of equal or lower priority, until the interrupt handler has finished servicing the interrupt.

The interrupt priority level is used as an index into the primary interrupt vector table. This table contains seven long words beginning at address $64. Each long word contains the starting address of an interrupt handler (Figure 6).



$64	autoInt1	pointer to level-1 interrupt handler
$68	autoInt2	pointer to level-2 interrupt handler
$6C	autoInt3	pointer to level-3 interrupt handler
$70	autoInt4	pointer to level-4 interrupt handler
$74	autoInt5	pointer to level-5 interrupt handler
$78	autoInt6	pointer to level-6 interrupt handler
$7C	autoInt7	pointer to level-7 interrupt handler

Figure 6. Primary Interrupt Vector Table

Execution jumps to the interrupt handler at the address specified in the table. The interrupt handler then must identify and service the interrupt. Then, it must restore the processor priority, status register, and program counter to the values they contained before the interrupt occurred.

Level-1 (VIA) Interrupts

Level-1 interrupts are generated by the VIA. You'll need to read the Synertek manual describing the VIA to use most of the information provided in this section. The level-1 interrupt handler determines the source of the interrupt (via the VIA's IFR and IER registers) and then uses a table of secondary vectors in the system communication area to determine which interrupt handler to call (Figure 7).

byte 0	one-second interrupt	VIA's CA2 control line
4	vertical-retrace interrupt	VIA's CA1 control line
8	shift-register interrupt	VIA's shift register
12	not used	
16	not used	
20	T2 timer: Disk Driver	VIA's timer 2
24	T1 timer: Sound Driver	VIA's timer 1
28	not used	

Figure 7. Level-1 Secondary Interrupt Vector Table

The level-1 secondary interrupt vector table begins at the address of the global variable lvl1DT. Each vector in the table points to the interrupt handler for a different source of interrupt. The interrupts are handled in order of their entry in the table, and only one interrupt handler is called per level-1 interrupt (even if two or more sources are interrupting). This allows the level-1 interrupt handler to be reentrant, and interrupt handlers should lower the processor priority as soon as possible in order to enable other pending interrupts to be processed.

One-second interrupts occur every second, and simply update the system global variable time (explained in the Operating System Utilities manual *** doesn't yet exist ***) and invert menu items that are chosen. Vertical retrace interrupts are generated once every vertical retrace interval; control is passed to the Vertical Retrace Manager, which updates the global variable named ticks, handles changes in the state of the cursor, keyboard, and mouse button, and executes tasks installed in the vertical retrace queue.

The shift-register interrupt is used by the Keyboard/Mouse Handler. Whenever the Disk Driver or Sound Driver isn't being used, you can use the T1 and T2 timers for your own needs.

If the cumulative elapsed time for all tasks during a vertical retrace interrupt exceeds 16 milliseconds (one video frame), the vertical retrace interrupt may itself be interrupted by another vertical retrace interrupt. In this case, the second vertical retrace interrupt is ignored.

The base address of the VIA (stored in the global variable VIA) is passed to each interrupt handler in A1.

Level-2 (SCC) Interrupts

Level-2 interrupts are generated by the SCC. You'll need to read the Zilog manual describing the SCC to effectively use the information provided in this section. The level-2 interrupt handler determines the source of the interrupt, and then uses a table of secondary vectors in the system communication area to determine which interrupt handler to call (Figure 8).

byte 0	channel B transmit buffer empty	
4	channel B external/status change	mouse vertical
8	channel B receive character available	
12	channel B special receive condition	
16	channel A transmit buffer empty	
20	channel A external/status change	mouse horizontal
24	channel A receive character available	
28	channel A special receive condition	

Figure 8. Level-2 Secondary Interrupt Vector Table

The level-2 secondary interrupt vector table begins at the address of the global variable lvl2DT. Each vector in the table points to the interrupt handler for a different source of interrupt. The interrupts are handled according to the following fixed priority:

channel A receive character available and special receive
channel A transmit buffer empty
channel A external/status change
channel B receive character available and special receive
channel B transmit buffer empty
channel B external/status change

Only one interrupt handler is called per level-2 interrupt (even if two or more sources are interrupting). This allows the level-2 interrupt handler to be reentrant, and interrupt handlers should lower the processor priority as soon as possible in order to enable other pending interrupts to be processed.

External/status interrupts pass through a tertiary vector table in the system communication area to determine which interrupt handler to call (Figure 9).

byte 0	channel B communications interrupt
4	mouse vertical interrupt
8	channel A communications interrupt
12	mouse horizontal interrupt

Figure 9. Level-2 External/Status Interrupt Vector Table

The external/status interrupt vector table begins at the address of the global variable extStsDT. Each vector in the table points to the interrupt handler for a different source of interrupt. Communications interrupts (break/abort, for example) are always handled before mouse interrupts.

When a level-2 interrupt handler is called, DØ contains the address of the SCC read register Ø (external/status interrupts only), and D1 contains the bits of read register Ø that have changed since the last external/status interrupt. AØ points to the SCC channel A or channel B control read address and A1 points to SCC channel A or channel B control write address, depending on which channel is interrupting. The SCC's data read address and data write address are located four bytes beyond AØ and A1, respectively. The following global constants can be used to refer to these locations:

Global constant	Value	Refers to
bCtl	Ø	Offset for channel B control
aCtl	2	Offset for channel A control
bData	4	Offset for channel B data
aData	6	Offset for channel A data

Writing Your Own Interrupt Handlers

You can write your own interrupt handlers to replace any of the standard interrupt handlers just described. Be sure to place a vector that points to your interrupt handler in one of the vector tables.

Both the level-1 and level-2 interrupt handlers preserve AØ through A3 and DØ through D3. Every interrupt handler (except for external/status interrupt handlers) is responsible for clearing the source of the interrupt, and for saving and restoring any additional registers used. Interrupt handlers should return directly via an RTS instruction, unless the interrupt is handled immediately, in which case they should jump (via JMP) to the IODone routine.

SUMMARY OF THE DEVICE MANAGER

Constants

```
{ Values for posMode and ioPosMode }

CONST fsAtMark    = Ø; {at current position of mark }
                       { (ioPosOffset ignored)}
      fsFromStart = 1; {offset relative to beginning of file}
      fsFromLEOF  = 2; {offset relative to logical end-of-file}
      fsFromMark  = 3; {offset relative to current mark}

{ Values for requesting read/write access }

      fsCurPerm  = Ø; {whatever is currently allowed}
      fsRdPerm   = 1; {request to read only}
      fsWrPerm   = 2; {request to write only}
      fsRdWrPerm = 3; {request to read and write}
```

Data Types

```
TYPE ParmBlkPtr    = ^ParamBlockRec;

     ParamBlkType  = (ioParam, fileParam, volumeParam, cntrlParam);

     ParamBlockRec = RECORD
        qLink:        QElemPtr;  {next queue entry}
        qType:        INTEGER;   {queue type}
        ioTrap:       INTEGER;   {routine trap}
        ioCmdAddr:    Ptr;       {routine address}
        ioCompletion: ProcPtr;   {completion routine}
        ioResult:     OSErr;     {result code}
        ioNamePtr:    StringPtr; {driver name}
        ioVRefNum:    INTEGER;   {used by Disk Driver}
     CASE ParamBlkType OF
      ioParam:
       (ioRefNum:    INTEGER;    {driver reference number}
        ioVersNum:   SignedByte; {not used}
        ioPermssn:   SignedByte; {read/write permission}
        ioMisc:      Ptr;        {not used}
        ioBuffer:    Ptr;        {data buffer}
        ioReqCount:  LongInt;    {requested number of bytes}
        ioActCount:  LongInt;    {actual number of bytes}
        ioPosMode:   INTEGER;    {type of positioning operation}
        ioPosOffset: LongInt);   {size of positioning offset}
      fileParam:
        . . . {used by File Manager}
      volumeParam:
        . . . {used by File Manager}
      cntrlParam:
       (csCode:  INTEGER;              {type of Control or Status call}
        csParam: ARRAY[Ø..Ø] OF Byte); {control or status information}
     END;
```

High-Level Routines

```
FUNCTION OpenDriver  (name: Str255; VAR refNum: INTEGER) : OSErr;
FUNCTION CloseDriver (refNum: INTEGER) : OSErr;
FUNCTION FSRead  (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr)
                 : OSErr;
FUNCTION FSWrite (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr)
                 : OSErr;
FUNCTION Control (refNum: INTEGER; csCode: INTEGER; csParam: Ptr) :
                 OSErr;
FUNCTION Status  (refNum: INTEGER; csCode: INTEGER; csParam: Ptr) :
                 OSErr;
FUNCTION KillIO  (refNum: INTEGER) : OSErr;
```

Low-Level Routines

```
FUNCTION PBOpen    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBClose   (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBRead    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBWrite   (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBControl (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBStatus  (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBKillIO  (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Accessing a Driver's I/O Queue

```
FUNCTION GetDCtlQHdr (refNum: INTEGER) : QHdrPtr;
```

Assembly-Language Information

Constants

```
; I/O queue type

ioQType        .EQU      2       ;I/O request queue entry type

; Driver flags

dReadEnable    .EQU   Ø    ;set if driver can respond to Read calls
dWriteEnable   .EQU   1    ;set if driver can respond to Write calls
dCtlEnable     .EQU   2    ;set if driver can respond to Control calls
dStatEnable    .EQU   3    ;set if driver can respond to Status calls
dNeedGoodBye   .EQU   4    ;set if driver needs to be called before the
```

```
                                ; application heap is reinitialized
dNeedTime       .EQU    5       ;set if driver needs time for performing a
                                ; periodic action
dNeedLock       .EQU    6       ;set if driver will be locked in memory as
                                ; soon as it's opened (always set for
                                ; ROM drivers)

; Device control entry flags

dOpened         .EQU    5       ;set if driver is open
dRAMBased       .EQU    6       ;set if driver is RAM-based
drvrActive      .EQU    7       ;set if driver is currently executing

; Trap words for Device Manager calls

aRdCmd      .EQU    2       ;Read call (trap $AØØ2)
aWrCmd      .EQU    3       ;Write call (trap $AØØ3)
aCtlCmd     .EQU    4       ;Control call (trap $AØØ4)
aStsCmd     .EQU    5       ;Status call (trap $AØØ5)

; Offsets for SCC

bCtl      .EQU    Ø     ;Offset for SCC channel B control
aCtl      .EQU    2     ;Offset for SCC channel A control
bData     .EQU    4     ;Offset for SCC channel B data
aData     .EQU    6     ;Offset for SCC channel A data
```

Standard Parameter Block Data Structure

qLink	Next queue entry
qType	Queue type
ioTrap	Routine trap
ioCmdAddr	Routine address
ioCompletion	Completion routine
ioResult	Result code
ioFileName	File name (and possibly volume name too)
ioVNPtr	Volume name
ioVRefNum	Volume reference number
ioDrvNum	Drive number

Control and Status Parameter Block Data Structure

csCode	Type of Control or Status call
csParam	Parameters for Control or Status call

I/O Parameter Block Data Structure

ioRefNum	Driver reference number
ioFileType	Not used
ioPermssn	Open permission
ioBuffer	Data buffer
ioReqCount	Requested number of bytes
ioActCount	Actual number of bytes
ioPosMode	Type of positioning operation
ioPosOffset	Size of positioning offset

Driver Structure

drvrFlags	Flags
drvrDelay	Number of ticks between periodic actions
drvrEMask	Desk accessory event mask
drvrMenu	Menu ID of menu associated with driver
drvrOpen	Offset to open routine
drvrPrime	Offset to prime routine
drvrCtl	Offset to control routine
drvrStatus	Offset to status routine
drvrClose	Offset to close routine
drvrName	Length and characters of driver name

Device Control Entry Data Structure

dCtlDriver	Pointer to ROM driver or handle to RAM driver
dCtlFlags	Flags
dCtlQueue	Low-order byte is driver's version number
dCtlQHead	Pointer to first entry in driver's I/O queue
dCtlTail	Pointer to last entry in driver's I/O queue
dCtlPosition	Byte position used by Read and Write calls
dCtlStorage	Handle to RAM driver's private storage
dCtlRefNum	Driver's reference number
dCtlCurTicks	Used internally by Device Manager
dCtlWindow	Pointer to driver's window record (if any)
dCtlDelay	Number of ticks between periodic actions
dCtlEMask	Desk accessory event mask
dCtlMenu	Menu ID of menu associated with driver

Primary Interrupt Vector Table

autoInt1	Pointer to level-1 interrupt handler
autoInt2	Pointer to level-2 interrupt handler
autoInt3	Pointer to level-3 interrupt handler
autoInt4	Pointer to level-4 interrupt handler
autoInt5	Pointer to level-5 interrupt handler
autoInt6	Pointer to level-6 interrupt handler
autoInt7	Pointer to level-7 interrupt handler

I/O Parameter Block Data Structure

ioRefNum	Driver reference number

Macro Names

Routine name	Macro name
PBRead	_Read
PBWrite	_Write
PBControl	_Control
PBStatus	_Status
PBKillIO	_KillIO

Routines for Writing Drivers

Routine	Jump vector
Fetch	jFetch
Stash	jStash
IODone	jIODone

Variables

Name	Size	Contents
uTableBase	4 bytes	Pointer to unit table
unitNtryCnt	2 bytes	Maximum number of entries in unit table
lvl1DT	4 bytes	Beginning of level-1 secondary interrupt vector table
lvl2DT	4 bytes	Beginning of level-2 secondary interrupt vector table
extStsDT	4 bytes	Beginning of external/status interrupt vector table
sccRBase	4 bytes	SCC base read address
sccWBase	4 bytes	SCC base write address
VIA	4 bytes	VIA base address

Result Codes

Name	Value	Meaning
abortErr	-27	I/O request aborted by KillIO
badUnitErr	-21	Reference number doesn't match unit table
controlErr	-17	Driver can't respond to this Control call
dInstErr	-26	Couldn't find driver in resource file
dRemoveErr	-25	Tried to remove an open driver
noErr	Ø	No error
notOpenErr	-28	Driver isn't open
openErr	-23	Requested read/write permission doesn't match driver's open permission
readErr	-19	Driver can't respond to Read calls
statusErr	-18	Driver can't respond to this Status call
unitEmptyErr	-22	Reference number specifies NIL handle in unit table
writErr	-2Ø	Driver can't respond to Write calls

GLOSSARY

asynchronous execution: After calling a routine asynchronously, an application is free to perform other tasks until the routine is completed.

block device: A device that reads and writes blocks of 512 characters at a time; it can read or write any accessible block on demand.

character device: A device that reads or writes a stream of characters, one at a time: it can neither skip characters nor go back to a previous character.

closed driver: A device driver that cannot be read from or written to.

close routine: The part of a driver's code that implements Device Manager Close calls.

completion routine: Any application-defined code to be executed when an asynchronous call to a Device Manager routine is completed.

control information: Information transmitted by an application to a device driver; it can typically select modes of operation, start or stop processes, enable buffers, choose protocols, and so on.

control routine: The part of a device driver's code that implements Device Manager Control and KillIO calls.

data buffer: Heap space containing information to be written to a file or driver from an application, or read from a file or driver to an application.

device: A part of the Macintosh or a piece of external equipment, that can transfer information into or out of the Macintosh.

device control entry: A 4Ø-byte relocatable block of heap space that tells the Device Manager the location of a driver's routines, the location of a driver's I/O queue, and other information.

device driver: A program that exchanges information between an application and a device.

driver name: A sequence of up to 254 printing characters used to refer to an open device driver; driver names always begin with a period (.).

driver reference number: A number that uniquely identifies an individual device driver.

exception: An error or abnormal condition detected by the processor in the course of program execution.

interrupt: An exception that's signaled to the processor by a device, to notify the processor of a change in condition of the device, such as the completion of an I/O request.

interrupt handler: A routine that services interrupts.

interrupt priority level: A number identifying the importance of the interrupt. It indicates which device is interrupting, and which interrupt handler should be executed.

interrupt vector: A pointer to an interrupt handler.

I/O queue: A queue containing the parameter blocks of all I/O requests for one driver.

I/O request: A request for input from or output to a file or device driver; caused by calling a File Manager or Device Manager routine asynchronously.

open driver: A driver that can be read from and written to.

open routine: The part of a device driver's code that implements Device Manager Open calls.

parameter block: An area of heap space used to transfer information between applications and the Device Manager.

prime routine: The part of a device driver's code that implements Device Manager Read and Write calls.

processor priority: Bits 8, 9, and 1Ø of the MC68ØØØ's status register, that indicate which interrupts will be processed and which will be ignored.

status information: Information transmitted to an application by a device driver; it may indicate the current mode of operation, the readiness of the device, the occurrence of errors, and so on.

status routine: The part of a device driver's code that implements Device Manager Status calls.

synchronous execution: After calling a routine synchronously, an application cannot continue execution until the routine is completed.

unit number: The number of each device driver's entry in the unit table.

unit table: A 128-byte nonrelocatable block containing a handle to the device control entry for each device driver.

vector: A pointer.

vector table: A table of vectors in the system communication area.

MACINTOSH USER EDUCATION

The Disk Driver: A Programmer's Guide /DRIVER/DISK

See Also: Inside Macintosh: A Road Map
Macintosh Memory Management: An Introduction
Programming Macintosh Applications in Assembly Language
The Memory Manager: A Programmer's Guide
The Device Manager: A Programmer's Guide
The File Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
Macintosh Packages: A Programmer's Guide

Modification History:	First Draft	Bradley Hacker	9/18/84

ABSTRACT

The Disk Driver is a Macintosh device driver used for storing and retrieving information on Macintosh 3 1/2-inch disk drives. This manual describes the Disk Driver in detail. It's intended for programmers who wish to access Macintosh drives directly, bypassing the File Manager.

TABLE OF CONTENTS

ABOUT THIS MANUAL

The Disk Driver is a Macintosh device driver used for storing and retrieving information on Macintosh 3 1/2-inch disk drives. This manual describes the Disk Driver in detail. It's intended for programmers who wish to access Macintosh drives directly, bypassing the File Manager. *** Eventually it will become part of the comprehensive Inside Macintosh manual. ***

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal and the information in the following manuals:

- Inside Macintosh: A Road Map
- Macintosh Memory Management: An Introduction
- Programming Macintosh Applications in Assembly Language, if you're using asssembly language

You should also be familiar with the following:

- the Memory Manager
- files and disk drives, as described in the File Manager manual
- interrupts and the use of devices and device drivers, as described in the Device Manager manual
- events, as discussed in the Toolbox and Operating System Event Manager manuals *** (The Operating System Event Manager manual doesn't yet exist) ***

ABOUT THE DISK DRIVER

The Disk Driver is a standard Macintosh device driver in ROM. It allows Macintosh applications to read from disks, write to disks, and eject disks. The Disk Driver cannot format disks; this task is accomplished by the Disk Initialization Package.

Information on disks is stored in 512-byte sectors. There are 800 sectors on one 400K-byte Macintosh disk. Each sector consists of an address mark that contains information used by the Disk Driver to determine the position of the sector on the disk, and a data mark that primarily contains data stored in that sector.

Consecutive sectors on a disk are grouped into tracks. There are 80 tracks on one 400K-byte Macintosh disk. Track 0 is the outermost and track 79 is the innermost. Each track corresponds to a ring of constant radius around the disk. *** Future double-sided disks may contain 1600 sectors. ***

Macintosh disks are formatted in a manner that allows a more efficient use of disk space than most microcomputer formatting schemes: The tracks are divided into five groups of 16 tracks each, and each group of tracks is accessed at a different speed from the other groups. (Those at the edge of the disk are accessed at slower disk speeds than those toward the center.)

Each group of tracks contains a different number of sectors:

Tracks	Sectors per track	Sectors
Ø-15	12	Ø-191
16-31	11	192-367
32-47	1Ø	368-527
48-63	9	528-671
64-79	8	672-799

An application can read or write data in **whole** disk sectors only. The application must specify the data to be read or written in 512-byte multiples, and the Disk Driver automatically calculates which sector to access. The application specifies where on the disk the data should be read or written by providing a positioning operation and a positioning offset. Data can be read from or written to the disk:

- at the current sector on the disk (the sector following the last sector read or written)

- from a position relative to the current sector on the disk

- from a position relative to the beginning of first sector on the disk

The following constants are used to specify the positioning operation:

```
        CONST currPos = Ø;  {at current sector}
              absPos  = 1;  {relative to first sector}
              relPos  = 3;  {relative to current sector}
```

If the positioning operation is relative to a sector (absPos or relPos), the relative offset from that sector must be given as a 512-byte multiple.

Whenever the Disk Driver reads a sector from a disk, it places the sector's 12 bytes of file tags at a special location in low memory called the file tags buffer (the remaining 512 bytes in the sector are passed on to the File Manager). Each time one sector's file tags are written there, the previous file tags are overwritten.

Conversely, whenever the Disk Driver writes a sector on a disk, it takes the 12 bytes in the file tags buffer and writes them on the disk.

Assembly-language note: The low memory location TagData + 2 contains the file tags buffer.

The Disk Driver disables interrupts for 12 to 24 milliseconds during disk accesses. During this interval it stores any serial data received via the modem port and later passes the data to the Serial Driver. This allows the modem port to be used simultaneously with disk accesses without fear of hardware overrun errors.

USING THE DISK DRIVER

This section introduces you to the Disk Driver 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 Disk Driver is opened automatically when the system starts up. It allocates space in the system heap for variables, installs entries in the drive queue for each drive that's attached to the Macintosh, and installs an application task into the vertical retrace queue. The Disk Driver's name is '.Sony', and its reference number is -5.

To write data onto a disk, make a Device Manager Write call. You must pass the following parameters:

- The driver reference number -5.
- The drive number 1 (internal drive) or 2 (external drive).
- A positioning operation indicating where on the disk the information should be written.
- A positioning offset that's a multiple of 512 bytes.
- A buffer that contains the data you want to write.
- The number of bytes (in multiples of 512) that you want to write.

The Disk Driver's prime routine returns one of the following result codes to the Write routine:

noErr	No error
badBtSlpErr	Bad address mark
badCksmErr	Bad address mark
badDBtSlp	Bad data mark
badDCksum	Bad data mark
cantStepErr	Hardware error
initIWMErr	Hardware error
noAdrMkErr	Can't find an address mark

noDriveErr	Drive isn't connected
noDtaMkErr	Can't find data mark
noNybErr	Disk is probably blank
nsDrvErr	No such drive
offLinErr	No disk in drive
paramErr	Bad positioning information
sectNFErr	Can't find sector
seekErr	Hardware error
spdAdjErr	Hardware error
tk∅BadErr	Hardware error
twoSideErr	Tried to read side 2 of a disk in a single-sided drive
wPrErr	Disk is locked

To read data from a disk, make a Device Manager Read call. You must pass the following parameters:

- The driver reference number -5.

- The drive number 1 (internal drive) or 2 (external drive).

- A positioning operation indicating where on the disk the information should be read from. If the following constant is added to the positioning operation, the Disk Driver will verify that the data written to the disk exactly matches the data in memory (the result code dataVerErr will be returned if any of the data doesn't match):

```
CONST rdVerify = 64;  {read-verify mode}
```

- A positioning offset that's a multiple of 512 bytes.

- A buffer to receive the data that's read.

- The number of bytes (in multiples of 512) that you want to read.

The Disk Driver's prime routine returns one of the following result codes to the Read routine:

noErr	No error
badBtSlpErr	Bad address mark
badCksmErr	Bad address mark
badDBtSlp	Bad data mark
badDCksum	Bad data mark
cantStepErr	Hardware error
dataVerErr	Read-verify failed
initIWMErr	Hardware error
noAdrMkErr	Can't find an address mark
noDriveErr	Drive isn't connected
noDtaMkErr	Can't find data mark
noNybErr	Disk is probably blank
nsDrvErr	No such drive
offLinErr	No disk in drive
tk∅BadErr	Hardware error

paramErr	Bad positioning information
sectNFErr	Can't find sector
seekErr	Hardware error
spdAdjErr	Hardware error
twoSideErr	Tried to read side 2 of a disk in a single-sided drive

The Disk Driver can read and write sectors in any order, and therefore operates faster on one large data request than it would on a series of equivalent but smaller data requests.

There are three different calls you can make to the Disk Driver's control routine:

- KillIO causes all current I/O requests to be aborted. KillIO is a Device Manager call.

- SetTagBuffer specifies the information to be used in the file tags buffer.

- DiskEject ejects a disk from a drive.

An application using the File Manager should always unmount the volume in a drive before ejecting the disk.

You can make one call, DriveStatus, to the Disk Driver's status routine, to learn about the state of the driver.

An application can bypass the implicit mounting of volumes done by the File Manager by calling the Operating System Event Manager function GetOSEvent and looking for disk-inserted events. Once the volume has been inserted in the drive it can be read from normally.

DISK DRIVER ROUTINES

This section describes the Disk Driver routines. They return an integer result code of type OSErr; each routine description lists all of the applicable result codes.

> Assembly-language note: There are no trap macros for these routines, but assembly-language programmers can make equivalent Control and Status calls, as indicated in the routine descriptions.

FUNCTION DiskEject (drvNum: INTEGER) : OSErr;

Assembly-language note: DiskEject is equivalent to a Control call with csCode equivalent to the global constant ejectCode.

DiskEject ejects the disk from the internal drive if drvNum is 1, or from the external drive if drvNum is 2.

Result codes		
	noErr	No error
	nsDrvErr	No such drive

FUNCTION SetTagBuffer (buffPtr: Ptr) : OSErr;

Assembly-language note: SetTagBuffer is equivalent to a Control call with csCode = 8.

An application can change the information used in the file tags buffer by calling SetTagBuffer. The buffPtr parameter points to a buffer that contains the information to be used. If buffPtr is NIL, the information in the file tags buffer isn't changed.

If buffPtr isn't NIL, every time the Disk Driver reads a sector from the disk, it stores the file tags in the file tags buffer and in the buffer pointed to by buffPtr. Every time the Disk Driver writes a sector onto the disk, it reads 12 bytes from the buffer pointed to by buffPtr, places them in the file tags buffer, and then writes them onto the disk.

The contents of the buffer pointed to by buffPtr are overwritten at the end of every read request (which can be composed of a number of sectors) instead of at the end of every sector. Each read request places 12 bytes in the buffer for **each** sector, always beginning at the start of the buffer. This way an application can examine the file tags for a number of sequentially read sectors. If a read request is composed of a number of sectors, the Disk Driver reads 12 bytes from the buffer for each sector. For example, for a read request of five sectors, the Disk Driver will read 6Ø bytes from the buffer.

Assembly-language note: An assembly-language program can change the information used in the file tags buffer by storing a pointer to the buffer containing the information in the global variable TagBufPtr. If TagBufPtr is Ø, the information in the file tags buffer isn't changed.

<u>Result codes</u>	noErr	No error

FUNCTION DriveStatus (drvNum: INTEGER; VAR status: DrvSts) : OSErr;

<u>Assembly-language note</u>: DriveStatus is equivalent to a Status call with csCode equivalent to the global constant drvStsCode.

DriveStatus returns information about the internal drive if drvNum is 1, or about the external drive if drvNum is 2. The information is returned in a record of type DrvSts:

```
TYPE DrvSts = RECORD
                track:        INTEGER;    {current track}
                writeProt:    SignedByte; {bit 7=1 if volume is locked}
                diskInPlace:  SignedByte; {disk in place}
                installed:    SignedByte; {drive installed}
                sides:        SignedByte; {bit 7=Ø if single-sided drive}
                qLink:        QElemPtr;   {next queue entry}
                qType:        INTEGER;    {not used}
                dqDrive:      INTEGER;    {drive number}
                dqRefNum:     INTEGER;    {driver reference number}
                dqFSID:       INTEGER;    {file-system identifier}
                twoSideFmt:   SignedByte; {-1 if two-sided disk}
                needsFlush:   SignedByte; {reserved}
                diskErrs:     INTEGER     {error count}
              END;
```

The diskInPlace field is Ø if there's no disk in the drive, 1 or 2 if there is a disk in the drive, or -4 to -1 if the disk was ejected in the last 1.5 seconds. The installed field is 1 if the drive is connected to the Macintosh, Ø if the drive might be connected to the Macintosh, and -1 if the drive isn't installed. The value of twoSideFmt is valid only when diskInPlace = 2. The value of diskErrs is incremented every time an error occurs internally within the Disk Driver.

<u>Result codes</u>	noErr	No error
	nsDrvErr	No such drive

ASSEMBLY-LANGUAGE EXAMPLE

The following assembly-language example ejects the disk in drive 1:

```
MyEject   MOVEQ     #<ioQElSize/2>-1,DØ       ;prepare an I/O
@1        CLR.W     -(SP)                     ; request block
          DBRA      DØ,@1                     ; on the stack
          MOVE.L    SP,AØ                     ;AØ points to it
          MOVE.W    #dskRfN,ioRefNum(AØ)      ;driver refNum
          MOVE.W    #1,ioDrvNum(AØ)           ;internal drive
          MOVE.W    #ejectCode,csCode(AØ)     ;eject control code
          _Eject                              ;synchronous call
          ADD       #ioQElSize,SP             ;clean up stack
```

To asynchronously read sector 4 from the disk in drive 1, you would do the following:

```
MyRead    MOVEQ     #<ioQElSize/2>-1,DØ       ;prepare an I/O
@1        CLR.W     -(SP)                     ; request block
          DBRA      DØ,@1                     ; on the stack
          MOVE.L    SP,AØ                     ;AØ points to it
          MOVE.W    #dskRfN,ioRefNum(AØ)      ;driver refNum
          MOVE.W    #1,ioDrvNum(AØ)           ;internal drive
          MOVE.W    #1,ioPosMode(AØ)          ;absolute positioning
          MOVE.L    #<512*4>,ioPosOffset(AØ)  ;sector 4

          MOVE.L    #512,ioByteCount(AØ)      ;read one sector
          LEA       MyBuffer,A1
          MOVE.L    A1,ioBuffer(AØ)           ;buffer address
          _Read     ,ASYNC                    ;read data

; Do any other processing here.  Then, when the sector is needed:

@2        MOVE.W    ioResult(AØ),DØ           ;wait for completion
          BGT.S     @2
          ADD       #ioQElSize,SP             ;clean up stack

MyBuffer  .BLOCK    512,Ø
```

SUMMARY OF THE DISK DRIVER

Constants

```
CONST { Positioning information }

      currPos  = Ø;   {at current sector}
      absPos   = 1;   {relative to first sector}
      relPos   = 3;   {relative to current sector}
      rdVerify = 64;  {read-verify mode}
```

Data Types

```
TYPE DrvSts = RECORD
                track:       INTEGER;     {current track}
                writeProt:   SignedByte;  {bit 7=1 if volume is locked}
                diskInPlace: SignedByte;  {disk in place}
                installed:   SignedByte;  {drive installed}
                sides:       SignedByte;  {bit 7=Ø if single-sided drive}
                qLink:       QElemPtr;    {next queue entry}
                qType:       INTEGER;     {not used}
                dQDrive:     INTEGER;     {drive number}
                dQRefNum:    INTEGER;     {driver reference number}
                dQFSID:      INTEGER;     {file-system identifier}
                twoSideFmt:  SignedByte;  {-1 if two-sided disk}
                needsFlush:  SignedByte;  {reserved}
                diskErrs:    INTEGER      {error count}
             END;
```

Disk Driver Routines [No trap macro]

```
FUNCTION DiskEject    (drvNum: INTEGER) : OSErr;
FUNCTION SetTagBuffer (buffPtr: Ptr) : OSErr;
FUNCTION DriveStatus  (drvNum: INTEGER; VAR status: DrvSts) : OSErr;
```

Assembly-Language Information

Constants

```
ejectCode    .EQU    7
drvStsCode   .EQU    8
```

Structure of Status Information

track	Current track
writeProt	Bit 7=1 if volume is locked
diskInPlace	Disk in place
installed	Drive installed
sides	Bit 7=Ø if single-sided drive
dQEl	Drive queue entry
twoSideFmt	-1 if two-sided disk
diskErrs	Error count

Variables

Name	Size	Contents
TagData + 2	12 bytes	Default file tags buffer
TagBufPtr	4 bytes	Pointer to information for file tags buffer

Equivalent Device Manager Calls

Pascal routine	Call	CSCode
DiskEject	Control	ejectCode
SetTagBuffer	Control	8
DriveStatus	Status	drvStsCode

Result Codes

These values are available as predefined constants in both Pascal and assembly language.

Name	Value	Meaning
badBtSlpErr	-7Ø	Bad address mark
badCksmErr	-69	Bad address mark
badDBtSlp	-73	Bad data mark
badDCksum	-72	Bad data mark
cantStepErr	-75	Hardware error
dataVerErr	-68	Read-verify failed
initIWMErr	-77	Hardware error
noAdrMkErr	-67	Can't find an address mark
noDriveErr	-64	Drive isn't connected
noDtaMkErr	-71	Can't find data mark
noErr	Ø	No error
noNybErr	-66	Disk is probably blank
nsDrvErr	-56	No such drive
offLinErr	-65	No disk in drive
paramErr	-5Ø	Bad positioning information
sectNFErr	-81	Can't find sector
seekErr	-8Ø	Hardware error
spdAdjErr	-79	Hardware error
tkØBadErr	-76	Hardware error

twoSideErr	-78	Tried to read side 2 of a disk in a single-sided drive
wPrErr	-44	Disk is locked

GLOSSARY

address mark: In a sector, information that's used internally by the Disk Driver, including information it uses to determine the position of the sector on the disk.

data mark: In a sector, information that primarily contains data from an application.

file tags buffer: A location in memory where file tags are read from and written to.

sector: Disk space composed of 512 consecutive bytes of standard information and 12 bytes of file tags.

track: Disk space composed of 8 to 12 consecutive sectors. A track corresponds to one ring of constant radius around the disk.

MACINTOSH USER EDUCATION

The Sound Driver: A Programmer's Guide /SNDRVR/SOUND

See Also: Programming Macintosh Applications in Assembly Language
Inside Macintosh: A Road Map
Macintosh Memory Management: An Introduction
The Memory Manager: A Programmer's Guide
The Device Manager: A Programmer's Guide

Modification History:			
	First Draft	Bradley Hacker	7/16/84
	Second Draft	Bradley Hacker & Caroline Rose	11/15/84

ABSTRACT

The Sound Driver is a Macintosh device driver for handling sound and music generation in a Macintosh application. This manual describes the Sound Driver in detail.

Summary of significant changes and additions since last draft:

- The definition of the term "amplitude" has been corrected, and the term "magnitude" added (page 4).

- The note concerning how to call StopSound from assembly language has been corrected (page 15).

- The equivalent assembly-language instructions for the SetSoundVol procedure have been removed; in assembly-language, you can just call this Pascal procedure from your program (page 16).

- The summary now includes numbers not only for Ptolemy's diatonic scale, but also for an equal-tempered scale (page 2∅).

TABLE OF CONTENTS

ABOUT THIS MANUAL

The Sound Driver is a Macintosh device driver for handling sound and music generation in a Macintosh application. This manual describes the Sound Driver in detail. *** Eventually it will become part of the comprehensive Inside Macintosh manual. ***

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal and the information in the following manuals:

- Inside Macintosh: A Road Map

- Macintosh Memory Management: An Introduction

- Programming Macintosh Applications in Assembly Language, if you're using assembly language

You should also be familiar with the following:

- the Memory Manager

- devices and device drivers, as described in the Device Manager Manual

ABOUT THE SOUND DRIVER

The Sound Driver is a standard Macintosh device driver used to synthesize sound. You can generate sound characterized by any kind of waveform by using the three different sound synthesizers in the Sound Driver:

- The four-tone synthesizer is used to make simple harmonic tones, with up to four "voices" producing sound simultaneously; it requires about 5Ø% of the microprocessor's attention during any given time interval.

- The square-wave synthesizer is used to produce less harmonic sounds such as beeps, and requires about 2% of the processor's time.

- The free-form synthesizer is used to make complex music and speech; it requires about 2Ø% of the processor's time.

Figure 1 depicts the waveform of a typical sound wave, and the terms used to describe it. The magnitude is the vertical distance between any given point on the wave and the horizontal line about which the wave oscillates; you can think of the magnitude as the volume level. The amplitude is the maximum magnitude of a periodic wave. The wavelength is the horizontal extent of one complete cycle of the wave. Magnitude and wavelength can be measured in any unit of distance. The period is the time elapsed during one complete cycle of a wave. The frequency is the reciprocal of the period, or the number of cycles per second (also called Hertz). The phase is some fraction of a wave cycle (measured from a fixed point on the wave).

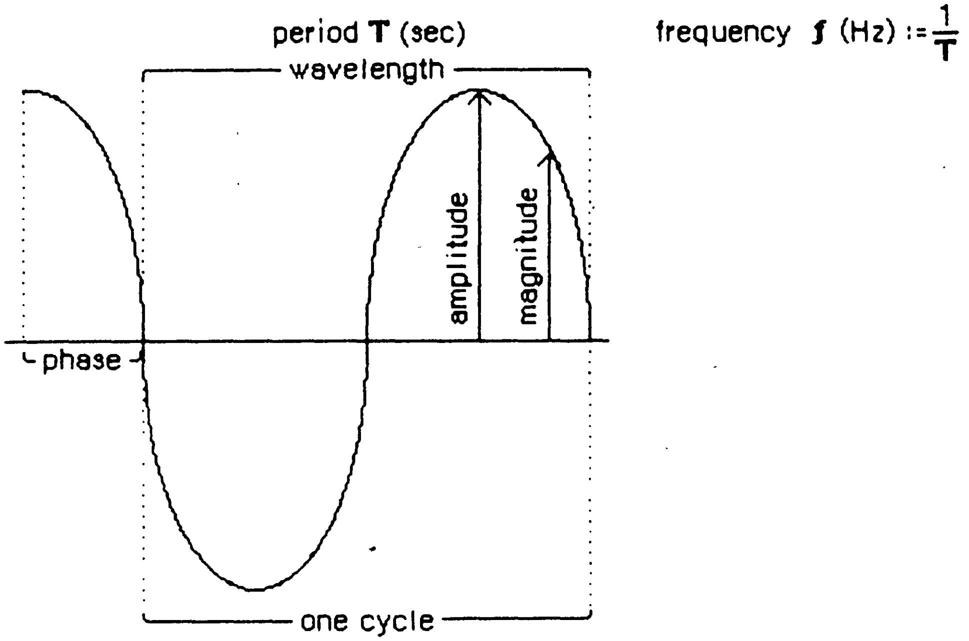
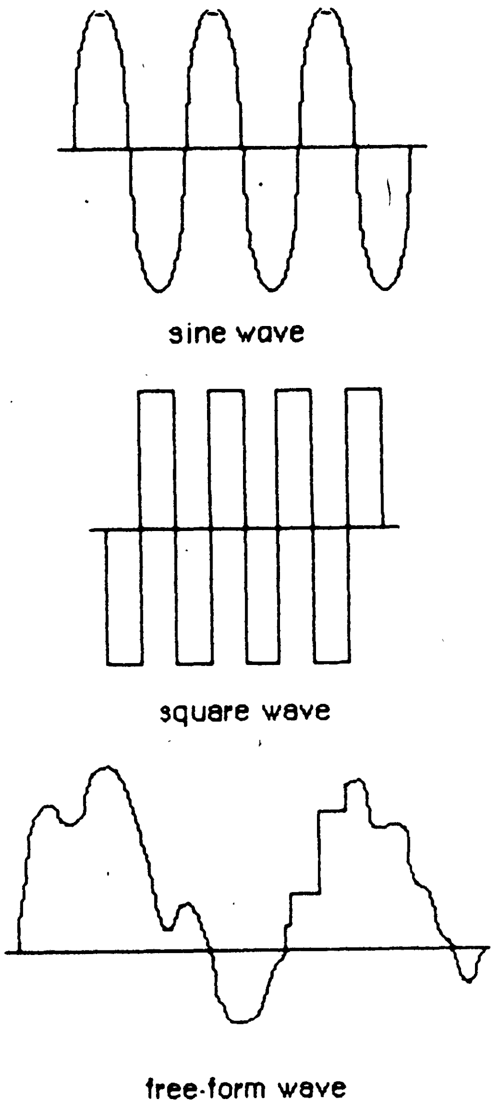


Figure 1. A Waveform

There are many different types of waveforms, three of which are depicted in Figure 2. Sine waves are generated by objects that oscillate periodically at a single frequency (such as a guitar string). Square waves are generated by objects that toggle instantly between two states at a single frequency (such as a doorbell buzzer). Free-form waves are the most common waves of all, and are generated by all objects that vibrate at rapidly changing frequencies with rapidly changing magnitudes (such as your vocal cords or the instruments of an orchestra all playing at once).


sine wave

square wave

free-form wave

Figure 2. Types of Waveforms

Figure 3 shows analog and digital representations of a waveform. The Sound Driver represents waveforms digitally, so all waveforms must be converted from their analog representation to a digital representation. The rows of numbers at the bottom of the figure are digital representations of the waveform. The numbers in the upper row are the magnitudes relative to the horizontal zero-magnitude line. The numbers in the lower row all represent the same relative magnitudes, but have been normalized to positive numbers; you'll use numbers like these when calling the Sound Driver.

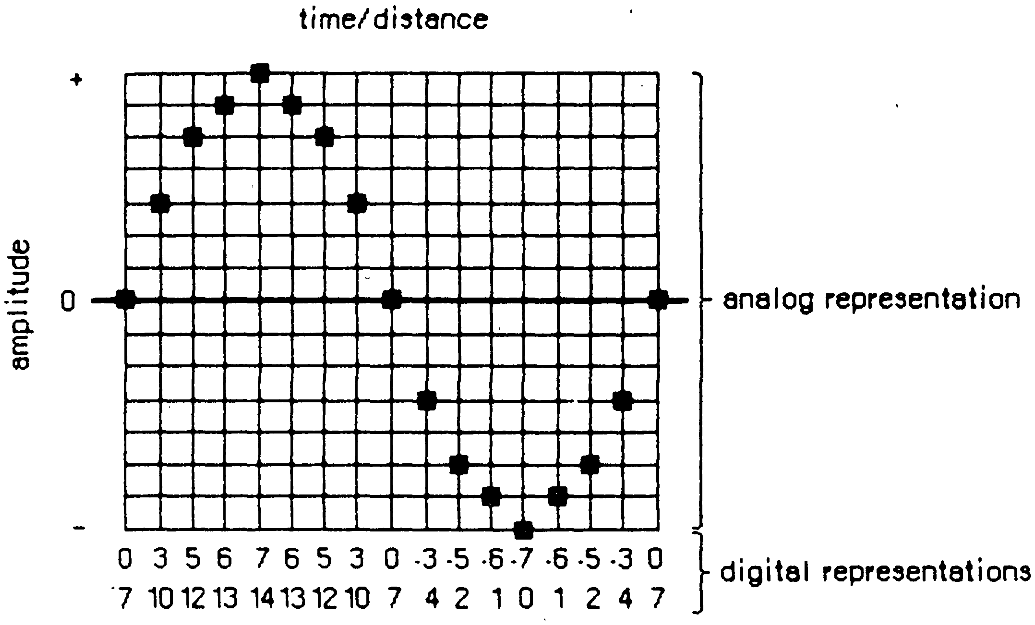


Figure 3. Analog and Digital Representations of a Waveform

A digital representation of a waveform is simply a sequence of wave magnitudes measured at fixed intervals. This sequence of magnitudes is stored in the Sound Driver as a sequence of bytes, each one of which specifies an instantaneous voltage to be sent to the speaker. The bytes are stored in a data structure called a waveform description. Since a sequence of bytes can only represent a group of numbers whose maximum and minimum values differ by less than 256, the magnitudes of your waveforms must be constrained to these same limits.

SOUND DRIVER SYNTHESIZERS

A description of the sound to be generated by a synthesizer is contained in a data structure called a synthesizer buffer. A synthesizer buffer contains the duration, pitch, phase, and waveform of the sound the synthesizer will generate. The exact structure of a synthesizer buffer differs for each type of synthesizer being used. The first word in every synthesizer buffer is an integer that identifies the synthesizer, and must be one of the following predefined constants:

```
CONST swMode = -1; {square-wave synthesizer}
      ftMode =  1; {four-tone synthesizer}
      ffMode =  Ø; {free-form synthesizer}
```

Square-Wave Synthesizer

The square-wave synthesizer is used to make sounds such as beeps. A square-wave synthesizer buffer has the following structure:

```
TYPE SWSynthRec = RECORD
                    mode:     INTEGER; {always swMode}
                    triplets: Tones    {sounds}
                  END;

     SWSynthPtr = ^SWSynthRec;

     Tones = ARRAY [Ø..5ØØØ] OF Tone;
     Tone  = RECORD
               count:     INTEGER; {frequency}
               amplitude: INTEGER; {amplitude, Ø-255}
               duration:  INTEGER  {duration in ticks}
             END;
```

Each tone triplet contains the count, amplitude, and duration of a different sound. You can store as many triplets in a synthesizer buffer as there's room for.

The count integer can range in value from Ø to 65535. The actual frequency the count corresponds to is given by the relationship:

```
frequency (Hz) = 78336Ø / count
```

A partial list of count values and corresponding frequencies for notes is given in the summary at the end of this manual.

The type Tones is declared with 5ØØ1 elements to allow you to pass up to 5ØØØ sounds (the last element must contain Ø). To be space-efficient, your application shouldn't declare a variable of type Tones; instead, you can do something like this:

```
VAR myPtr: Ptr;
    myHandle: Handle;
    mySWPtr: SWSynthPtr;
    . . .
myHandle := NewHandle(buffSize);    {allocate space for the buffer}
HLock(myHandle);                    {lock the buffer}
myPtr := myHandle^;                 {dereference the handle}
mySWPtr := SWSynthPtr(myPtr);       {coerce type to SWSynthPtr}
mySWPtr^.mode := swMode;            {identify the synthesizer}
mySWPtr^.triplets[Ø].count := 2;    {fill the buffer with values }
  . . .                             { describing the sound}
StartSound(myPtr,buffSize,POINTER(-1));  {produce the sound}
HUnlock(myHandle)                        {unlock the buffer}
```

where buffSize contains the number of bytes in the synthesizer buffer. This example dereferences handles instead of using pointers directly, to minimize the number of nonrelocatable objects on the heap.

Assembly-language note: The global variable CurPitch contains the current value of the count field.

The amplitude integer can range from Ø to 255. The duration integer specifies the number of ticks that the sound will be generated.

The list of tones ends with a triplet in which all fields are set to Ø. When the square-wave synthesizer is used, the sound specified by each triplet is generated once, and then the synthesizer stops.

Four-Tone Synthesizer

The four-tone synthesizer is used to produce harmonic sounds such as music. It can simultaneously generate four different sounds, each with its own frequency, phase, and waveform.

A four-tone synthesizer buffer has the following structure:

```
TYPE FTSynthRec = RECORD
                    mode:   INTEGER;     {always ftMode}
                    sndRec: FTSndRecPtr  {tones to play}
                  END;

     FTSynthPtr = ^FTSynthRec;
```

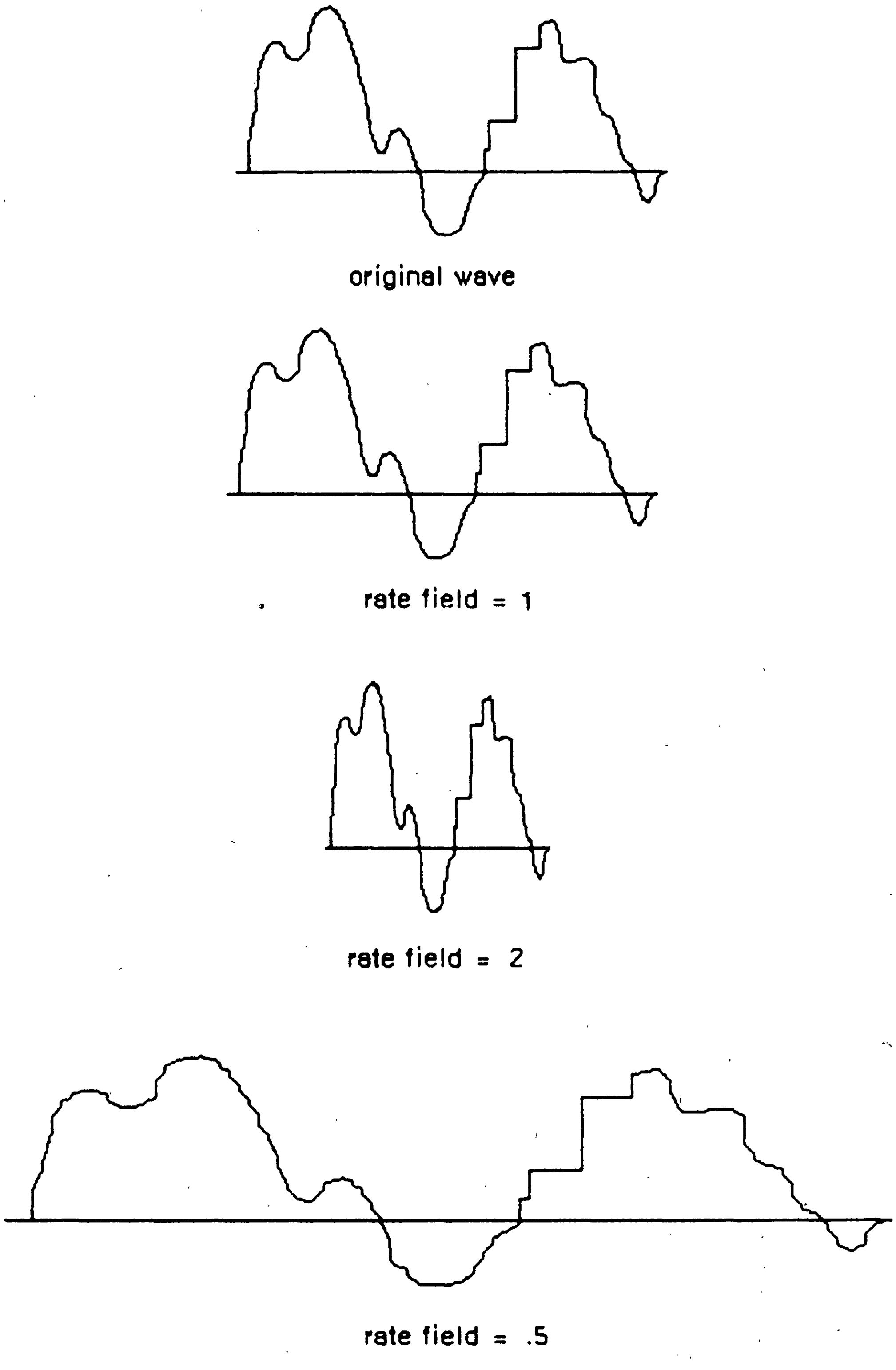
The sndRec field points to a four-tone record, which describes the four tones:

```
TYPE FTSoundRec  = RECORD
                     duration:    INTEGER; {duration in ticks}
                     sound1Rate:  Fixed;   {tone 1 cycle rate}
                     sound1Phase: LONGINT; {tone 1 byte offset}
                     sound2Rate:  Fixed;   {tone 2 cycle rate}
                     sound2Phase: LONGINT; {tone 2 byte offset}
                     sound3Rate:  Fixed;   {tone 3 cycle rate}
                     sound3Phase: LONGINT; {tone 3 byte offset}
                     sound4Rate:  Fixed;   {tone 4 cycle rate}
                     sound4Phase: LONGINT; {tone 4 byte offset}
                     sound1Wave:  WavePtr; {tone 1 waveform}
                     sound2Wave:  WavePtr; {tone 2 waveform}
                     sound3Wave:  WavePtr; {tone 3 waveform}
                     sound4Wave:  WavePtr  {tone 4 waveform}
                   END;

     FTSndRecPtr = ^FTSoundRec;

     Wave    = PACKED ARRAY [0..255] OF Byte;
     WavePtr = ^Wave;
```

Assembly-language note: The address of the four-tone record currently in use is stored in the global variable SoundPtr.

The duration integer indicates the number of ticks that the sound will be generated. Each phase long integer indicates the byte within the waveform description at which the synthesizer should begin producing sound (the first byte is byte number 0). Each rate value determines the speed at which the synthesizer cycles through the waveform, from 0 to 256.

The four-tone synthesizer creates sound by starting at the byte in the waveform description specified by the phase, and skipping rate bytes ahead every 44.93 microseconds; when the time specified by the duration integer has elapsed, the synthesizer stops. The rate field determines how the waveform will be "sized", as shown in Figure 4. The rate field is, in effect, a way of changing the frequency of the waveform, based on multiples of 44.93 microseconds. For nonperiodic waveforms, this effect is best illustrated by example: If the rate field is 1, each byte value of the waveform will produce sound for 44.93 microseconds; if the rate field is 0.1, each byte will produce sound for 449.3 microseconds; if the rate field is 5, every fifth byte in the waveform will produce sound for 44.93 microseconds.



original wave

rate field = 1

rate field = 2

rate field = .5

Figure 4. Effect of the Rate Field

If the waveform contains one wavelength, the frequency the rate corresponds to is given by:

frequency (Hz) = 1ØØØØØØ / (44.93 * 256 / rate)

The maximum rate of 256 corresponds to approximately 22.3 kHz if the waveform contains one wavelength, and a rate of Ø produces no sound. A partial list of rate values and corresponding frequencies for notes is given in the summary at the end of this manual.

Free-Form Synthesizer

The free-form synthesizer is used to synthesize complex music and speech. The sound to be produced is represented as a waveform whose complexity and length are limited only by available memory.

A free-form synthesizer buffer has the following structure:

```
TYPE FFSynthRec = RECORD
                    mode:      INTEGER;  {always ffMode}
                    count:     Fixed;    {"sizing" factor}
                    waveBytes: FreeWave  {waveform description}
                  END;

     FFSynthPtr = ^FFSynthRec;

     FreeWave   = PACKED ARRAY [Ø..3ØØØØ] OF Byte;
```

Each magnitude in the waveform description will be generated once; when the end of the waveform is reached, the synthesizer will stop. The type FreeWave is declared with 3ØØØ1 elements to allow you to pass a very long waveform. To be space-efficient, your application shouldn't declare a variable of type FreeWave; instead, you can do something like this:

```
VAR myPtr: Ptr;
    myHandle: Handle;
    myFFPtr: FFSynthPtr;
    . . .
myHandle := NewHandle(buffSize);      {allocate space for the buffer}
HLock(myHandle);                      {lock the buffer}
myPtr := myHandle^;                   {dereference the handle}
myFFPtr := FFSynthPtr(myPtr);         {coerce type to FFSynthPtr}
myFFPtr^.mode := ffMode;              {identify the synthesizer}
myFFPtr^.count := 1;                  {fill the buffer with values }
myFFPtr^.waveBytes[Ø] := Ø;           {describing the sound}
    . . .
StartSound(myPtr,buffSize,POINTER(-1));  {produce the sound}
HUnlock(myHandle)                        {unlock the buffer}
```

where buffSize contains the number of bytes in the synthesizer buffer. This example dereferences handles instead of using pointers directly, to minimize the number of nonrelocatable objects on the heap.

The free-form synthesizer creates sound by starting at the first byte in the waveform and skipping count bytes ahead every 44.93 microseconds. The count field determines how the waveform will be "sized", in a manner analogous to that of the rate field of the four-tone synthesizer, as shown in Figure 4 above.

For periodic waveforms, you can determine the frequency of the wave cycle by using the following relationship:

frequency (Hz) = 1ØØØØØØ / (44.93 * (wavelength / count))

where the wavelength is given in bytes. For example, the frequency of a wave with a 1ØØ-byte wavelength played at a count value of 2 would be approximately 445 Hz.

Assembly-language note: The address of the free-form buffer currently in use is stored in the global variable SoundBase.

USING THE SOUND DRIVER

The Sound Driver is a standard Macintosh device driver in ROM. It's automatically opened when the system starts up. Its driver name is .Sound, and its driver reference number is -4. To close or open the Sound Driver, you can use the Device Manager Close and Open functions. Because the driver is in ROM, there's really no reason to close it.

To use one of the three types of synthesizers to generate sound, you can do the following: Use the Memory Manager function NewHandle to allocate heap space for a synthesizer buffer; then lock the buffer, fill it with values describing the sound, and make a StartSound call to the Sound Driver. StartSound can be called either synchronously or asynchronously (with an optional completion routine). When called synchronously, control returns to your application after the sound is completed. When called asynchronously, control returns to your application immediately, and your application is free to perform other tasks while the sound is produced.

The Sound Driver uses interrupts to produce sound. Other device drivers, such as the Disk Driver, turn off interrupts while they're operating. Be sure you don't call such drivers while you're producing sounds.

(note)

To produce continuous, unbroken sounds, it's sometimes advantageous to preallocate space for all the sound buffers you require before you make the first StartSound call. Then, while one asynchronous StartSound call is being completed, you can calculate the waveform values for the next call.

To avoid the click that may occur between StartSound calls when using the four-tone synthesizer, set the duration field to a large value and just change the value of one of the rate fields to start a new sound.

To determine when the sound initiated by a StartSound call has been completed, you can poll the SoundDone function. You can cancel any current StartSound call and any pending asynchronous StartSound calls by calling StopSound. By calling GetSoundVol and SetSoundVol, you can get and set the current speaker volume level.

SOUND DRIVER ROUTINES

Assembly-language note: There are no trap macros for these routines; assembly-language programmers can take the equivalent actions noted in the routine descriptions.

```
PROCEDURE StartSound (synthRec: Ptr; numBytes: LONGINT; completionRtn:
            ProcPtr);
```

StartSound begins producing the sound(s) described by the synthesizer buffer pointed to by synthRec. NumBytes indicates the length of the synthesizer buffer (in bytes), and completionRtn points to a completion routine to be executed when the sound finishes:

- If completionRtn is POINTER(-1), the sound will be produced synchronously.

- If completionRtn is NIL, the sound will be produced asynchronously, but no completion routine will be executed.

- Otherwise, the sound will be produced asynchronously and the routine pointed to by completionRtn will be executed when the sound finishes.

(warning)

> You may want the completion routine to start the next sound when one sound finishes, but beware: Completion routines are executed at the interrupt level, and shouldn't make any calls to the Memory Manager. Be sure to preallocate all the space you'll need. Or, instead of using a completion routine to start the next sound, the

completion routine can post an application-defined event and your application's main event loop can start the next sound when it gets the event.

Because the type of pointer for each type of synthesizer buffer is different and the type of the synthRec parameter is Ptr, you'll need to do something like the following example (which applies to the free-form synthesizer):

```
VAR myPtr: Ptr;
    myHandle: Handle;
    myFFPtr: FFSynthPtr;
    . . .
myHandle := NewHandle(buffSize);     {allocate space for the buffer}
HLock(myHandle);                     {lock the buffer}
myPtr := myHandle^;                  {dereference the handle}
myFFPtr := FFSynthPtr(myPtr);        {coerce type to FFSynthPtr}
myFFPtr^.mode := ffMode;             {identify the synthesizer}
    . . .                            {fill the buffer with values }
                                     {describing the sound}
StartSound(myPtr,buffSize,POINTER(-1));  {produce the sound}
HUnlock(myHandle)                        {unlock the buffer}
```

where buffSize is the length of the synthesizer record.

The sounds are generated as follows:

- Free-form synthesizer: The magnitudes described by each byte in the waveform description are generated sequentially until the number of bytes specified by the numBytes parameter have been written.

- Square-wave synthesizer: The sounds described by each sound triplet are generated sequentially until either the end of the buffer has been reached (indicated by a count, amplitude, and duration of Ø in the square-wave buffer), or the number of bytes specified by the numBytes parameter have been written.

- Four-tone synthesizer: All four sounds are generated for the length of time specified by the duration integer in the four-tone record.

Assembly-language note: Assembly-language programmers can make a Device Manager Write call with the following parameters: ioRefNum must be -4, ioBuffer must point to the synthesizer buffer, and ioReqCount must contain the length of the synthesizer buffer.

PROCEDURE StopSound;

StopSound immediately stops the current StartSound call (if any), executes the current StartSound call's completion routine (if any), and cancels any pending asynchrounous StartSound calls.

Assembly-language note: To stop sound from assembly-language, you can make a Device Manager KillIO call (and, when using the square-wave synthesizer, set the global variable CurPitch to Ø). Although StopSound executes the completion routine of only the current StartSound call, KillIO executes the completion routine of every pending asynchronous call.

FUNCTION SoundDone : BOOLEAN;

SoundDone returns TRUE if the Sound Driver isn't currently producing sound and there are no asynchronous StartSound calls pending; otherwise it returns FALSE.

Assembly-language note: Assembly-language programmers can poll the ioResult field of the most recent Device Manager Write call's parameter block to determine when the Write call finishes.

PROCEDURE GetSoundVol (VAR level: INTEGER);

GetSoundVol returns the current speaker volume, from Ø (silence) to 7 (loudest).

Assembly-language note: Assembly-language programmers can get the speaker volume level from the low-order three bits of the global variable SdVolume.

PROCEDURE SetSoundVol (level: INTEGER);

SetSoundVol immediately sets the speaker volume to the specified level, from Ø (silence) to 7 (loudest).

Assembly-language note: To set the speaker volume level from assembly language, call this Pascal routine from your program. As a side effect, it will set the low-order three bits of the global variable SdVolume to the specified level.

(note)

Your program shouldn't change the speaker volume unless it's a Control Panel-like desk accessory, since it's really up to the user to choose the desired volume level via the Control Panel.

SOUND DRIVER HARDWARE

This section describes how the Sound Driver uses the Macintosh hardware to produce sound, and how advanced programmers can intervene in the process to control the square-wave synthesizer. You can skip this section if it doesn't interest you, and you'll still be able to use the Sound Driver as described.

The Sound Driver and disk-motor speed-control circuitry share a special 74Ø-byte buffer in memory, of which the Sound Driver uses the 37Ø even-numbered bytes to generate sound. Every horizontal retrace interval (every 44.93 microseconds--when the beam of the video screen moves from the right edge of the screen to the left) the 68ØØØ automatically fetches two bytes from this buffer and sends the high-order byte to the speaker. Every vertical retrace interval (every 16.6 milliseconds--when the beam of the video screen moves from the bottom of the screen to the top) the Sound Driver fills its half of the 74Ø-byte buffer with the next set of values. For square-wave sound the buffer is filled with a constant value; for more complex sound the buffer is filled with many values.

(note)

All the frequencies generated by the Sound Driver are multiples of this 44.93 microsecond period. The highest frequency the Sound Driver can physically generate corresponds to twice this period, 89.96 microseconds, or a frequency of 11116 Hz.

Assembly-language note: Assembly-language programmers can determine the value in the 740-byte buffer from the global variable SoundLevel. You can cause the square-wave synthesizer to start generating sound, and then change the amplitude of the sound being generated any time you wish:

1. Make an asynchronous Device Manager Write call to the Sound Driver specifying the count, amplitude, and duration of the sound you want. The amplitude you specify will be placed in the 740-byte buffer, and the Sound Driver will begin producing sound.

2. Whenever you want to change the sound being generated, make an **immediate** Control call to the Sound Driver with the following parameters: ioRefNum must be -4, csCode must 3, and csParam must provide the new amplitude level. The amplitude you specify will be placed in the 740-byte buffer, and the sound will change. You can continue to change the sound until the time specified by the duration has elapsed.

   When the immediate Control call is completed, the Device Manager will execute the completion routine (if any) of the currently executing Write call. For this reason, the Write call shouldn't have a completion routine.

SUMMARY OF THE SOUND DRIVER

Constants

```
CONST { Mode values for synthesizers }

      swMode = -1; {square-wave synthesizer}
      ftMode =  1; {four-tone synthesizer}
      ffMode =  0; {free-form synthesizer}
```

Data Types

```
TYPE { Free-form synthesizer }

     FFSynthRec = RECORD
                    mode:      INTEGER;  {always ffMode}
                    count:     Fixed;    {"sizing" factor}
                    waveBytes: FreeWave  {waveform description}
                  END;

     FFSynthPtr = ^FFSynthRec;

     FreeWave   = PACKED ARRAY [0..30000] OF Byte;

     { Square-wave synthesizer }

     SWSynthRec = RECORD
                    mode:     INTEGER; {always swMode}
                    triplets: Tones    {sounds}
                  END;

     SWSynthPtr = ^SWSynthRec;

     Tones = ARRAY [0..5000] OF Tone;
     Tone  = RECORD
               count:     INTEGER; {frequency}
               amplitude: INTEGER; {amplitude, 0-255}
               duration:  INTEGER  {duration in ticks}
             END;

     { Four-tone synthesizer }

     FTSynthRec  = RECORD
                     mode:   INTEGER;    {always ftMode}
                     sndRec: FTSndRecPtr {tones to play}
                   END;

     FTSynthPtr  = ^FTSynthRec;
```

```
FTSoundRec  = RECORD
                  duration:    INTEGER; {duration in ticks}
                  sound1Rate:  Fixed;   {tone 1 cycle rate}
                  sound1Phase: LONGINT; {tone 1 byte offset}
                  sound2Rate:  Fixed;   {tone 2 cycle rate}
                  sound2Phase: LONGINT; {tone 2 byte offset}
                  sound3Rate:  Fixed;   {tone 3 cycle rate}
                  sound3Phase: LONGINT; {tone 3 byte offset}
                  sound4Rate:  Fixed;   {tone 4 cycle rate}
                  sound4Phase: LONGINT; {tone 4 byte offset}
                  sound1Wave:  WavePtr; {tone 1 waveform}
                  sound2Wave:  WavePtr; {tone 2 waveform}
                  sound3Wave:  WavePtr; {tone 3 waveform}
                  sound4Wave:  WavePtr  {tone 4 waveform}
                END;

FTSndRecPtr = ^FTSoundRec;

Wave    = PACKED ARRAY [Ø..255] OF Byte;
WavePtr = ^Wave;
```

Routines [No trap macros]

```
PROCEDURE StartSound  (synthRec: Ptr; numBytes: LONGINT; completionRtn:
                        ProcPtr);
PROCEDURE StopSound;
FUNCTION  SoundDone :  BOOLEAN;
PROCEDURE GetSoundVol (VAR level: INTEGER);
PROCEDURE SetSoundVol (level: INTEGER);
```

Assembly-Language Information

Routines

Name	Equivalent for Assembly-Language
StartSound	Call Write with ioRefNum=-4, ioBuffer=pointer to synthesizer buffer, ioReqCount=length of buffer
StopSound	Call KillIO and (for square-wave) set CurPitch to Ø
SoundDone	Poll ioResult field of most recent Write call's parameter block
GetSoundVol	Get low-order three bits of variable SdVolume
SetSoundVol	Call this Pascal procedure from your program

Variables

Name	Size	Contents
SdVolume	1 byte	Speaker volume level (low-order three bits only)
SoundPtr	4 bytes	Pointer to four-tone record
SoundBase	4 bytes	Pointer to free-form buffer
SoundLevel	1 byte	Amplitude in 740-byte buffer
CurPitch	2 bytes	Value of count in square-wave synthesizer buffer

Sound Driver Values For Notes

The following table contains values for the rate field of a four-tone synthesizer and the count field of a square-wave synthesizer. The four left columns give Ptolemy's diatonic scale, which you can use for a perfectly tuned C major scale, and the four right columns give an equal-tempered scale, for when the application will use other keys.

	Ptolemy's Diatonic Scale				Equal-Tempered Scale			
	Rate for Four-Tone		Count for Square-Wave		Rate for Four-Tone		Count for Square-Wave	
Note	Long	Fixed	Word	Integer	Long	Fixed	Word	Integer
3 octaves below middle C								
C	612B	0.37957	5CBA	23749	604C	0.37615	5D9D	23965
C#	67A5	0.40487	56EF	22265	6606	0.39853	585C	22620
D	6D50	0.42701	52D6	21111	6C16	0.42222	5367	21351
D#	749A	0.45548	4D46	19791	7286	0.44736	4EB7	20151
E	7976	0.47446	4A2F	19000	7953	0.47392	4A4D	19021
F	818E	0.50609	458C	17812	808A	0.50210	4622	17954
F#	8A35	0.53988	4131	16697	882F	0.53197	4232	16946
G	91C0	0.56935	3DD1	15833	9048	0.56360	3E76	15995
G#	9B78	0.60731	39F4	14843	98DC	0.59710	3AF9	15097
A	A1F2	0.63261	37A3	14250	A7F3	0.63261	37AA	14250
A#	AA0B	0.66424	34FD	13571	AB94	0.67023	348A	13450
B	B631	0.71169	3174	12666	B5C8	0.71009	3197	12695

2 octaves below middle C

C	C256	0.75914	2E5D	11875	C097	0.75231	2ECF	11983
C#	CF4B	0.80974	2B77	11133	CC0C	0.79706	2C2E	11310
D	DAA1	0.85403	2936	10555	D82D	0.84443	2983	10675
D#	E934	0.91096	26A3	9896	E50C	0.89472	275B	10075
E	F2EC	0.94892	2517	9500	F2A6	0.94784	2527	9511
F	1031D	1.01218	22C6	8906	10113	1.00420	2311	8977
F#	1146A	1.07976	2099	8349	1105E	1.06394	2119	8473
G	12381	1.13870	1EE9	7916	12090	1.12720	1F3D	7997
G#	136F0	1.21462	1CFA	7422	13167	1.19420	1D7D	7549
A	143E5	1.26523	1BD1	7125	143E6	1.26523	1BD5	7125
A#	15417	1.32849	1A7E	6786	15728	1.34046	1A45	6725
B	16C62	1.42338	18BA	6333	16B91	1.42018	18CB	6347

1 octave below middle C

C	184AC	1.51827	172F	5937	1812E	1.50461	1767	5991
C#	19E96	1.61949	15BC	5566	19818	1.59411	1617	5655
D	1B542	1.70805	149B	5278	1B059	1.68886	14DA	5338
D#	1D269	1.82193	1351	4948	1CA18	1.78943	13AE	5038
E	1E5D8	1.89784	128C	4750	1E54B	1.89568	1293	4755
F	2063B	2.02436	1163	4453	20227	2.00840	1188	4488
F#	228D5	2.15951	104C	4174	220BD	2.12788	10BC	4236
G	24703	2.27741	F74	3958	24121	2.25440	F9F	3999
G#	26DE1	2.42923	E7D	3711	2636E	2.38840	EBE	3774
A	287CA	2.53045	DE9	3562	287CC	2.53045	DEA	3562
A#	2A82E	2.65697	D3F	3393	2AE50	2.68091	D22	3362
B	2D8C4	2.84676	C5D	3167	2D722	2.84036	C66	3174

Middle C

C	30959	3.03654	B97	2969	3025D	3.00922	BB4	2996
C#	33D2C	3.23898	ADE	2783	33030	3.18823	B0B	2827
D	36A85	3.41611	A4E	2639	360B2	3.37772	A6D	2669
D#	3A4D2	3.64385	9A9	2474	39430	3.57886	9D7	2519
E	3CBB0	3.79568	946	2375	3CD97	3.79136	94A	2378
F	4AC77	4.04872	8B1	2227	4044D	4.01680	8C4	2244
F#	451AA	4.31902	826	2087	43E4F	4.25576	846	2118
G	48E06	4.55481	7BA	1979	48241	4.50880	7CF	1999
G#	4DBC3	4.85847	73F	1855	4C6DD	4.77680	75F	1887
A	50F95	5.06090	6F4	1781	50F97	5.06090	6F5	1781
A#	5505D	5.31395	6A0	1696	55CA1	5.36183	691	1681
B	5B188	5.69352	62F	1583	5AE44	5.68072	633	1587

1 octave above middle C

C	612B3	6.07308	5CC	1484	604B9	6.01845	5DA	1498
C#	67A59	6.47796	56F	1392	6605F	6.37645	586	1414
D	6D50A	6.83222	527	1319	6C165	6.75544	536	1334
D#	749A4	7.28770	4D4	1237	72861	7.15773	4EB	1259
E	79760	7.59136	4A3	1187	7952E	7.58273	4A5	1189
F	818EF	8.09745	459	1113	8089B	8.03361	462	1122
F#	8A354	8.63804	413	1044	882F3	8.51152	423	1059
G	91C0D	9.10963	3DD	989.6	9048B	9.01761	3E8	999.7
G#	9B786	9.71693	39F	927.7	980B9	9.55361	3B0	943.6
A	A1F2B	10.12181	37A	890.6	A1F2F	10.12181	37B	890.6
A#	AA0BA	10.62790	350	848.2	AB941	10.72365	349	840.6
B	B6311	11.38703	317	791.6	B5C87	11.36144	319	793.4

2 octaves above middle C

C	C2567	12.14617	2E6	742.2	C0972	12.03690	2ED	748.9
C#	CF4B2	12.95591	2B7	695.8	CC0BE	12.75290	2C3	706.9
D	DAA14	13.66444	293	659.7	D82C9	13.51089	29B	667.2
D#	E9349	14.57540	26A	618.5	E50C2	14.31546	276	629.7
E	F2EC1	15.18271	251	593.7	F2A5B	15.16546	252	594.4
F	1031DF	16.19489	22D	556.6	101135	16.06722	231	561
F#	1146A8	17.27608	20A	521.8	1105E6	17.02304	212	529.5
G	12381B	18.21925	1EF	494.8	120904	18.03522	1F4	499.8
G#	136F0C	19.43387	1D0	463.9	131B72	19.10721	1D8	471.8
A	143E57	20.24361	1BD	445.3	143E5D	20.24361	1BD	445.3
A#	154175	21.25579	1A8	424.1	157282	21.44730	1A4	420.3
B	16C622	22.77407	18C	395.8	16B90F	22.72288	18D	396.7

3 octaves above middle C

C	184ACF	24.29222	173	371	1812E4	24.07380	176	374.5
C#	19E965	25.91171	15C	348	19817C	25.50580	161	353.4
D	1B5429	27.32875	14A	330	1B0593	27.02177	14E	333.6
D#	1D2692	29.15067	135	309	1CA183	28.63091	13B	314.9
E	1E5D83	30.36528	129	297	1E54B7	30.33091	129	297.2
F	2063BF	32.38963	116	278	20226A	32.13444	119	280.5
F#	228D50	34.82806	105	261	220BCC	34.04608	109	264.8
G	247036	36.43834	F7	247	241208	36.07044	FA	249.9
G#	260E18	38.86756	E8	232	2636E4	38.21442	EC	235.9
A	287CAE	40.48704	DF	223	287CBB	40.48723	DF	222.7
A#	2A82EA	42.51139	D4	212	2AE505	42.89461	D2	210.2
B	2D8C44	45.54790	C6	198	2D721D	45.44576	C6	198.4

GLOSSARY

amplitude: The maximum vertical distance of a periodic wave from the horizontal line about which the wave oscillates.

four-tone record: A data structure describing the four tones produced by a four-tone synthesizer.

four-tone synthesizer: The part of the Sound Driver used to make simple harmonic tones, with up to four "voices" producing sound simultaneously.

free-form synthesizer: The part of the Sound Driver used to make complex music and speech.

frequency: The number of cycles per second (also called Hertz) at which a wave oscillates.

magnitude: The vertical distance between any given point on a wave and the horizontal line about which the wave oscillates.

period: The time elapsed during one complete cycle of a wave.

phase: Some fraction of a wave cycle (measured from a fixed point on the wave).

square-wave synthesizer: The part of the Sound Driver used to produce less harmonic sounds than the four-tone synthesizer, such as beeps.

synthesizer buffer: A description of the sound to be generated by a synthesizer.

waveform: The physical shape of a wave.

waveform description: A sequence of bytes describing a waveform.

wavelength: The horizontal extent of one complete cycle of a wave.

The Serial Drivers: A Programmer's Guide /SDRVR/SERIAL

See Also: Inside Macintosh: A Road Map
Macintosh Memory Management: An Introduction
Programming Macintosh Applications in Assembly Language
The Memory Manager: A Programmer's Guide
The Device Manager: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide

Modification History: First Draft Bradley Hacker 9/28/84

ABSTRACT

The Macintosh RAM Serial Driver and ROM Serial Driver are Macintosh device drivers for handling asynchronous serial communication between a Macintosh application and serial devices. This manual describes the Serial Drivers in detail.

TABLE OF CONTENTS

ABOUT THIS MANUAL

The Macintosh RAM Serial Driver and ROM Serial Driver are Macintosh device drivers for handling asynchronous serial communication between a Macintosh application and serial devices. This manual describes the Serial Drivers in detail. *** Eventually it will become part of the comprehensive Inside Macintosh manual. ***

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal and the information in the following manuals:

- Inside Macintosh: A Road Map

- Macintosh Memory Management: An Introduction

- Programming Macintosh Applications in Assembly Language, if you're using assembly language

You should also be familiar with the following:

- resources, as discussed in the Resource Manager manual

- events, as discussed in the Toolbox Event Manager manual

- the Memory Manager

- interrupts and the use of devices and device drivers, as described in the Device Manager manual

- asynchronous serial data communication

SERIAL COMMUNICATION

The Serial Drivers support full-duplex asynchronous serial communication. Serial data is transmitted over a single-path communication line, one bit at a time (as opposed to parallel data, which is transmitted over a multiple-path communication line, multiple bits at a time). Full-duplex means that the Macintosh and another serial device connected to it can transmit data simultaneously (as opposed to half-duplex operation, in which data can be transmitted by only one device at a time). Asynchronous communication means that the Macintosh and other serial devices communicating with it don't share a common timer, and no timing data is transmitted. The time interval between characters transmitted asynchronously can be of any length. The format of asynchronous serial data communication used by the Serial Drivers is shown in Figure 1.

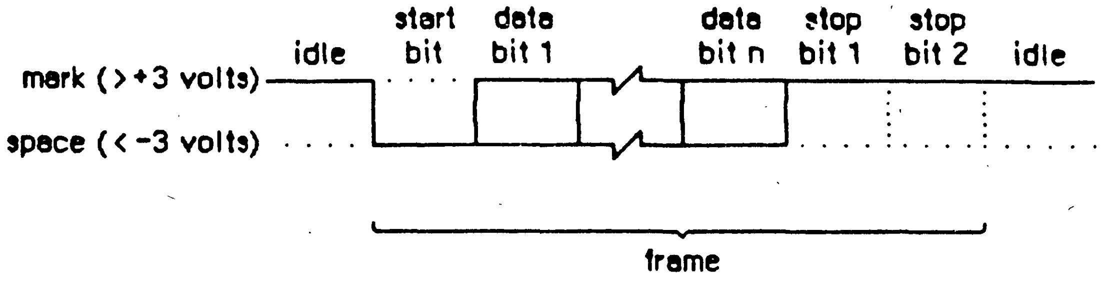


Figure 1. Asynchronous Data Transmission

When a transmitting serial device is idle (not sending data), it maintains the transmission line in a continuous state ("mark" in Figure 1). The transmitting device may begin sending a character at any time by sending a start bit. The start bit tells the receiving device to prepare to receive a character. The transmitting device then transmits 5, 6, 7, or 8 data bits, optionally followed by a parity bit. The value of the parity bit is chosen such that the number of 1's among the data and parity bits is even or odd, depending on whether the parity is even or odd, respectively. Finally, the transmitting device sends 1, 1.5, or 2 stop bits, indicating the end of the character. The measure of the total number of bits sent over the transmission line per second is called the baud rate.

If a parity bit is set incorrectly, the receiving device will note a parity error. The time elapsed from the start bit to the last stop bit is called a frame. If the receiving device doesn't get a stop bit after the data and parity bits, it will note a framing error. After the stop bits, the transmitting device may send another character or maintain the line in the mark state. If the line is held in the "space" state (Figure 1) for one frame or longer, a break occurs. Breaks are used to interrupt data transmission.

ABOUT THE SERIAL DRIVERS

There are two Macintosh device drivers for serial communication: the RAM Serial Driver and the ROM Serial Driver. The two drivers are nearly identical, although the RAM driver has a few features the ROM driver doesn't. Both allow Macintosh applications to communicate with serial devices via the two RS-232/RS-422 serial ports on the back of the Macintosh.

(note)

> On a Lisa running MacWorks, the RAM Serial Driver acts as a ROM driver.

Each Serial Driver actually consists of four drivers: one input driver and one output driver for the modem port, and one input driver and one output driver for the printer port (Figure 2). Each input driver receives data via a serial port and transfers it to the application.

Each output driver takes data from the application and sends it out through a serial port. The input and output drivers for a port are closely related, and share some of the same routines. Each driver does, however, have a separate device control entry, which allows the Serial Drivers to support full-duplex communication. An individual port can both transmit and receive data at the same time. The serial ports are controlled by the Macintosh's Zilog Z8530 Serial Communications Controller (SCC). Channel A of the SCC controls the modem port, and channel B controls the printer port.

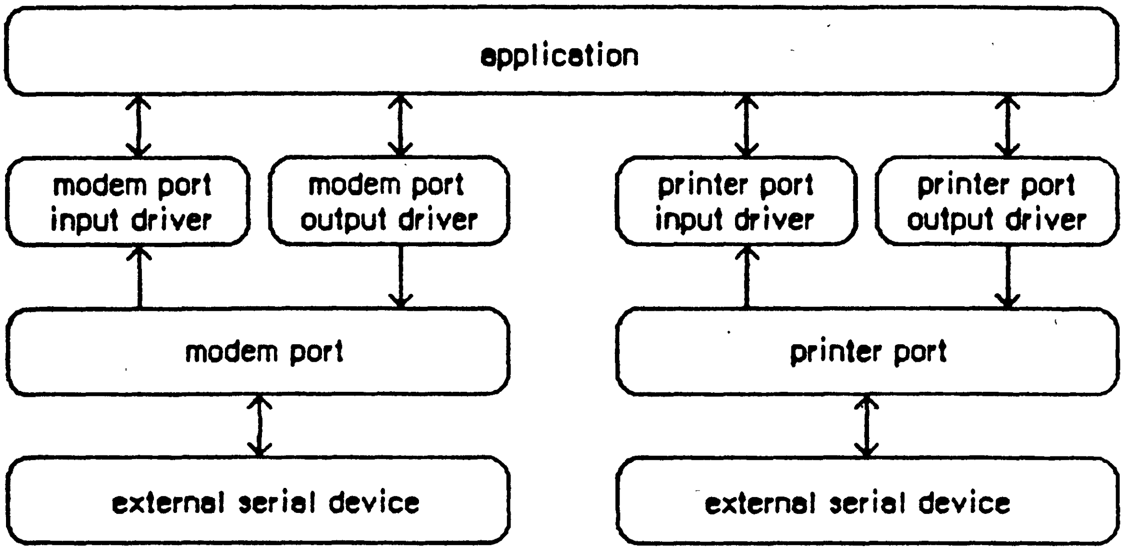


Figure 2. Input and Output Drivers of a Serial Driver

Data received via a serial port passes through a three-character buffer in the SCC and then into a buffer in the input driver for the port. Characters are removed from the input driver's buffer each time an application issues a Read call to the driver. Each input driver's buffer can initially hold up to 64 characters, but your application can increase this if necessary. The following errors may occur:

- If the SCC buffer ever overflows (because the input driver doesn't read it often enough), a hardware overrun error occurs.

- If an input driver's buffer ever overflows (because the application doesn't issue Read calls to the driver often enough), a software overrun error occurs.

The printer port should be used for output-only connections to devices such as printers, or at low baud rates (300 baud or less). The modem port has no such restrictions. It may be used simultaneously with disk accesses without fear of hardware overrun errors, because whenever the Disk Driver must turn off interrupts for longer than 100 microseconds, it stores any data received via the modem port and later passes the data to the modem port's input driver.

All four drivers default to 9600 baud, eight data bits per character, no parity bit, and two stop bits. You can change any of these options.

The Serial Drivers support CTS (clear to send) hardware handshaking and XOn/XOff software flow control.

(note)

> The ROM Serial Driver defaults to hardware handshake only; it doesn't support XOn/XOff input flow control— only output flow control. Use the RAM Serial Driver if you want XOn/XOff input flow control. The RAM Serial Driver defaults to no hardware handshaking and no software flow control.

Whenever an input driver receives a break, it terminates any pending Read requests, but not Write requests. You can choose to have the input drivers terminate Read requests whenever a parity, overrun, or framing error occurs.

(note)

> The ROM Serial Driver always terminates input requests when an error occurs. Use the RAM Serial Driver if you don't want input requests to be terminated by errors.

You can request the Serial Drivers to post device driver events whenever a change in the hardware handshake status or a break occurs, if you want your application to take some specific action upon these occurrences.

USING THE SERIAL DRIVERS

This section introduces you to the Serial Driver routines described in detail in the next section, and discusses other calls you can make to communicate with the Serial Drivers.

Drivers are referred to by name and reference number:

Driver	Driver name	Reference number
Modem port input	.AIn	-6
Modem port output	.AOut	-7
Printer port input	.BIn	-8
Printer port output	.BOut	-9

Before you can receive data through a port, both the input and output drivers for the port must be opened. Before you can send data through a port, the output driver for the port must be opened. To open the ROM input and output drivers, call the Device Manager Open function; to open the RAM input and output drivers, call the Serial Driver function RAMSDOpen. The RAM drivers occupy less than 2K bytes of memory in the application heap.

When you open an output driver, the Serial Driver initializes local variables for the output driver and the associated input driver, allocates and locks buffer storage for both drivers, installs interrupt handlers for both drivers, and initializes the correct SCC channel (ROM

Serial Driver only). When you open an input driver, the Serial Driver only notes the location of its device control entry.

If you would like to reclaim the space occupied by a driver's storage, you can can call the Device Manager Close function to close the ROM Serial Driver, and RAMSDClose to close the RAM Serial Driver; RAMSDClose will also release the memory occupied by the driver itself. When you close an output driver, the Serial Driver resets the appropriate SCC channel, releases all local variable and buffer storage space, and restores any changed interrupt vectors. If it's already open, the ROM Serial Driver is automatically closed when you call RAMSDOpen.

(warning)

> You should not close the ROM Serial Driver unless you're immediately going to open a RAM Serial Driver for the same port, or mouse interrupts will be lost.

To transmit serial data out through a port, make a Device Manager Write call to the output driver for the port. You must pass the following parameters:

- the driver reference number -7 or -9, depending on whether you're using the modem port or the printer port
- a data buffer that contains the data you want to transmit
- the number of bytes you want to transmit

To receive serial data from a port, make a Device Manager Read call to the input driver for the port. You must pass the following parameters:

- the driver reference number -6 or -8, depending on whether you're using the modem port or the printer port
- the location of the buffer where you want to receive the data
- the number of bytes you want to receive

There are six different calls you can make to the Serial Driver's control routine:

- KillIO causes all current I/O requests to be aborted and any bytes remaining in both input buffers to be discarded. KillIO is a Device Manager call.
- SerReset resets and reinitializes a driver with new data bits, stop bits, parity bit, and baud rate information.
- SerSetBuf allows you to specify a new input buffer.
- SerHShake allows you to specify handshake options.

- SerSetBrk sets break mode.

- SerClrBrk clears break mode.

Advanced programmers can make nine additional calls to the RAM Serial Driver's control routine; see the "Advanced Control Calls" section.

There are two different calls you can make to the Serial Driver's status routine:

- SerGetBuf returns the number of available unread bytes currently stored by an input driver.

- SerErrFlag returns information about errors, I/O requests, and handshake.

Assembly-language note: Control and Status calls to the RAM Serial Driver may be immediate (use IMMED as the second argument to the routine macro).

SERIAL DRIVER ROUTINES

This section describes the Serial Driver routines. Most of them return an integer result code of type OSErr; each routine description lists all of the applicable result codes.

Assembly-language note: There are no trap macros for these routines. Assembly-language programmers can in some cases make equivalent Control and Status calls, as indicated in the routine descriptions.

Opening and Closing the RAM Serial Driver

```
FUNCTION RAMSDOpen (whichPort: SPortSel; rsrcType: OSType; rsrcID:
            INTEGER) : OSErr;
```

RAMSDOpen closes the ROM Serial Driver and opens the RAM input and output drivers for the port identified by the whichPort parameter, which must be a member of the SPortSel set:

```
TYPE SPortSel = (sPortA, {modem port}
                 sPortB  {printer port});
```

RsrcType and rsrcID indicate the resource type and resource ID of the RAM Serial Driver, which should be stored in your application's resource file. (OSType is an Operating System Utility data type declared the same as ResType in the Resource Manager.)

<u>Result codes</u>	noErr	No error
	openErr	Can't open driver

PROCEDURE RAMSDClose (whichPort: SPortSel);

RAMSDClose closes the RAM input and output drivers for the port identified by the whichPort parameter, which must be a member of the SPortSel set (defined in the description of RAMSDOpen above).

Changing Serial Driver Information

FUNCTION SerReset (refNum: INTEGER; serConfig: INTEGER) : OSErr;

SerReset resets and reinitializes the input or output driver having the reference number refNum according to the information in serConfig. Figure 3 shows the format of serConfig.

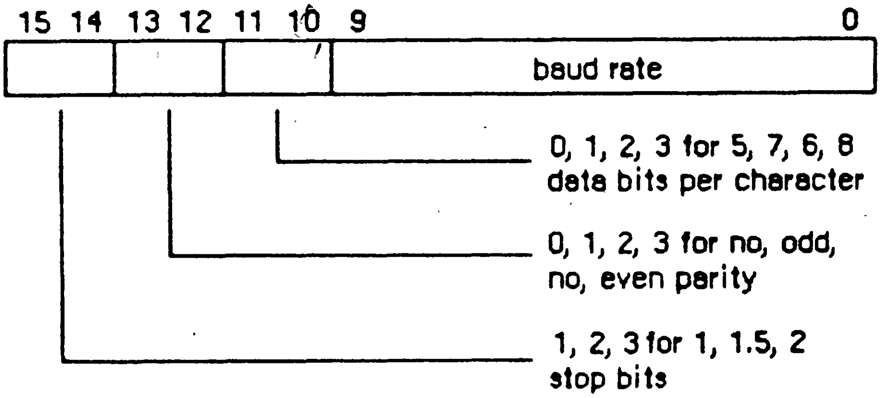

15 14 13 12 11 10 9 0
baud rate
0, 1, 2, 3 for 5, 7, 6, 8 data bits per character
0, 1, 2, 3 for no, odd, no, even parity
1, 2, 3 for 1, 1.5, 2 stop bits

Figure 3. Driver Reset Information

You can use the following predefined constants to set the values of various bits of serConfig:

```
CONST baud300     = 380;    {300 baud}
      baud600     = 189;    {600 baud}
      baud1200    = 94;     {1200 baud}
      baud1800    = 62;     {1800 baud}
      baud2400    = 46;     {2400 baud}
      baud3600    = 30;     {3600 baud}
      baud4800    = 22;     {4800 baud}
      baud7200    = 14;     {7200 baud}
      baud9600    = 10;     {9600 baud}
      baud19200   = 4;      {19200 baud}
      baud57600   = 0;      {57600 baud}
      stop10      = 16384;  {1 stop bit}
      stop15      = -32768; {1.5 stop bits}
      stop20      = -16384; {2 stop bits}
      noParity    = 8192;   {no parity}
      oddParity   = 4096;   {odd parity}
      evenParity  = 12288;  {even parity}
      data5       = 0;      {5 data bits}
      data6       = 2048;   {6 data bits}
      data7       = 1024;   {7 data bits}
      data8       = 3072;   {8 data bits}
```

For example, the default setting of 9600 baud, eight data bits, two stop bits, and no parity bit is equivalent to baud9600+data8+stop20+noParity.

<u>Assembly-language</u> <u>note</u>: SerReset is equivalent to a Control call with csCode=8 and csParam=serConfig.

<u>Result</u> <u>codes</u>   noErr   No error

FUNCTION SerSetBuf (refNum: INTEGER; serBPtr: Ptr; serBLen: INTEGER) :
         OSErr;

SerSetBuf specifies a new input buffer for the input driver having the reference number refNum. SerBPtr points to the buffer, and serBLen specifies the number of bytes in the buffer. If serBLen is 0, a 64-byte default buffer provided by the driver is used.

(warning)
You must lock this buffer while it's in use.

<u>Assembly-language</u> <u>note</u>: SerSetBuf is equivalent to a Control call with csCode=9, csParam=serBPtr, and csParam+2=serBLen.

Result codes noErr No error

FUNCTION SerHShake (refNum: INTEGER; flags: SerShk) : OSErr;

SerHShake sets handshake options and other control information, as specified by the flags parameter, for the input or output driver having the reference number refNum. The flags parameter has the following data structure:

```
TYPE SerShk = PACKED RECORD
                  fXOn: Byte; {XOn/XOff output flow control flag}
                  fCTS: Byte; {CTS hardware handshake flag}
                  xOn:  CHAR; {XOn character}
                  xOff: CHAR; {XOff character}
                  errs: Byte; {errors that cause abort}
                  evts: Byte; {status changes that cause events}
                  fInX: Byte; {XOn/XOff input flow control flag}
                  null: Byte  {not used}
                END;
```

If fXOn is nonzero, XOn/XOff output flow control is enabled; if fInX is nonzero, XOn/XOff input flow control is enabled. XOn and xOff specify the XOn character and XOff character used for XOn/XOff flow control. If fCTS is nonzero, CTS hardware handshake is enabled. The errs field indicates which errors will cause input requests to be aborted; for each type of error, there's a predefined constant in which the corresponding bit is set:

```
CONST parityErr    = 16;  {set for parity error}
      hwOverrunErr = 32;  {set for hardware overrun error}
      framingErr   = 64;  {set for framing error}
```

(note)

> The ROM Serial Driver doesn't support XOn/XOff input flow control or aborts caused by error conditions.

The evts field indicates whether changes in the CTS or break status will cause the Serial Driver to post device driver events; you can use the following predefined constants to set or test the value of evts:

```
CONST ctsEvent   = 32;    {set if CTS change will cause event to }
                          { be posted}
      breakEvent = 128;   {set if break status change will cause }
                          { event to be posted}
```

(warning)

> Use of this option is discouraged because of the long time that interrupts are disabled while such an event is posted.

Assembly-language note: SerHShake is equivalent to a Control call with csCode=1Ø and csParam through csParam+6 equivalent to the fields of a variable of type SerShk.

Result codes	noErr	No error

FUNCTION SerSetBrk (refNum: INTEGER) : OSErr;

SerSetBrk sets break mode in the input or output driver having the reference number refNum.

Assembly-language note: SerSetBrk is equivalent to a Control call with csCode=12.

Result codes	noErr	No error

FUNCTION SerClrBrk (refNum: INTEGER) : OSErr;

SerClrBrk clears break mode in the input or output driver having the reference number refNum.

Assembly-language note: SerClrBrk is equivalent to a Control call with csCode=11.

Result codes	noErr	No error

Getting Serial Driver Information

FUNCTION SerGetBuf (refNum: INTEGER; VAR count: LONGINT) : OSErr;

SerGetBuf returns, in the count parameter, the number of bytes in the buffer of the input driver having the reference number refNum.

Assembly-language note: SerGetBuf is equivalent to a Status call with csCode=2. The number of bytes in the buffer is returned in csParam.

Result codes	noErr	No error

```
FUNCTION SerErrFlag (refNum: INTEGER; VAR serSta: SerStaRec) : OSErr;
```

SerErrFlag returns in serSta three words of status information for the input or output driver having the reference number refNum. The serSta parameter has the following data structure:

```
TYPE SerStaRec = PACKED RECORD
                   cumErrs:  Byte; {cumulative errors}
                   xOffSent: Byte; {XOff sent as input flow }
                                   { control}
                   rdPend:   Byte; {read pending flag}
                   wrPend:   Byte; {write pending flag}
                   ctsHold:  Byte; {CTS flow control hold flag}
                   xOffHold: Byte  {XOff received as output }
                                   { flow control}
                 END;
```

CumErrs indicates which errors have occurred since the last time SerErrFlag was called:

```
CONST swOverrunErr = 1;  {set for software overrun error}
      parityErr    = 16; {set for parity error}
      hwOverrunErr = 32; {set for hardware overrun error}
      framingErr   = 64; {set for framing error}
```

If the driver has sent an XOff character, xOffSent will be equal to the following predefined constant:

```
CONST xOffWasSent = $8Ø; {XOff character was sent}
```

If the driver has a Read or Write call pending, rdPend or wrPend, respectively, will be nonzero. If output has been suspended because the hardware handshake was negated, ctsHold will be nonzero. If output has been suspended because an XOff character was received, xOffHold will be nonzero.

Assembly-language note: SerStatus is equivalent to a Status call with csCode=8. The status information is returned in csParam through csParam+5.

Result codes noErr No error

ADVANCED CONTROL CALLS

This section describes the calls that advanced programmers can make to the RAM Serial Driver's control routine via a Device Manager Control call. *** If you use the high-level Device Manager function named Close, remember that its third parameter (csParam) is a pointer to the csParam value indicated below; to clarify this, the next draft of the Device Manager manual will rename that parameter csParamPtr. ***

csCode = 13 csParam = baudRate

This call provides an additional way (in addition to SerReset) to set the baud rate. CsParam specifies the baud rate. The closest baud rate that the Serial Driver will generate is returned in csParam.

csCode = 19 csParam = char

After this call is made, all incoming characters with parity errors will be replaced by the character specified by the ASCII code in csParam. If csParam is Ø, no character replacement will be done.

csCode = 21

This call unconditionally sets XOff for output flow control. It's equivalent to receiving an XOff character. Data transmission is halted until an XOn is received or a Control call with csCode=24 is made.

csCode = 22

This call unconditionally clears XOff for output flow control. It's equivalent to receiving an XOn character.

csCode = 23

This call sends an XOn character for input flow control if the last input flow control character sent was XOff.

csCode = 24

This call unconditionally sends an XOn character for input flow control, regardless of the current state of input flow control.

csCode = 25

This call sends an XOff character for input flow control if the last input flow control character sent was XOn.

csCode = 26

This call unconditionally sends an XOff character for input flow control, regardless of the current state of input flow control.

csCode = 27

This call resets the SCC channel belonging to the driver specified by ioRefNum. Immediately after this you should either close the RAM Serial Driver or call SerReset to reenable mouse interrupts.

<u>Assembly-language note</u>: Assembly-language programmers can set the volume level with the following instructions:

```
                MOVE.L   (SP)+,AØ         ;get return address
                MOVE.W   (SP)+,DØ         ;get volume level
                MOVE.L   AØ,-(SP)         ;restore return address
                CMP.B    #$FF,$4ØØØØ9     ;Mac or Lisa?
                BEQ.S    LisaSound

                MOVE     SR,-(SP)         ;save status register
                ORI      #$Ø3ØØ,SR        ;only debug interrupts
                MOVE.B   avBufA,D1        ;get VIA port byte
                AND      #$ØØF8,D1        ;clear low 3 bits
                AND      #7,DØ            ;use only low 3 bits
                MOVE.B   DØ,SdVolume      ;update global variable
                OR       DØ,D1            ;combine them
                MOVE.B   D1,avBufA        ;store it back
                RTE

LisaSound       AND.W    #7,DØ            ;use only low 3 bits
                MOVE.B   DØ,SdVolume      ;update global variable
                LSL.W    #1,DØ            ;shift into position
                MOVE.B   $FCDD81,D1       ;read port B
                AND.B    #$F1,D1          ;clear low 3 bits
                OR.B     DØ,D1            ;combine them
                MOVE.B   D1,$FCDD81       ;store it back
                RTS
```

THE SOUND DRIVER HARDWARE

The following paragraphs describe how the Sound Driver uses the Macintosh hardware to produce sound, and how advanced programmers can intervene in the process to control the square-wave synthesizer. You can skip this section if it doesn't interest you, and you'll still be able to use the Sound Driver as described.

The Sound Driver and disk-motor speed-control circuitry share a special 74Ø-byte buffer in memory, of which the Sound Driver uses the 37Ø even-numbered bytes to generate sound. Every horizontal retrace interval (every 44.93 microseconds--when the beam of the video screen moves from the right edge of the screen to the left) the 68ØØØ automatically fetches two bytes from this buffer and sends the high-order byte to the speaker. Every vertical retrace interval (every 16.6 milliseconds--when the beam of the video screen moves from the bottom of the screen to the top) the Sound Driver fills its half of the 74Ø-byte buffer with the next set of values. For square-wave sound the buffer is filled with a constant value; for more complex sound the buffer is filled with

Data Types

```
TYPE SPortSel = (sPortA, {modem port}
                 sPortB  {printer port});

     SerShk = PACKED RECORD
                fXOn: Byte; {XOn/XOff output flow control flag}
                fCTS: Byte; {CTS hardware handshake flag}
                xOn:  CHAR; {XOn character}
                xOff: CHAR; {XOff character}
                errs: Byte; {errors that cause abort}
                evts: Byte; {status changes that cause events}
                fInX: Byte; {XOn/XOff input flow control flag}
                null: Byte  {not used}
              END;

     SerStaRec = PACKED RECORD
                   cumErrs:  Byte; {cumulative errors}
                   xOffSent: Byte; {XOff sent as input flow control}
                   rdPend:   Byte; {read pending flag}
                   wrPend:   Byte; {write pending flag}
                   ctsHold:  Byte; {CTS flow control hold flag}
                   xOffHold: Byte  {XOff received as output flow }
                                   { control}
                END;
```

Serial Driver Routines [No trap macros]

Opening and Closing the RAM Serial Driver

```
FUNCTION  RAMSDOpen  (whichPort: SPortSel; rsrcType: OSType; rsrcID:
                      INTEGER) : OSErr;
PROCEDURE RAMSDClose (whichPort: SPortSel);
```

Changing Serial Driver Information

```
FUNCTION SerReset  (refNum: INTEGER; serConfig: INTEGER) : OSErr;
FUNCTION SerSetBuf (refNum: INTEGER; serBPtr: Ptr; serBLen: INTEGER) :
                    OSErr;
FUNCTION SerHShake (refNum: INTEGER; flags: SerShk) : OSErr;
FUNCTION SerSetBrk (refNum: INTEGER) : OSErr;
FUNCTION SerClrBrk (refNum: INTEGER) : OSErr;
```

Getting Serial Driver Information

```
FUNCTION SerGetBuf  (refNum: INTEGER; VAR count: LONGINT) : OSErr;
FUNCTION SerErrFlag (refNum: INTEGER; VAR serSta: SerStaRec) : OSErr;
```

SUMMARY OF THE SOUND DRIVER

Constants

```
CONST { Mode values for synthesizers }

      swMode = -1; {square-wave synthesizer}
      ftMode =  1; {four-tone synthesizer}
      ffMode =  Ø; {free-form synthesizer}
```

Data Types

```
TYPE { Free-Form synthesizer }

     FFSynthRec = RECORD
                    mode:       INTEGER;  {always ffMode}
                    count:      Fixed;    {"sizing" factor}
                    waveBytes:  FreeWave  {waveform description}
                  END;

     FFSynthPtr = ^FFSynthRec;

     FreeWave   = PACKED ARRAY [Ø..3ØØØØ] OF Byte;

     { Square-Wave synthesizer }

     SWSynthRec = RECORD
                    mode:      INTEGER; {always swMode}
                    triplets:  Tones    {sounds}
                  END;

     SWSynthPtr = ^SWSynthRec;

     Tones      = ARRAY [Ø..5ØØØ] OF Tone;

     Tone       = RECORD
                    count:      INTEGER; {frequency}
                    amplitude:  INTEGER; {amplitude, Ø-255}
                    duration:   INTEGER  {duration, Ø-255 ticks}
                  END;

     {Four-Tone synthesizer }

     FTSynthRec  = RECORD
                     mode:   INTEGER;     {always ftMode}
                     sndRec: FTSndRecPtr  {tones to play}
                   END;

     FTSynthPtr  = ^FTSynthRec;
```

wrPend	Write pending flag
ctsHold	CTS flow control hold flag
xOffHold	XOff received as output flow control

Equivalent Device Manager Calls

Pascal routine	Call	CsCode
SerReset	Control	8
SerSetBuf	Control	9
SerHShake	Control	10
SerSetBrk	Control	12
SerClrBrk	Control	11
SerGetBuf	Status	2
SerErrFlag	Status	8

GLOSSARY

asynchronous communication: A method of data transmission where the receiving and sending devices don't share a common timer, and no timing data is transmitted.

baud rate: The measure of the total number of bits sent over a transmission line per second.

break: The condition resulting when a device maintains its transmission line in the space state for at least one frame.

data bits: Data communication bits that encode transmitted characters.

frame: The time elapsed from the start bit to the last stop bit.

framing error: The condition resulting when a device doesn't receive a stop bit when expected.

full-duplex communication: A method of data transmission where two devices transmit data simultaneously.

hardware overrun error: The condition that occurs when the SCC's buffer becomes full.

input driver: A device driver that receives serial data via a serial port and transfers it to an application.

mark state: The state of a transmission line indicating a binary '1'.

output driver: A device driver that receives data via a serial port and transfers it to an application.

overrun error: See hardware overrun error and software overrun error.

parity bit: A data communication bit used to verify that data bits received by a device match the data bits transmitted by another device.

parity error: The condition resulting when the parity bit received by a device isn't what was expected.

serial data: Data communicated over a single-path communication line, one bit at a time.

software overrun error: The condition that occurs when an input driver's buffer becomes full.

space state: The state of a transmission line indicating a binary 'Ø'.

start bit: A serial data communications bit that signals that the next bits transmitted are data bits.

MACINTOSH USER EDUCATION

The AppleTalk Manager: A Programmer's Guide /NET/ATALK

See Also: Inside Macintosh: A Road Map
Macintosh Memory Management: An Introduction
Programming Macintosh Applications in Assembly Language
The Resource Manager: A Programmer's Guide
The Toolbox Event Manager: A Programmer's Guide
The Memory Manager: A Programmer's Guide
The Device Manager: A Programmer's Guide
Inside AppleTalk

Modification History: First Draft Bradley Hacker &
Bob Anders 1/31/85

ABSTRACT

The AppleTalk Manager is a set of routines and a pair of RAM device drivers that allow Macintosh programs to send and receive information via AppleTalk. This manual describes the AppleTalk Manager in detail.

*** This document doesn't discuss the relevant interface files needed to use the AppleTalk Manager. They are as follows:

- TlAsm/ATalkEqu: include this in your assembly pass to use the equates of the .MPP driver and the .ATP driver

- ABPasIntf: add this to your USES clause in your Pascal program, if you're using any of the Pascal calls to the AppleTalk manager

- ABPasCalls: link this with your Pascal program ***

TABLE OF CONTENTS

ABOUT THIS MANUAL

The AppleTalk Manager is a set of routines and a pair of RAM device drivers that allow Macintosh programs to send and receive information via AppleTalk. This manual describes the AppleTalk Manager in detail. *** Eventually it will become part of the comprehensive Inside Macintosh manual. ***

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal and the information in the following manuals:

- Inside Macintosh: A Road Map

- Macintosh Memory Management: An Introduction

- Programming Macintosh Applications in Assembly Language, if you're using assembly language

You should also be familiar with the following:

- interrupts and the use of devices and device drivers, as described in the Device Manager manual, if you want to write your own assembly-language additions to the AppleTalk Manager

- Inside AppleTalk (Apple Product #nnn) *** number to be supplied ***, if you want to understand AppleTalk protocols in detail

APPLETALK PROTOCOLS

The AppleTalk Manager provides a variety of services that allow Macintosh programs to interact with programs in devices connected to an AppleTalk network. This interaction, achieved through the exchange of variable-length blocks of data (known as packets) over AppleTalk, follows well-defined sets of rules known as protocols.

Although most programmers using AppleTalk needn't understand the details of these protocols, they should understand the information in this section--what the services provided by the different protocols are, and how the protocols are interrelated. Detailed information about AppleTalk protocols is available in Inside AppleTalk.

The AppleTalk system architecture consists of a number of protocols arranged in layers. Each protocol in a specific layer provides services to higher-level layers (known as the protocol's clients) by building on the services provided by lower-level layers. A Macintosh program can use services provided by any of the layers in order to construct more sophisticated or more specialized services.

The AppleTalk Manager contains the following protocols:

- AppleTalk Link Access Protocol
- Datagram Delivery Protocol
- Routing Table Maintenance Protocol
- Name-Binding Protocol
- AppleTalk Transaction Protocol

Figure 1 illustrates the layered structure of the protocols in the AppleTalk Manager; the heavy connecting lines indicate paths of interaction. Note that the Routing Table Maintenance Protocol isn't directly accessible to Macintosh programs.

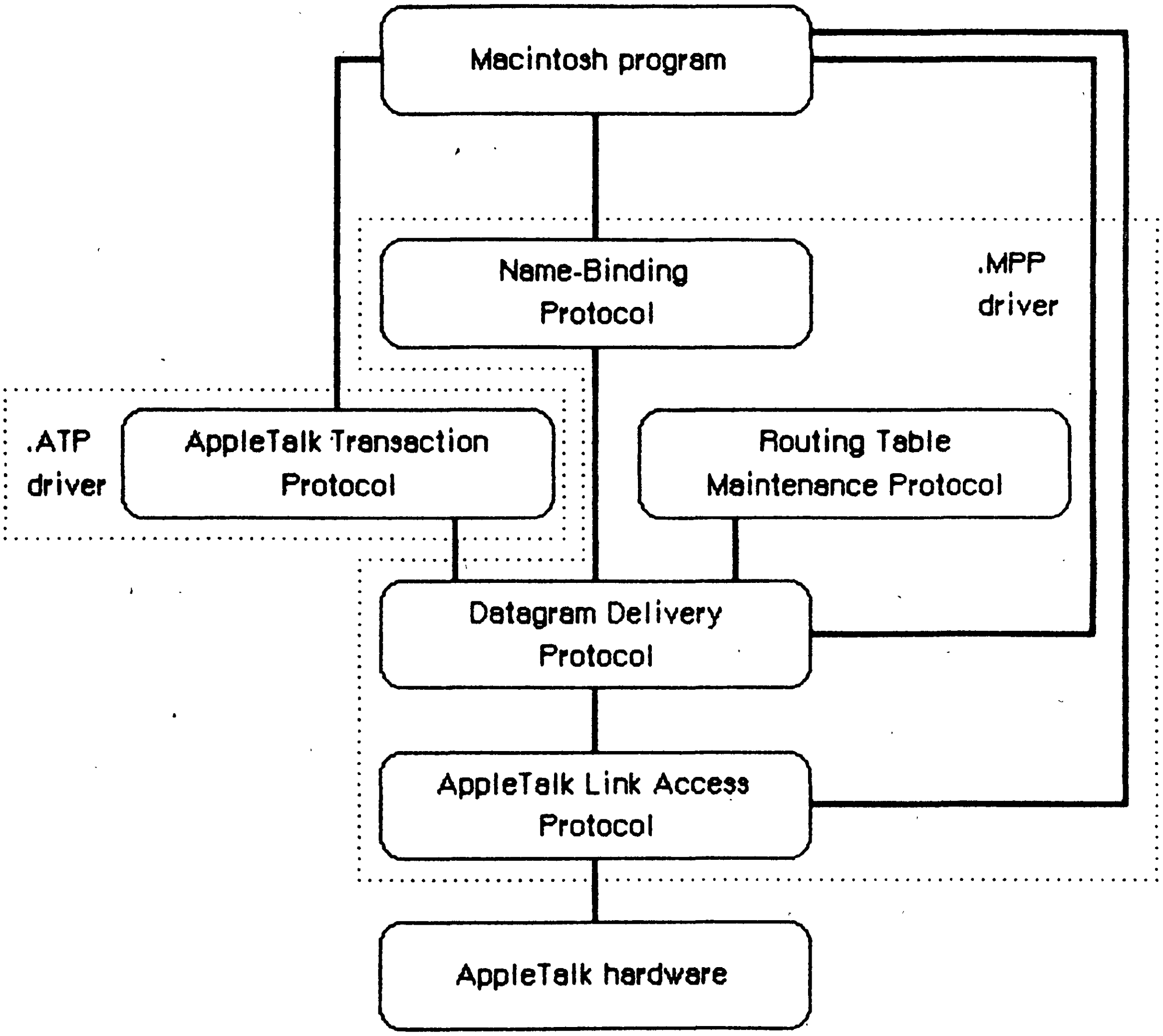

Macintosh program
Name-Binding Protocol
.MPP driver
.ATP driver
AppleTalk Transaction Protocol
Routing Table Maintenance Protocol
Datagram Delivery Protocol
AppleTalk Link Access Protocol
AppleTalk hardware

Figure 1. AppleTalk Manager Protocols

The AppleTalk Link Access Protocol (ALAP) provides the lowest-level services of the AppleTalk system. Its main function is to control access to the AppleTalk network among various competing devices. Each device connected to an AppleTalk network, known as a node, is assigned an 8-bit node ID number that identifies the node. ALAP ensures that each node on an AppleTalk network has a unique node number, assigned dynamically when the node is started up.

ALAP provides its clients with node-to-node delivery of data frames on a **single** AppleTalk network. An ALAP frame is a variable-length packet of data preceded and followed by control information referred to as the ALAP frame header and frame trailer, respectively. The ALAP frame header includes the node numbers (eight bits each) of the frame's destination and source nodes. The AppleTalk hardware uses the destination node number to deliver the frame. The frame's source node ID allows a program in the receiving node to determine the identity of the source. A sending node can ask ALAP to send a frame to all nodes on the AppleTalk; this broadcast service is obtained by specifying a destination node number of 255.

ALAP can have multiple clients in a single node. When a frame arrives at a node, ALAP determines which client it should be delivered to by reading the frame's LAP protocol type. The LAP protocol type is an 8-bit quantity, contained in the frame's header, that identifies the LAP client to whom the frame will be sent. ALAP calls the client's protocol handler, which is a software process in the node that reads in and then services the frames. The protocol handlers for a node are listed in a protocol handler table.

An ALAP frame trailer contains a 16-bit frame check sequence generated by the AppleTalk hardware. The receiving node uses the frame check sequence to detect transmission errors, and discards frames with errors. In effect, a frame with an error is "lost" in the AppleTalk network, because ALAP doesn't attempt to recover from errors by requesting the sending node to retransmit such frames. Thus ALAP is said to make a "best effort" to deliver frames, without any guarantee of delivery.

An ALAP frame can contain up to 600 bytes of client data. The first two bytes must be an integer equal to the length of the client data (including the length bytes themselves).

Datagram Delivery Protocol (DDP) provides the next-higher level protocol in the AppleTalk architecture, managing socket-to-socket delivery of datagrams over AppleTalk internets. DDP is an ALAP client, and uses the node-to-node delivery service provided by ALAP to send and receive datagrams. Datagrams are packets of data transmitted by DDP. A DDP datagram can contain up to 586 bytes of client data. Sockets are logical entities within the nodes of a network; each socket within a given node has a unique 8-bit socket number.

On a single AppleTalk network, a socket is uniquely identified by its AppleTalk address--its socket number together with its node number. To identify a socket in the scope of an AppleTalk internet, the socket's

AppleTalk address and network number are needed. Internets are formed by interconnecting AppleTalk networks via intelligent nodes called bridges. A network number is a 16-bit number that uniquely identifies a network in an internet. A socket's AppleTalk address together with its network number provide an internet-wide unique socket identifier called an internet address.

Sockets are owned by socket clients, which typically are software processes in the node. Socket clients include code called the socket listener, which receives and services datagrams addressed to that socket. Socket clients must open a socket before datagrams can be sent or received through it. Each node contains a socket table that lists the listener for each open socket.

A datagram is sent from its source socket through a series of AppleTalk networks, being passed on from bridge to bridge, until it reaches its destination network. The ALAP in the destination network then delivers the datagram to the node containing the destination socket. Within that node the datagram is received by ALAP calling the DDP protocol handler, and by the DDP protocol handler in turn calling the destination socket listener, which for most applications will be a higher-level protocol such as the AppleTalk Transaction Protocol. You can't send a datagram between two sockets in the same node.

Bridges on AppleTalk internets use the Routing Table Maintenance Protocol (RTMP) to maintain routing tables for routing datagrams through the internet. In addition, nonbridge nodes use RTMP to determine the number of the network to which they're connected and the node number of one bridge on their network. The RTMP code in nonbridge nodes contains only a subset of RTMP (the RTMP stub), and is a DDP client owning socket number 1 (the RTMP socket).

Socket clients are also known as network-visible entities, because they're the primary accessible entities on an internet. Network-visible entities can choose to identify themselves by an entity name, an identifier of the form

    object:type@zone

Each of the three fields of this name is an alphanumeric string of up to 32 characters. The object and type fields are arbitrary identifiers assigned by a socket client, to provide itself with a name and type descriptor (for example, abs:Mailbox). The zone field identifies the zone in which the socket client is located; a zone is an arbitrary subset of AppleTalk networks in an internet. A socket client can identify itself by as many different names as it chooses. These aliases are all treated as independent identifiers for the same socket client.

The Name-Binding Protocol (NBP) maintains a names table in each node that contains the name and internet address of each entity in **that** node. These name-address pairs are called NBP tuples. The collection of names tables in an internet is known as the names directory.

NBP allows its clients to add or delete their name-address tuples from the node's names table. It also allows its clients to obtain the internet addresses of entities from their names. This latter operation, known as *name lookup* (in the names directory), requires that NBP install itself as a DDP client and broadcast special name-lookup packets to the nodes in a specified zone. These datagrams are sent by NBP to the *names information socket*--socket number 2 in every node using NBP.

NBP clients can use special meta-characters in place of one or more of the three fields of the name of an entity it wishes to look up. The character "=" in the object or type field signifies "all possible values". The zone field can be replaced by "*", which signifies "this zone"--the zone in which the NBP client's node is located. For example, an NBP client performing a lookup with the name

```
=:Mailbox@*
```

will obtain in return the entity names and internet addresses of all mailboxes in the client's zone (excluding the client's own names and addresses). The client can specify whether one or all of the matching names should be returned.

NBP clients specify how thorough a name lookup should be by providing NBP with the number of times (*retry count*) that NBP should broadcast the lookup packets and the time interval (*retry interval*) between these retries.

As noted above, ALAP and DDP provide "best effort" delivery services with no recovery mechanism when packets are lost or discarded because of errors. Although for many situations such a service suffices, the *AppleTalk Transaction Protocol* (ATP) provides a reliable loss-free transport service. ATP uses *transactions*, consisting of a *transaction request* and a *transaction response*, to deliver data reliably. Each transaction is assigned a 16-bit *transaction ID* number to distinguish it from other transactions. A transaction request is retransmitted by ATP until a complete response has been received, thus allowing for recovery from packet-loss situations. The retry interval and retry count are specified by the ATP client sending the request.

Although transaction requests must be contained in a single datagram, transaction responses can consist of as many as eight datagrams. Each datagram in a response is assigned a *sequence number* from Ø to 7, to indicate its ordering within the response.

ATP is a DDP client, and uses the services provided by DDP to transmit requests and responses. ATP supports both *at-least-once* and *exactly-once* transactions. Four of the bytes in an ATP header, called the *user bytes*, are provided for use by ATP's clients--they're ignored by ATP.

ATP's transaction model and means of recovering from datagram loss are covered in detail under "AppleTalk Transaction Protocol" below.

APPLETALK TRANSACTION PROTOCOL

This section covers ATP in greater depth, providing more detail about three of its fundamental concepts: transactions, buffer allocation, and recovery of lost datagrams.

Transactions

A transaction is a interaction between two ATP clients, known as the requester and the responder. The requester calls the .ATP driver in its node to send a transaction request (TReq) to the responder, and then awaits a response. The TReq is received by the .ATP driver in the responder's node and is delivered to the responder. The responder then calls its .ATP driver to send back a transaction response (TResp), which is received by the requester's .ATP driver and delivered to the requester. Figure 2 illustrates this process.

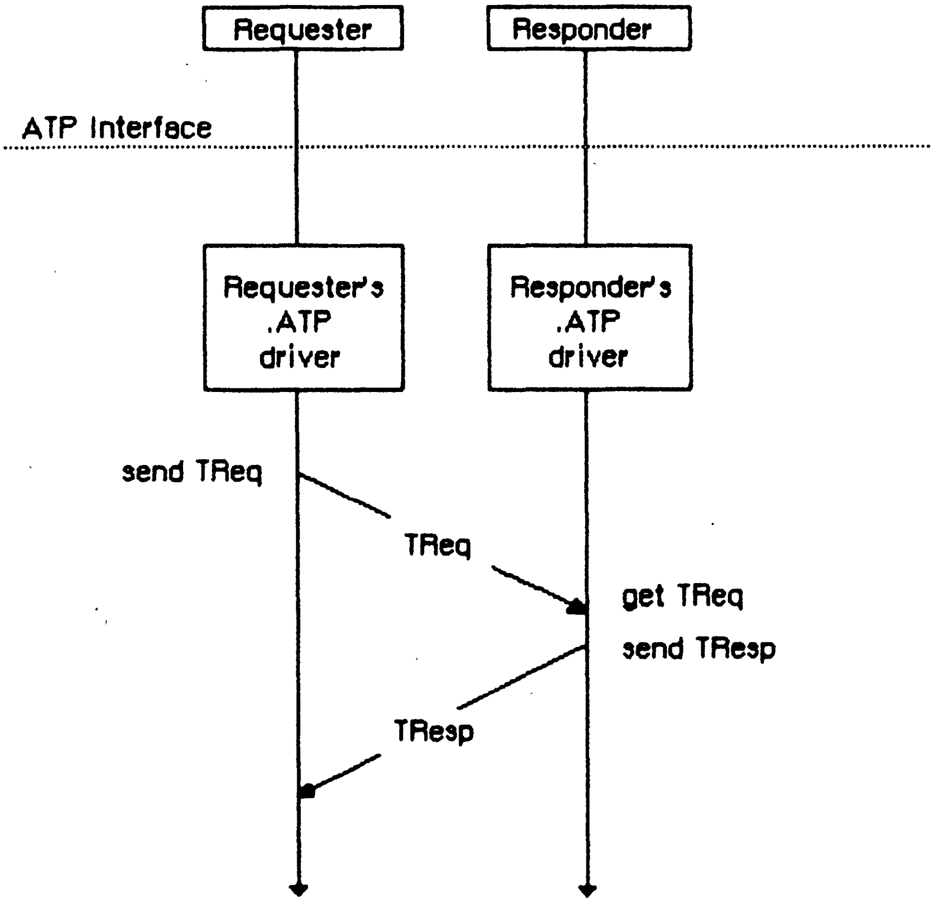


Figure 2. Transaction Process

Simple examples of transactions are:

- read a counter, reset it, and send back the value read
- read six sectors of a disk and send back the data read
- write the data sent in the TReq to a printer

A basic assumption of the transaction model is that the amount of ATP data sent in the TReq specifying the operation to be performed is small enough to fit in a single datagram. A TResp, on the other hand, may span several datagrams, as in the second example. Thus, a TReq is a single datagram, while a TResp consists of up to eight datagrams, each of which is assigned a sequence number from Ø to 7 to indicate its position in the response.

The requester must, before calling for a TReq to be sent, set aside enough buffer space to receive the datagram(s) of the TResp. The number of buffers allocated (in other words, the maximum number of datagrams that the responder can send) is indicated in the TReq by an eight-bit bit map. The bits of this bit map are numbered Ø to 7 (the least significant bit being number Ø); each bit corresponds to the response datagram with the respective sequence number.

Datagram Loss Recovery

The way that ATP recovers from datagram loss situations is best explained by an example; see Figure 3. Assume that the requester wants to read six sectors of 512 bytes each from the responder's disk. The requester puts aside six 512-byte buffers (which may or may not be contiguous) for the response datagrams, and calls ATP to send a TReq. In this TReq the bit map is set to binary ØØ111111 or decimal 63. The TReq carries a 16-bit transaction ID, generated by the requester's .ATP driver before sending it. (This example assumes that the fact that each buffer can hold 512 bytes has already been agreed upon by the requester and responder.) The TReq is delivered to the responder, which reads the six disk sectors and sends them back, through ATP, in TResp datagrams bearing sequence numbers Ø through 5. Each TResp datagram also carries exactly the same transaction ID as the TReq to which they're responding.

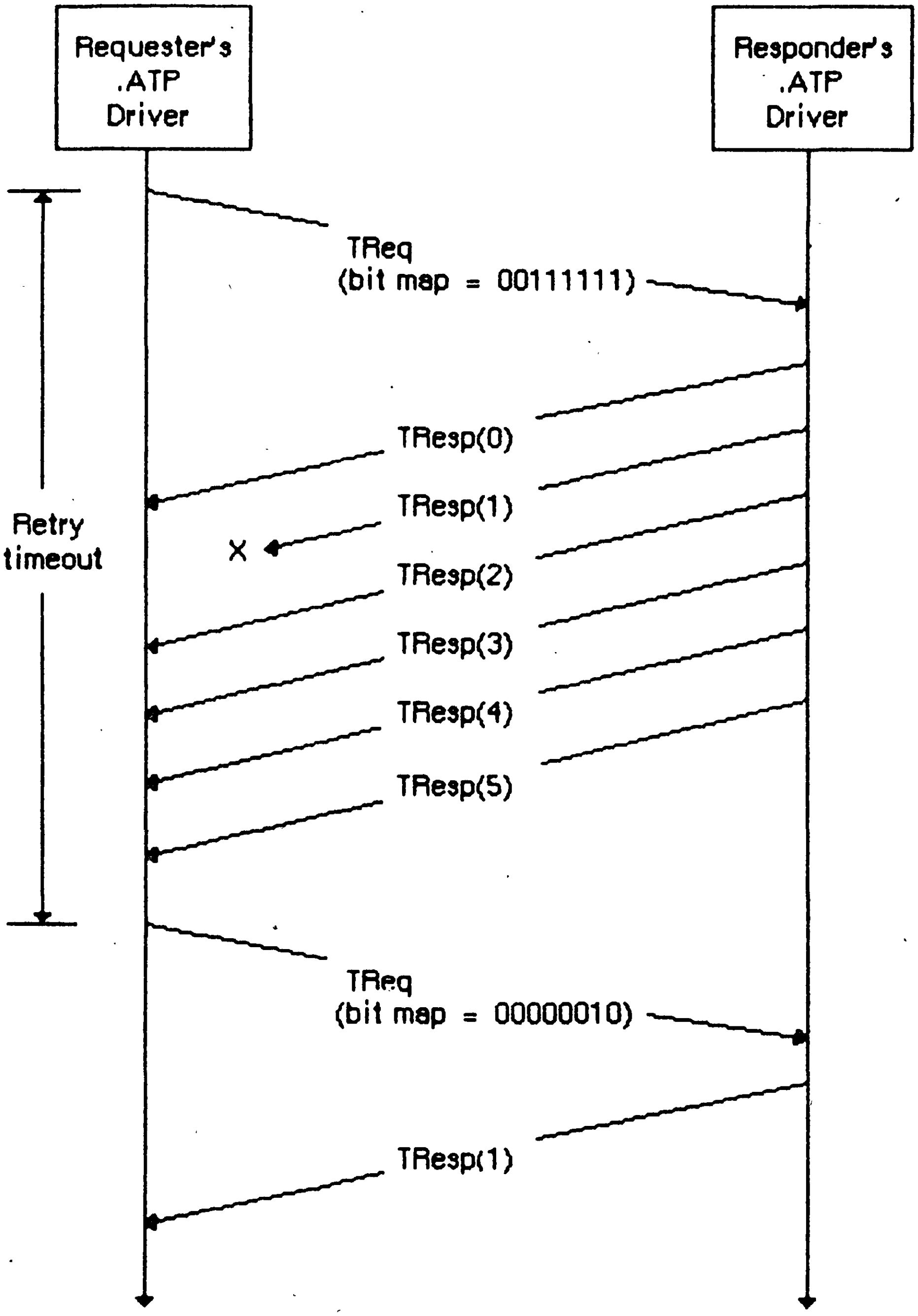


Figure 3. Datagram Loss Recovery

There are several ways that datagrams may be lost in this case. The original TReq could be lost for one of many reasons. The responding node might be too busy to receive the TReq or might be out of buffers for receiving it, there could be an undetected collision on the network, a bit error in the transmission line, and so on. To recover from such errors, the requester's .ATP driver maintains an ATP retry timer for each transaction sent. If this timer expires and the complete TResp has not been received, the TReq is retransmitted and the retry timer is restarted.

A second error situation occurs when one or more of the TResp datagrams is not received correctly by the requester's .ATP driver (datagram 1 in Figure 3). Again, the retry timer will expire and the complete TResp will not have been received; this will result in a retransmission of the TReq. However, to avoid unnecessary retransmission of the TResp datagrams already properly received, the bit map of this retransmitted TReq is modified to reflect only those datagrams not yet received. Upon receiving this TReq, the responder retransmits only the missing response datagrams.

Another possible failure is that the responder's .ATP driver goes down or the responder becomes unreachable through the underlying network system. In this case, retransmission of the TReq could continue indefinitely. To avoid this situation, the requester provides a maximum retry count; if this count is exceeded, the requester's .ATP driver returns an appropriate error message to the requester.

(note)

> There may be situations where, due to an anticipated delay, you'll want a request to be retransmitted more than 255 times; specifying a retry count of 255 indicates "infinite retries" to ATP and will cause a message to be retransmitted until the request has either been serviced, or been cancelled through a specific call.

Finally, in our example, what if the responder is able to provide only four disk sectors (having reached the end of the disk) instead of the six requested? To handle this situation, there's an end-of-message (EOM) flag in each TResp datagram. In this case, the TResp datagram numbered 3 would come with this flag set. The reception of this datagram informs the requester's .ATP driver that TResps numbered 4 and 5 will not be sent and should not be expected.

When the transaction completes successfully (all expected TResp datagrams are received or TResp datagrams numbered Ø to n are received with datagram n's EOM flag set), the requester is informed and can then use the data received in the TResp.

ATP provides two classes of service: at-least-once (ALO) and exactly-once (XO). The TReq datagram contains an XO flag that's set if XO service is required and cleared if ALO service is adequate. The main difference between the two is in the sequence of events that occurs when the TReq is received by the responder's .ATP driver.

In the case of ALO service, each time a TReq is received (with the XO flag cleared), it's delivered to the responder by its .ATP driver; this is true even for retransmitted TReqs of the same transaction. Each time the TReq is delivered, the responder performs the requested operation and sends the necessary TResp datagrams. Thus, the requested operation is performed at least once, and perhaps several times, until the transaction is completed at the requester's end.

The at-least-once service is satisfactory in a variety of situations. For instance, if the requester wishes to read a clock or a counter being maintained at the responder's end. However, in other circumstances, repeated execution of the requested operation is unacceptable. This is the case, for instance, if the requester is sending data to be printed at the responding end; it's for such situations that exactly-once service is designed.

The responder's .ATP driver maintains a transactions list of recently received XO TReqs. Whenever a TReq is received with its XO flag set, the driver goes through this list to see if this is a retransmitted TReq. If it's the first TReq of a transaction, it's entered into the list and delivered to the responder. The responder executes the requested operation and calls its driver to send a TResp. Before sending it out, the .ATP driver **saves** the TResp in the list.

When a retransmitted TReq for the same XO transaction is received, the responder's .ATP driver will find a corresponding entry in the list. The retransmitted TReq is **not** delivered to the responder; instead the driver automatically retransmits the response datagrams that were saved in the list. In this way, the responder never sees the retransmitted TReqs and the requested operation is performed only once.

ATP must include a mechanism for eventually removing XO entries from the responding end's transaction list; two provisions are made for this. When the requester's .ATP driver has received all the TResp datagrams of a particular transaction, it sends a datagram known as a transaction release (TRel); this tells the responder's .ATP driver to remove the transaction from the list. However, the TRel could be lost in the network (or the responding end may die, and so on), leaving the entry in the list forever. To account for this situation, the responder's .ATP driver maintains a release timer for each transaction. If this timer expires and no activity has occurred for the transaction, its entry is removed from the transaction list.

ABOUT THE APPLETALK MANAGER

The AppleTalk Manager is divided into three parts (see Figure 4):

- A lower-level driver called ".MPP" that contains code to implement ALAP, DDP, NBP, and the RTMP stub; this includes separate code resources loaded in when an NBP name is registered or looked up.

- A higher-level driver called ".ATP" that implements ATP.

- A Pascal interface to these two drivers, which is a set of Pascal data types and routines to aid Pascal programmers in calling the AppleTalk Manager.

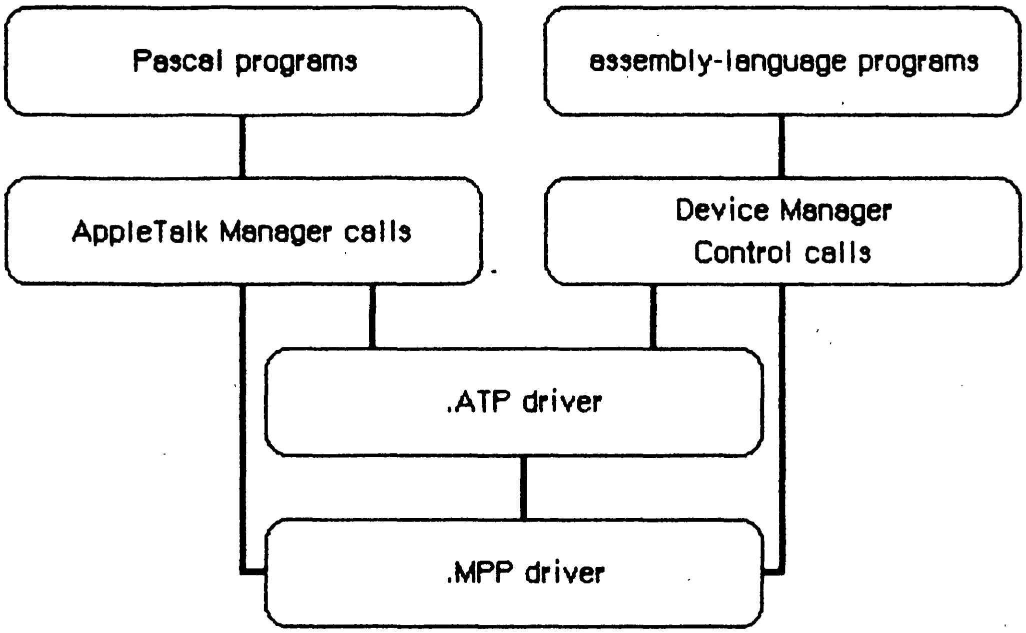


Figure 4. Calling the AppleTalk Manager

Pascal programmers make calls to the AppleTalk Manager's Pascal interface, which in turn calls the two drivers. Assembly-language programmers make Device Manager Control calls directly to the drivers.

(note)
Pascal programmers can, of course, make PBControl calls directly if they wish.

The AppleTalk Manager provides ALAP routines that allow a program to:

- send a frame to another node
- receive a frame from another node
- add a protocol handler to the protocol handler table
- remove a protocol handler from the protocol handler table

Each node may have up to four protocol handlers in its protocol handler table, two of which are currently used by DDP.

By calling DDP, socket clients can:

- send a datagram via a socket
- receive a datagram via a socket
- open a socket and add a socket listener to the socket table

- close a socket and remove a socket listener from the socket table

Each node may have up to 12 open sockets in its socket table.

Programs cannot access RTMP directly via the AppleTalk Manager; RTMP exists solely for the purpose of providing DDP with routing information.

The NBP code allows a socket client to:

- register the name and socket number of an entity in the node's names table

- determine the address (and confirm the existence) of an entity

- delete the name of an entity from the node's names table

The AppleTalk Manager's .ATP driver allows a socket client to do the following:

- open a responding socket to receive requests

- send a request to another socket and get back a response

- receive a request via a responding socket

- send a response via a responding socket

- close a responding socket

(note)

> Although the AppleTalk Manager provides four different protocols for your use, you're not bound to use all of them. In fact, most programmers will use only the NBP and ATP protocols.

AppleTalk communicates via channel B of the Serial Communications Controller (SCC). When the Macintosh is started up with a disk containing the AppleTalk code, the status of serial port B is checked. If port B isn't being used by another device driver, and is available for use by AppleTalk, the .MPP driver is loaded into the system heap. On a Macintosh 128K, only the MPP code is loaded at system startup; the .ATP driver and NBP code are read into the application heap when the appropriate commands are issued. On a Macintosh 512K or XL, all AppleTalk code is loaded into the system heap at system startup.

After loading the AppleTalk code, the .MPP driver installs its own interrupt handlers, installs a task into the vertical retrace queue, and prepares the SCC for use. It then chooses a node ID for the Macintosh and confirms that the node ID isn't already being used by another node on the network.

(warning)

> For this reason it's imperative that the Macintosh be connected to the AppleTalk network through serial port B (the printer port) **before** being switched on.

The AppleTalk Manager also provides Pascal routines for opening and closing the .MPP and .ATP drivers. The open calls allow a program to load AppleTalk code at other than system startup. The close calls allow a program to remove the AppleTalk code from the Macintosh; the use of close calls is highly discouraged, since other co-resident programs are then "disconnected" from AppleTalk. Both sets of calls are described in detail in "Calling the AppleTalk Manager from Pascal" below.

(warning)

> If, at system startup, serial port B isn't available for use by AppleTalk, the .MPP driver won't open. However, a driver doesn't return an error message when it fails to open. Pascal programmers must ensure the proper opening of AppleTalk by calling one of the two routines for opening the AppleTalk drivers (either MPPOpen or ATPLoad). If AppleTalk was successfully loaded at system startup, these calls will have no effect; otherwise they'll check the availability of port B, attempt to load the AppleTalk code, and return an appropriate result code.

Assembly-language note: Assembly-language programmers can use the Pascal routines for opening AppleTalk. They can also check the availability of port B themselves and then decide whether to open MPP or ATP. Detailed information on how to do this is provided in the section "Calling the AppleTalk Manager from Assembly Language" below.

CALLING THE APPLETALK MANAGER FROM PASCAL

This section discusses how to use the AppleTalk Manager from Pascal. Equivalent assembly-language information is given in the next section.

Many Pascal calls to the AppleTalk Manager require information passed in a data structure of type ABusRecord. The exact content of an ABusRecord depends on the protocol being called:

```
TYPE ABProtoType = (lapProto,ddpProto,nbpProto,atpProto);

     ABusRecord  = RECORD
                     abOpcode:        ABCallType; {type of call}
                     abResult:        INTEGER;    {result code}
                     abUserReference: LONGINT;    {for your use}
                     CASE ABProtoType OF
                       lapProto:
                         . . . {ALAP parameters}
                       ddpProto:
                         . . . {DDP parameters}
                       nbpProto:
                         . . . {NBP parameters}
                       atpProto:
                         . . . {ATP parameters}
                     END;
                   END;

     ABRecPtr    = ^ABusRecord;
     ABRecHandle = ^ABRecPtr;
```

The value of the abOpcode field is inserted by the AppleTalk Manager when the call is made, and is always a member of the following set:

```
TYPE ABCallType = (tLAPRead,tLAPWrite,tDDPRead,tDDPWrite,
                   tNBPLookup,tNBPConfirm,tNBPRegister,
                   tATPSndRequest,tATPGetRequest,tATPSndRsp,
                   tATPAddRsp,tATPRequest,tATPRespond);
```

The abUserReference field is available for use by the calling program in any way it wants. This field isn't used by the AppleTalk Manager routines or drivers.

The size of an ABusRecord data structure in bytes is given by one of the following constants:

```
CONST lapSize = 2Ø;
      ddpSize = 26;
      nbpSize = 26;
      atpSize = 56;
```

Variables of type ABusRecord must be allocated on the heap with Memory Manager NewHandle calls. For example:

```
myABRecord := ABRecHandle(NewHandle(ddpSize))
```

(warning)
These Memory Manager calls can't be made inside interrupts.

Most AppleTalk Manager routines return a result code of type OSErr. Each routine description lists all of the applicable result codes generated by the AppleTalk Manager, along with a short description of what the result code means. If no error occurred, it returns the result code noErr. Lengthier explanations of all the result codes can be found in the summary at the end of the manual. Result codes from other parts of the Operating System may also be returned. (See the Operating System Utilities manual for a list of all result codes.)

Many AppleTalk Manager routines can be executed either synchronously (meaning that the application can't continue until the routine is completed) or asynchronously (meaning that the application is free to perform other tasks while the routine is being executed).

When you call an AppleTalk Manager routine asynchronously, an I/O request is placed in the appropriate driver's I/O queue, and control returns to the calling program--possibly even before the actual I/O is completed. Requests are taken from the queue, one at a time, and processed; meanwhile, the calling program is free to work on other things.

The routines that can be executed asynchronously contain a Boolean parameter called async. If async is TRUE, the call is executed asynchronously; otherwise the call is executed synchronously. Every time an asynchronous routine call is completed, the AppleTalk Manager posts a network event. The message field of the event record will contain a handle to the ABusRecord that was used to make that call.

Routines that are executed asynchronously return control to the calling program with noErr as soon as the call is placed in the driver's I/O queue; this isn't an indication of successful call completion. It simply indicates that the call was successfully queued to the appropriate driver. To determine when the call is actually completed, you can either check for a network event or poll the abResult field of the call's ABusRecord. The abResult field, set to 1 when the call is made, receives the appropriate result code upon completion of the call.

(warning)

> Since a data structure of type ABusRecord is often used by the AppleTalk Manager during an asynchronous call, it's locked by the AppleTalk Manager. Don't attempt to unlock or use such a variable.

Each routine description includes a list of the ABusRecord fields affected by the routine. The arrow next to each field name indicates whether it's an input, output, or input/output parameter:

Arrow	Meaning
-->	Parameter must be passed to the routine
<--	Parameter will be returned by the routine
<->	Parameter must be passed to and will be returned by the routine

Opening and Closing AppleTalk

FUNCTION MPPOpen : OSErr; [Not in ROM]

MPPOpen first checks whether the .MPP driver is already loaded; if it is, MPPOpen does nothing and returns noErr. If MPP hasn't been loaded, MPPOpen attempts to load it into the system heap. If it succeeds, it then initializes the driver's variables and goes through the process of dynamically assigning a node ID to that Macintosh. On a Macintosh 512K or XL, it also loads the .ATP driver and NBP code into the system heap.

If serial port B isn't configured for AppleTalk, or is already in use, the .MPP driver isn't loaded and an appropriate result code is returned.

Result codes	noErr	No error
	portInUse	Port B is already in use
	portNotCf	Port B not configured for AppleTalk

FUNCTION MPPClose : OSErr; [Not in ROM]

MPPClose removes the .MPP driver, and any data structures associated with it, from memory. If the .ATP driver or NBP code were also installed, they'll also be removed. MPPClose also returns the use of port B to the Serial Driver.

(warning)
> Since other co-resident programs may be using AppleTalk, it's strongly recommended that you never use this call. MPPClose will completely disable AppleTalk; the only way to restore AppleTalk is to call MPPOpen again.

AppleTalk Link Access Protocol

Data Structures

ALAP calls use the following ABusRecord fields:

```
lapProto:
  (lapAddress:  LAPAdrBlock; {destination or source node ID}
   lapReqCount: INTEGER;     {length of frame data or }
                             { buffer size in bytes}
   lapActCount: INTEGER;     {number of frame data bytes }
                             { actually received}
   lapDataPtr:  Ptr);        {pointer to frame data or }
                             { pointer to buffer}
```

When an ALAP frame is sent, the lapAddress field indicates the ID of the destination node. When an ALAP frame is received, lapAddress returns the ID of the source node. The lapAddress field also indicates the LAP protocol type of the frame:

```
TYPE LAPAdrBlock = PACKED RECORD
                     dstNodeID:   Byte;  {destination node ID}
                     srcNodeID:   Byte;  {source node ID}
                     lapProtType: ABByte {LAP protocol type}
                   END;
```

When an ALAP frame is sent, lapReqCount indicates the size of the frame data in bytes and lapDataPtr points to a buffer containing the frame data to be sent. When an ALAP frame is received, lapDataPtr points to a buffer in which the incoming data can be stored and lapReqCount indicates the size of the buffer in bytes. The number of bytes actually sent or received is returned in the lapActCount field.

Each ALAP frame contains an 8-bit LAP protocol type in the header. LAP protocol types 128 through 255 are reserved for internal use by ALAP, hence the declaration:

```
TYPE ABByte = 1..127; {LAP protocol type}
```

(warning)

> Don't use LAP protocol type values 1 and 2; they're reserved for use by DDP. Value 3 through 15 are reserved for internal use by Apple and also shouldn't be used.

Using ALAP

Most programs will never need to call ALAP, because higher-level protocols will automatically call it as necessary. If you do want to send a frame directly via ALAP, call the LAPWrite function. If you want to read ALAP frames, you have two choices:

- Call LAPOpenProtocol with NIL for protoPtr (see below); this installs the default protocol handler provided by the AppleTalk Manager. Then call LAPRead to receive frames.

- Write your own protocol handler, and call LAPOpenProtocol to add it to the node's protocol handler table. The ALAP code will examine every incoming frame and send all those with the correct LAP protocol type to your protocol handler. See the section "Protocol Handlers and Socket Listeners" for information on how to write a protocol handler.

When your program no longer wants to receive frames with a particular LAP protocol type value, it can call LAPCloseProtocol to remove the corresponding protocol handler from the protocol handler table.

ALAP Routines

FUNCTION LAPOpenProtocol (theLAPType: ABByte; protoPtr: Ptr) : OSErr;
[Not in ROM}

LAPOpenProtocol adds the LAP protocol type specified by theLAPType to the node's protocol table. If you provide a pointer to a protocol handler in protoPtr, ALAP will send each frame with a LAP protocol type of theLAPType to that protocol handler.

If protoPtr is NIL, the default protocol handler will be used for receiving frames with a LAP protocol type of theLAPType. In this case, to receive a frame you must call LAPRead to provide the default protocol handler with a buffer for placing the data. If, however, you've written your own protocol handler and protoPtr points to it, your protocol handler will have the responsibility for receiving the frame and it's not necessary to call LAPRead.

Result codes		
	noErr	No error
	lapProtErr	Error attaching protocol type

FUNCTION LAPCloseProtocol (theLAPType: ABByte) : OSErr; [Not in ROM]

LAPCloseProtocol removes from the node's protocol table the specified LAP protocol type, as well as its protocol handler.

(warning)

> Don't close LAP protocol type values 1 or 2. If you close these protocol types, DDP will be disabled; once disabled, the only way to restore DDP is to reboot, or to close and then reopen AppleTalk.

Result codes		
	noErr	No error
	lapProtErr	Error detaching protocol type

```
FUNCTION LAPWrite (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
          [Not in ROM]
```

ABusRecord

<--	abOpcode	{always tLAPWrite}
<--	abResult	{result code}
-->	abUserReference	{for your use}
-->	lapAddress.dstNodeID	{destination node ID}
-->	lapAddress.lapProtType	{LAP protocol type}
-->	lapReqCount	{length of frame data}
-->	lapDataPtr	{pointer to frame data}

LAPWrite sends a frame to another node. LAPReqCount and lapDataPtr specify the length and location of the data to send. The lapAddress.lapProtType field indicates the LAP protocol type of the frame and the lapAddress.dstNodeID indicates the node ID of the node to which the frame should be sent.

(note)

The first two bytes of an ALAP frame's data must contain the length in bytes of that data, including the length bytes themselves.

Result codes		
	noErr	No error
	excessCollsns	Unable to contact destination node; packet not sent

FUNCTION LAPRead (abRecord: ABRecHandle; async: BOOLEAN) : OSErr; [Not in ROM]

ABusRecord

<--	abOpcode	{always tLAPRead}
<--	abResult	{result code}
-->	abUserReference	{for your use}
<--	lapAddress.dstNodeID	{packet's destination node ID}
<--	lapAddress.srcNodeID	{packet's source node ID}
-->	lapAddress.lapProtType	{LAP protocol type}
-->	lapReqCount	{buffer size in bytes}
<--	lapActCount	{number of frame data bytes actually } { received}
-->	lapDataPtr	{pointer to buffer}

LAPRead receives a frame from another node. LAPReqCount and lapDataPtr specify the length and location of the buffer that will receive the frame data. If the buffer isn't large enough to hold all of the incoming frame data, the extra bytes will be discarded and buf2SmallErr will be returned. The number of bytes actually received is returned in lapActCount. Only frames with LAP protocol type equal to lapAddress.lapProtType will be received. The node number of the frame's source and destination nodes are returned in lapAddress.srcNodeID and lapAddress.dstNodeID respectively. You can determine if the packet was broadcast to you by examining the value of lapAddress.dstNodeID--if the packet was broadcast it's equal to 255, otherwise it's equal to your node ID.

(note)

You should issue LAPRead calls only for LAP protocol types that were opened (via LAPOpenProtocol) to use the default protocol handler.

Result codes		
	noErr	No error
	buf2SmallErr	Frame too large for buffer
	readQErr	Invalid protocol type or protocol type not found in table

FUNCTION LAPRdCancel (abRecord: ABRecHandle) : OSErr; [Not in ROM]

Given the handle to the ABusRecord of a previously made LAPRead call, LAPRdCancel dequeues the LAPRead call, provided that a packet satisfying the LAPRead has not already arrived. LAPRdCancel returns noErr if the LAPRead call is successfully removed from the queue. If LAPRdCancel returns recNotFnd, check the abResult field to verify that the LAPRead has been completed and determine its outcome.

Result codes		
	noErr	No error
	readQErr	Invalid protocol type or protocol type not found in table
	recNotFnd	ABRecord not found in queue

Example

This example sends a LAP packet synchronously and waits asynchronously for a response. Assume that both nodes are using a known protocol type (in this case, 73) to receive packets, and that the destination node has a node ID of 4.

```
VAR myABRecord: ABRecHandle;
    myBuffer: PACKED ARRAY[Ø..599] OF CHAR;  {buffer for both send and }
                                             { receive}
    myLAPType: Byte;
    errCode,index,dataLen: INTEGER;
    someText: Str255;
    async: BOOLEAN;

BEGIN
errCode := MPPOpen;
IF errCode <> noErr
  THEN
    WRITELN('Error in opening AppleTalk')
    {Maybe serial port B isn't available for use by AppleTalk}
  ELSE
    BEGIN
    {Call Memory Manager to allocate ABusRecord}
    myABRecord := ABRecHandle(NewHandle(lapSize));
    myLAPType := 73;
    {Enter myLAPType into protocol handler table and install default }
    { handler to service frames of that LAP type.  No packets of }
    { that LAP type will be received until we call LAPRead.}
    errCode := LAPOpenProtocol(myLAPType,NIL);
    IF errCode <> noErr
      THEN
        WRITELN('Error while opening the protocol type')
        {Have we opened too many protocol types?  Remember that DDP }
        { uses two of them.}
      ELSE
        BEGIN
        {Prepare data to be sent}
        someText := 'This data will be in the LAP data area';
        {The .MPP implementation requires that the first two bytes}
        { of the LAP data field contain the length of the data, }
        { including the length bytes themselves.}
        dataLen := LENGTH(someText)+2;
        buffer[Ø] := CHR(dataLen DIV 256); {high byte of data length}
        buffer[1] := CHR(dataLen MOD 256); {low byte of data length}
        FOR index := 1 TO dataLen-2 DO   {stuff buffer with packet data}
          buffer[index+1] := someText[index];
        async := FALSE;
        WITH myABRecord^^ DO   {fill parameters in the ABusRecord}
          BEGIN
          lapAddress.lapProtType := myLAPType;
          lapAddress.dstNodeID := 4;
          lapReqCount := dataLen;
          lapDataPtr := @buffer;
          END;
        {Send the frame}
        errCode := LAPWrite(myABRecord,async);
        {In the case of a sync call, errCode and the abResult }
        { field of the myABRecord will contain the same result }
        { code. We can also reuse myABRecord, since we know }
        { whether the call has completed.}
        IF errCode <> noErr
          THEN
            WRITELN('Error while writing out the packet')
            {Maybe the receiving node wasn't on-line}
          ELSE
            BEGIN
            {We have sent out the packet and are now waiting for a }
            { response. We issue an async LAPRead call so that we }
            { don't "hang" waiting for a response that may not come.}
            async := TRUE;
            WITH myABRecord^^ DO
              BEGIN
              lapAddress.lapProtType := myLAPType; {LAP type we want }
                                                   { to receive}
              lapReqCount := 6ØØ;  {our buffer is maximum size}
              lapDataPtr := @buffer;
              END;
            errCode := LAPRead(myABRecord,async);  {wait for a packet}
            IF errCode <> noErr
              THEN
                WRITELN('Error while trying to queue up a LAPRead')
                {Was the protocol handler installed correctly?}
              ELSE
                BEGIN
                {We can either sit here in a loop and poll the abResult }
                { field or just exit our code and use the event }
                { mechanism to flag us when the packet arrives.}
                CheckForMyEvent;   {your procedure for checking for a }
                                   { network event}
                errCode := LAPCloseProtocol(myLAPType);
                IF errCode <> noErr
                  THEN
                    WRITELN('Error while closing the protocol type');
                END;
            END;
        END;
    END;
END.
```

Datagram Delivery Protocol

Data Structures

DDP calls use the following ABusRecord fields:

```
        ddpProto:
          (ddpType:     Byte;      {DDP protocol type}
           ddpSocket:   Byte;      {source or listening socket number}
           ddpAddress:  AddrBlock; {destination or source socket address}
           ddpReqCount: INTEGER;   {length of datagram data or }
                                   { buffer size in bytes}
           ddpActCount: INTEGER;   {number of bytes actually received}
           ddpDataPtr:  Ptr;       {pointer to buffer}
           ddpNodeID:   Byte);     {original destination node ID}
```

When a DDP datagram is sent, ddpReqCount indicates the size of the datagram data in bytes and ddpDataPtr points to a buffer containing the datagram data. DDPSocket specifies the socket from which the datagram should be sent. DDPAddress is the internet address of the socket to which the datagram should be sent:

```
       TYPE AddrBlock = PACKED RECORD
                          aNet:    INTEGER; {network number}
                          aNode:   Byte;    {node ID}
                          aSocket: Byte     {socket number}
                        END;
```

(note)
The network number you specify in ddpAddress.aNet tells .MPP whether to create a long header (for an internet) or a short header (for a local network only). A short DDP header will be sent if ddpAddress.aNet is Ø or equal to the network number of the local network.

When a DDP datagram is received, ddpDataPtr points to a buffer in which the incoming data can be stored and ddpReqCount indicates the size of the buffer in bytes. The number of bytes actually sent or received is returned in the ddpActCount field. DDPAddress is the internet address of the socket from which the datagram was sent.

DDPType is the DDP protocol type of the datagram, and ddpSocket specifies the socket that will receive the datagram.

(warning)
DDP protocol types 1 through 15 and DDP socket numbers 1 through 63 are reserved by Apple for internal use. Socket numbers 64 through 127 are available for experimental use. Use of these experimental sockets isn't recommended for commercial products, since there's no mechanism for eliminating conflicting usage by

different developers.

Using DDP

Before it can use a socket, the program must call DDPOpenSocket, which adds a socket and its socket listener to the socket table. When a program is finished using a socket, call DDPCloseSocket, which removes the socket's entry from the socket table. To send a datagram via DDP, call DDPWrite. To receive datagrams, you have two choices:

- Call DDPOpenSocket with NIL for sktListener (see below); this installs the default socket listener provided by the AppleTalk Manager. Then call DDPRead to receive datagrams.

- Write your own socket listener and call DDPOpenSocket to install it. DDP will call your socket listener for every incoming datagram for that socket; in this case, you shouldn't call DDPRead. For information on how to write a socket listener, see the section "Protocol Handlers and Socket Listeners".

To cancel a previously issued DDPRead call (provided it's still in the queue), call DDPRdCancel.

DDP Routines

```
FUNCTION DDPOpenSocket (VAR theSocket: Byte; sktListener: Ptr) : OSErr;
            [Not in ROM]
```

DDPOpenSocket adds a socket and its socket listener to the socket table. If theSocket is nonzero (it must be in the range of 64 to 127), it specifies the socket's number. If theSocket is Ø, DDPOpenSocket dynamically assigns a socket number in the range 128 to 254, and returns it in theSocket. SktListener contains a pointer to the socket listener; if it's NIL, the default listener will be used.

If you're using the default socket listener, you must then call DDPRead to receive a datagram (in order to specify buffer space for the default socket listener). If, however, you've written your own socket listener and sktListener points to it, your listener will provide buffers for receiving datagrams and you shouldn't use DDPRead calls.

DDPOpenSocket will return ddpSktErr if you pass the number of an already opened socket, if you pass a socket number greater than 127, or if the socket table is full.

(note)

> The range of static socket numbers 1 through 63 is reserved for use by AppleTalk. Socket numbers 64 through 127 are available for unrestricted experimental use.

Result codes	noErr	No error
	ddpSktErr	Socket error

FUNCTION DDPCloseSocket (theSocket: Byte) : OSErr; [Not in ROM]

DDPCloseSocket removes the entry of the specified socket from the socket table and cancels all pending DDPRead calls that have been made for that socket. If you pass a socket number of Ø, or if you attempt to close a socket that isn't open, DDPCloseSocket will return ddpSktErr.

Result codes	noErr	No error
	ddpSktErr	Socket error

FUNCTION DDPWrite (abRecord: ABRecHandle; doChecksum: BOOLEAN; async:
BOOLEAN) : OSErr; [Not in ROM]

ABusRecord

<--	abOpcode	{always tDDPWrite}
<--	abResult	{result code}
-->	abUserReference	{for your use}
-->	ddpType	{DDP protocol type}
-->	ddpSocket	{source socket number}
-->	ddpAddress	{destination socket address}
-->	ddpReqCount	{length of datagram data}
-->	ddpDataPtr	{pointer to buffer}

DDPWrite sends a datagram to another socket. DDPReqCount and ddpDataPtr specify the length and location of the data to send. The ddpType field indicates the DDP protocol type of the frame, and ddpAddress is the complete internet address of the socket to which the datagram should be sent. DDPSocket specifies the socket from which the datagram should be sent. Datagrams sent over the internet to a node on an AppleTalk network different from the sending node's network have an optional software checksum to detect errors that might occur inside the intermediate bridges. If doChecksum is TRUE, DDPWrite will compute this checksum; if it's FALSE, this software checksum feature is ignored.

(note)
The destination socket can't be in the same node as the program making the DDPWrite call.

Result codes	noErr	No error
	ddpLenErr	Datagram length too big
	ddpSktErr	Source socket not open

```
FUNCTION DDPRead (abRecord: ABRecHandle; retCksumErrs: BOOLEAN; async:
         BOOLEAN) : OSErr;  [Not in ROM]
```

ABusRecord

<--	abOpcode	{always tDDPRead}
<--	abResult	{result code}
-->	abUserReference	{for your use}
<--	ddpType	{DDP protocol type}
-->	ddpSocket	{listening socket number}
<--	ddpAddress	{source socket address}
-->	ddpReqCount	{buffer size in bytes}
<--	ddpActCount	{number of bytes actually received}
-->	ddpDataPtr	{pointer to buffer}
<--	ddpNodeID	{original destination node ID}

DDPRead receives a datagram from another socket. The length and location of the buffer that will receive the data are specified by ddpReqCount and ddpDataPtr, respectively. If the buffer isn't large enough to hold all of the incoming frame data, the extra bytes will be discarded and buf2SmallErr will be returned. The number of bytes actually received is returned in ddpActCount. DDPSocket specifies the socket to receive the datagram (the "listening" socket). The node to which the packet was sent is returned in ddpNodeID; if the packet was broadcast ddpNodeID will contain 255. The address of the socket that sent the packet is returned in ddpAddress. If retCksumErrs is FALSE, DDPRead will discard any packets received with an invalid checksum and inform the caller of the error. If retCksumErrs is TRUE, DDPRead will deliver all packets, regardless of whether the checksum is valid or not; it will also notify the caller when there's a checksum error.

(note)

> The sender of the datagram must be in a different node from the receiver. You should issue DDPRead calls only for receiving datagrams for sockets opened with the default socket listener; see the description of DDPOpenSocket.

(note)

> If DDPRead returns buf2SmallErr, it will deliver packets even if retCksumErrs is FALSE.

Result codes		
	noErr	No error
	buf2SmallErr	Datagram too large for buffer
	cksumErr	Checksum error
	ddpLenErr	Datagram length too big
	ddpSktErr	Socket error
	readQErr	Invalid socket or socket not found in table

FUNCTION DDPRdCancel (abRecord: ABRecHandle) : OSErr; [Not in ROM]

Given the handle to the ABusRecord of a previously made DDPRead call, DDPRdCancel dequeues the DDPRead call, provided that a packet satisfying the DDPRead hasn't already arrived. DDPRdCancel returns noErr if the DDPRead call is successfully removed from the queue. If DDPRdCancel returns recNotFnd, check the abResult field of abRecord to verify that the DDPRead has been completed and determine its outcome.

Result codes		
	noErr	No error
	readQErr	Invalid socket or socket not found in table
	recNotFnd	ABRecord not found in queue

Example

This example sends a DDP packet synchronously and waits asynchronously for a response. Assume that both nodes are using a known socket number (in this case 3Ø) to receive packets. Normally, you would want to use NBP to look up your destination's socket address.

```
VAR myABRecord: ABRecHandle;
    myBuffer: PACKED ARRAY[Ø..599] OF CHAR;  {buffer for both send }
                                             { and receive}
    mySocket: Byte;
    errCode,index,dataLen: INTEGER;
    someText: Str255;
    async,retCksumErrs,doChecksum: BOOLEAN;

BEGIN
errCode := MPPOpen;
IF errCode <> noErr
  THEN
    WRITELN('Error in opening AppleTalk')
    {Maybe serial port B isn't available for use by AppleTalk}
  ELSE
    BEGIN
    {Call Memory Manager to allocate ABusRecord}
    myABRecord := ABRecHandle(NewHandle(ddpSize));
    mySocket := 3Ø;
    {Add mySocket to socket table and install default socket }
    { listener to service datagrams addressed to that socket. }
    { No packets addressed to mySocket will be received until }
    { we call DDPRead.}
    errCode := DDPOpenSocket(mySocket,NIL);
    IF errCode <> noErr
      THEN
        WRITELN('Error while opening the socket')
        {Have we opened too many socket listeners? Remember that }
        { DDP uses two of them.}
      ELSE
        BEGIN
```

```
{Prepare data to be sent}
someText := 'This is a sample datagram';
dataLen := LENGTH(someText);
FOR index := Ø TO dataLen-1 DO  {stuff buffer with packet data}
  myBuffer[index] := someText[index+1];
async := FALSE;
WITH myABRecord^^ DO  {fill the parameters in the ABusRecord}
  BEGIN
  ddpType := 5;
  ddpAddress.aNet := Ø;  {send on "our" network}
  ddpAddress.aNode := 34;
  ddpAddress.aSocket := mySocket;
  ddpReqCount := dataLen;
  ddpDataPtr := @myBuffer;
  END;
doChecksum := FALSE;
{If packet contains a DDP long header, compute checksum and }
{ insert it into the header.}
errCode := DDPWrite(myABRecord,doChecksum,async); {send packet}
{In the case of a sync call, errCode and the abResult field of }
{ myABRecord will contain the same result code.  We can also }
{ reuse myABRecord, since we know whether the call has completed.}
IF errCode <> noErr
  THEN
    WRITELN('Error while writing out the packet')
    {Maybe the receiving node wasn't on-line}
  ELSE
    BEGIN
    {We have sent out the packet and are now waiting for a }
    { response.  We issue an async DDPRead call so that we }
    { don't "hang" waiting for a response that may not }
    { come.  To cancel the async read call, we must close }
    { the socket associated with the call or call DDPRdCancel.}
    async := TRUE;
    retCksumErrs := TRUE;  {return packets even if they }
                           { have a checksum error}
    WITH myABRecord^^ DO
      BEGIN
      ddpType := 5;    {DDP type we want to receive}
      ddpSocket := mySocket;
      ddpReqCount := 6ØØ; {our reception buffer is max size}
      ddpDataPtr := @myBuffer;
      END;
    {Wait for a packet asynchronously}
    errCode := DDPRead(myABRecord,retCksumErrs,async);
    IF errCode <> noErr
      THEN
        WRITELN('Error while trying to queue up a DDPRead')
        {Was the socket listener installed correctly?}
      ELSE
        BEGIN
        {We can either sit here in a loop and poll the abResult }
        { field or just exit our code and use the event }
        { mechanism to flag us when the packet arrives.}
```

```
                CheckForMyEvent;    {your procedure for checking for }
                                    { a network event}
                {If there were no errors, a packet is inside the array }
                { mybuffer, the length is in ddpActCount, and the }
                { address of the sending socket is in ddpAddress. }
                { Process the packet received here and report any }
                { errors.}
                errCode := DDPCloseSocket(mySocket); {we're done with it}
                IF errCode <> noErr
                  THEN
                    WRITELN('Error while closing the socket');
                END;
            END;
        END;
    END;
END.
```

AppleTalk Transaction Protocol

Data Structures

ATP calls use the following ABusRecord fields:

```
     atpProto:
       (atpSocket:    Byte;       {listening or responding socket }
                                  { number}
        atpAddress:   AddrBlock;  {destination or source socket}
                                  { address}
        atpReqCount:  INTEGER;    {request size or buffer size}
                                  { in bytes}
        atpDataPtr:   Ptr;        {pointer to buffer}
        atpRspBDSPtr: BDSPtr;     {pointer to response BDS}
        atpBitMap:    BitMapType; {transaction bit map}
        atpTransID:   INTEGER;    {transaction ID}
        atpActCount:  INTEGER;    {number of bytes actually }
                                  { received}
        atpUserData:  LONGINT;    {user bytes}
        atpXO:        BOOLEAN;    {exactly-once flag}
        atpEOM:       BOOLEAN;    {end-of-message flag}
        atpTimeOut:   Byte;       {retry timeout interval in seconds}
        atpRetries:   Byte;       {maximum number of retries}
        atpNumBufs:   Byte;       {number of elements in response BDS }
                                  { or number of response packets sent}
        atpNumRsp:    Byte;       {number of response packets }
                                  { received or sequence number}
        atpBDSSize:   Byte;       {number of elements in response }
                                  { BDS}
        atpRspUData:  LONGINT;    {user bytes sent or received }
                                  { in transaction response}
```

```
          atpRspBuf:     Ptr;         {pointer to response message }
                                      { buffer}
          atpRspSize:    INTEGER);    {size of response message buffer}
```

The socket receiving the request or sending the response is identified by atpSocket. ATPAddress is the address of either the destination or the source socket of a transaction, depending on whether the call is sending or receiving data, respectively. ATPDataPtr and atpReqCount specify the location and length of a buffer that either contains a request or will receive a request. The number of bytes actually received in a request is returned in atpActCount. ATPTransID specifies the transaction ID. The transaction bit map is contained in atpBitMap, in the form:

```
       TYPE BitMapType = PACKED ARRAY[Ø..7] OF BOOLEAN;
```

Each bit in the bit map corresponds to one of the eight possible packets in a response. For example, when a request is made for which five response packets are expected, the bit map sent is binary ØØØ11111 or decimal 31. If the second packet in the response is lost, the requesting socket will retransmit the request with a bit map of binary ØØØØØØ1Ø or decimal 2.

ATPUserData contains the user bytes of an ATP header. ATPXO is TRUE if the transaction is to be made with exactly-once service. ATPEOM is TRUE if the response packet is the last packet of a transaction. If the number of responses is less than the number that were requested, then ATPEOM must also be TRUE. ATPNumRsp contains either the number of responses received or the sequence number of a response.

The timeout interval in seconds and the maximum number of times that a request should be made are indicated by atpTimeOut and atpRetries, respectively.

(note)

> Setting atpRetries to 255 will cause the request to be retransmitted indefinitely, until a full response is received or the call is cancelled.

ATP provides a data structure, known as a response buffer data structure (response BDS), for allocating buffer space to receive the datagram(s) of the response. A response BDS is an array of one to eight elements. Each BDS element defines the size and location of a buffer for receiving one response datagram; they're numbered Ø to 7 to correspond to the sequence numbers of the response datagrams.

ATP needs a separate buffer for each response datagram expected, since packets may not arrive in the proper sequence. It does not, however, require you to set up and use the BDS data structure to describe the response buffers; if you don't, ATP will do it for you. Two sets of calls are provided for both requests and responses; one set requires you to allocate a response BDS and the other doesn't.

Assembly-language note: The two calls that don't require you to define a BDS data structure (ATPRequest and ATPResponse) are available in Pascal only.

The number of BDS elements allocated (in other words, the maximum number of datagrams that the responder can send) is indicated in the TReq by an eight-bit bit map. The bits of this bit map are numbered Ø to 7 (the least significant bit being number Ø); each bit corresponds to the response datagram with the respective sequence number.

ATPRspBDSPtr and atpBDSSize indicate the location and number of elements in the response BDS, which has the following structure:

```
TYPE BDSElement = RECORD
                    buffSize:  INTEGER; {buffer size in bytes}
                    buffPtr:   Ptr;     {pointer to buffer}
                    dataSize:  INTEGER; {packet size}
                    userBytes: LONGINT  {user bytes}
                  END;

     BDSType = ARRAY[Ø..7] OF BDSElement; {response BDS}
     BDSPtr = ^BDSType;
```

ATPNumBufs indicates the number of elements in the response BDS that contain information. In most cases, you can allocate space for your variables of BDSType statically with a VAR declaration. However, you can allocate only the minimum space required by your ATP calls by doing the following:

```
VAR myBDSPtr: BDSPtr;

BEGIN
numOfBDS := 3; {number of elements needed}
myBDSPtr := BDSPtr(NewPtr(SIZEOF(BDSElement) * numOfBDS));
. . .
END;
```

(note)

The userBytes field of the BDSElement and the atpUserData field of the aBusRecord represent the same information in the datagram. Depending on the ATP call made, one or both of these fields will be used.

Using ATP

Before you can use ATP on a Macintosh 128K, the .ATP driver must be read from the system resource file via an ATPLoad call. The .ATP driver loads itself into the application heap and installs a task into the vertical retrace queue.

(warning)

> When another application is started up, the application heap is reinitialized; on a Macintosh 128K, this means that the ATP code is lost and must be reloaded by the next application.

When you're through using ATP on a Macintosh 128K, call ATPUnload--the system will be returned to the state it was in before the .ATP driver was opened.

On a Macintosh 512K or XL, the .ATP driver will have been loaded into the system heap either at system startup or upon execution of MPPOpen or ATPLoad. ATPUnload has no effect on a Macintosh 512K or XL.

To send a transaction request, call ATPSndRequest or ATPRequest. The .ATP driver will automatically select and open a socket through which the request datagram will be sent, and through which the response datagrams will be received. The transaction requester can't specify the number of this socket. However, the requester must specify the full network address (network number, node ID, and socket number) of the socket to which the request is to be sent. This socket is known as the responding socket, and its address must be known in advance by the requester.

(note)

> The requesting and responding sockets can't be in the same node.

At the responder's end, before a transaction request can be received, a responding socket must be opened, and the appropriate calls be made, to receive a request. To do this, the responder first makes an ATPOpenSocket call which allows the responder to specify the address (or part of it) of the requesters from whom it's willing to accept transaction requests. Then it issues an ATPGetRequest call to provide ATP with a buffer for receiving a request; when a request is received, ATPGetRequest is completed. The responder can queue up several ATPGetRequest calls, each of which will be completed as requests are received.

Upon receiving a request, the responder performs the requested operation, and then prepares the information to be returned to the requester. It then calls ATPSndRsp (or ATPResponse) to send the response. Actually, the responder can issue the ATPSndRsp call with only part (or none) of the response specified. Additional portions of the response can be sent later by calling ATPAddRsp.

The ATPSndRsp and ATPAddRsp calls provide flexibility in the design (and range of types) of transaction responders. For instance, the responder may, for some reason, be forced to send the responses out of sequence. Also, there might be memory constraints that force sending the complete transaction response in parts. Even though eight response datagrams might need to be sent, the responder might have only enough memory to build one datagram at a time. In this case, it would build the first response datagram and call ATPSndRsp to send it. It would then build the second response datagram in the same buffer and call ATPAddRsp to send it; and so on, for the third through eighth response datagrams.

A responder can close a responding socket by calling ATPCloseSocket. This call cancels all pending ATP calls for that socket, such as ATPGetRequest, ATPSndRsp, and ATPResponse.

For exactly-once transactions, the ATPSndRsp and ATPAddRsp calls don't terminate until the entire transaction has completed (that is, the responding end receives a release packet, or the release timer has expired).

To cancel a pending, asynchronous ATPSndRequest or ATPRequest call, call ATPReqCancel. To cancel a pending, asynchronous ATPSndRsp or ATPResponse call, call ATPRspCancel. Pending asynchronous ATPGetRequest calls can be cancelled only by issuing the ATPCloseSocket call, but that will cancel all outstanding calls for that socket.

(warning)

> You cannot reuse a variable of type ABusRecord passed to an ATP routine until the entire transaction has either been completed or cancelled.

ATP Routines

FUNCTION ATPLoad : OSErr; [Not in ROM]

ATPLoad first verifies that the .MPP driver is loaded and running. If it isn't, ATPLoad verifies that port B is configured for AppleTalk, and is not in use, and then loads MPP into the system heap.

ATPLoad then loads the .ATP driver, unless it's already in memory. On a Macintosh 128K, ATPLoad reads the .ATP driver from the system resource file into the application heap; on a Macintosh 512K or XL, ATP is read into the system heap.

(note)

> On a Macintosh 512K or XL, ATPLoad and MPPOpen perform essentially the same function.

Result codes		
	noErr	No error
	portInUse	Port B is already in use

	portNotCf	Port B not configured for AppleTalk

FUNCTION ATPUnload : OSErr; [Not in ROM]

ATPUnload makes the .ATP driver purgeable; the space isn't actually released by the Memory Manager until necessary.

(note)

> This call applies only to a Macintosh 128K; on a Macintosh 512K or Macintosh XL, ATPUnload has no effect.

Result codes	noErr	No error

FUNCTION ATPOpenSocket (addrRcvd: AddrBlock; VAR atpSocket: Byte) : OSErr; [Not in ROM]

ATPOpenSocket opens a socket for the purpose of receiving requests. ATPSocket contains the socket number of the socket to open; if it's Ø, a number is dynamically assigned and returned in atpSocket. AddrRcvd contains a filter of the sockets from which requests will be accepted. A Ø in the network number, node ID, or socket number field of the addrRcvd record acts as a "wild card"; for instance, a Ø in the socket number field means that requests will be accepted from all sockets in the node(s) specified by the network and node fields.

Result codes	noErr	No error
	tooManySkts	Socket table full
	noDataArea	Too many outstanding ATP calls

(note)

> If you're only going to send requests and receive responses to these requests, you don't need to open an ATP socket. When you make the ATPSndRequest or ATPRequest call, ATP automatically opens a dynamically assigned socket for that purpose.

FUNCTION ATPCloseSocket (atpSocket: Byte) : OSErr; [Not in ROM]

ATPCloseSocket closes the responding socket whose number is specified by atpSocket. It releases the data structures associated with all pending, asynchronous calls involving that socket; these pending calls are completed immediately and return the result code sktClosed.

Result codes	noErr	No error
	noDataArea	Too many outstanding ATP calls

```
FUNCTION ATPSndRequest (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
            [Not in ROM]
```

```
      ABusRecord
          <--    abOpcode            {always tATPSndRequest}
          <--    abResult            {result code}
          -->    abUserReference     {for your use}
          -->    atpAddress          {destination socket address}
          -->    atpReqCount         {request size in bytes}
          -->    atpDataPtr          {pointer to buffer}
          -->    atpRspBDSPtr        {pointer to response BDS}
          -->    atpUserData         {user bytes}
          -->    atpXO               {exactly-once flag}
          <--    atpEOM              {end-of-message flag}
          -->    atpTimeOut          {retry timeout interval in seconds}
          -->    atpRetries          {maximum number of retries}
          -->    atpNumBufs          {number of elements in response BDS}
          <--    atpNumRsp           {number of response packets actually }
                                     { received}
```

ATPSndRequest sends a request to another socket. ATPAddress is the internet address of the socket to which the request should be sent. ATPDataPtr and atpReqCount specify the location and size of a buffer that contains the request information to be sent. ATPUserData contains the user bytes for the ATP header.

ATPSndRequest requires you to allocate a response BDS. ATPRspBDSPtr is a pointer to the response BDS; atpNumBufs indicates the number of BDS elements in the BDS (this is also the maximum number of response datagrams that will be accepted). The number of response datagrams actually received is returned in atpNumRsp; if a nonzero value is returned, you can examine the response BDS to determine which packets of the transaction were actually received. If the number returned is less than requested, one of the following is true:

- Some of the packets have been lost and the retry count has been exceeded.

- ATPEOM is TRUE; this means that the response consisted of fewer packets than were expected, but that all packets sent were received (the last packet came with the atpEOM flag set).

ATPTimeOut indicates the length of time that ATPSndRequest should wait for a response before retransmitting the request. ATPRetries indicates the maximum number of retries ATPSndRequest should attempt. ATPXO should be TRUE if you want the request to be part of an exactly-once transaction.

ATPSndRequest completes when either the transaction is completed or the retry count is exceeded.

Result codes		
	noErr	No error
	reqFailed	Retry count exceeded
	tooManyReqs	Too many concurrent requests
	noDataArea	Too many outstanding ATP calls

```
FUNCTION ATPRequest (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
           [Not in ROM]
```

```
ABusRecord
     <--   abOpcode          {always tATPRequest}
     <--   abResult          {result code}
     -->   abUserReference   {for your use}
     -->   atpAddress        {destination socket address}
     -->   atpReqCount       {request size in bytes}
     -->   atpDataPtr        {pointer to buffer}
     <--   atpActCount       {number of bytes actually received}
     -->   atpUserData       {user bytes}
     -->   atpXO             {exactly-once flag}
     <--   atpEOM            {end-of-message flag}
     -->   atpTimeOut        {retry timeout interval in seconds}
     -->   atpRetries        {maximum number of retries}
     <--   atpRspUData       {user bytes received in transaction }
                             { response}
     -->   atpRspBuf         {pointer to response message buffer}
     -->   atpRspSize        {size of response message buffer}
```

ATPRequest is functionally analogous to ATPSndRequest. It sends a request to another socket, but doesn't require the caller to set up and use the BDS data structure to describe the response buffers. ATPAddress indicates the socket to which the request should be sent. ATPDataPtr and atpReqCount specify the location and size of a buffer that contains the request information to be sent. ATPUserData contains the user bytes to be sent in the request's ATP header. ATPTimeOut indicates the length of time that ATPRequest should wait for a response before retransmitting the request. ATPRetries indicates the maximum number of retries ATPRequest should attempt.

To use this call, you must have an area of contiguous buffer space that's large enough to receive all expected datagrams. The various datagrams will be assembled in this buffer and returned to you as a complete message upon completion of the transaction. The address and size of this buffer are passed in atpRspBuf and atpRspSize, respectively. Upon completion of the call, the size of the received response message is returned in atpActCount. The user bytes received in the ATP header of the first response packet are returned in atpRspUData. ATPXO should be TRUE if you want the request to be part of an exactly-once transaction.

Although you don't provide a BDS, ATPRequest in fact creates one and calls the .ATP driver (as in an ATPSndRequest call). For this reason, the abRecord fields atpRspBDSPtr and atpNumBufs are used by ATPRequest;

you should not expect these fields to remain unaltered during or after the function's execution.

For ATPRequest to receive and correctly deliver the response as a single message, the responding end must, upon receiving the request (with an ATPGetRequest call), generate the complete response as a complete message in a single buffer and then call ATPResponse.

(note)

> The responding end could also use ATPSndRsp and ATPAddRsp provided that each response packet (except the last one) contains exactly 578 ATP data bytes; the last packet in the response can contain less than 578 ATP data bytes. Also, if this method is used, only the ATP user bytes of the first response packet will be delivered to the requester; any information in the user bytes of the remaining response packets will not be delivered.

ATPRequest completes when either the transaction is completed or the retry count is exceeded.

<u>Result codes</u>	noErr	No error
	reqFailed	Retry count exceeded
	tooManyReqs	Too many concurrent requests
	sktClosed	Socket closed by a cancel call
	noDataArea	Too many outstanding ATP calls

FUNCTION ATPReqCancel (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
[Not in ROM]

Given the handle to the ABusRecord of a previously made ATPSndRequest or ATPRequest call, ATPReqCancel dequeues the ATPSndRequest or ATPRequest call, provided that the call hasn't already completed. ATPReqCancel returns noErr if the ATPSndRequest or ATPRequest call is successfully removed from the queue. If it returns cbNotFound, check the abResult field of abRecord to verify that the ATPSndRequest or ATPRequest call has been completed and determine its outcome.

<u>Result codes</u>	noErr	No error
	cbNotFound	ATP control block not found

```
FUNCTION ATPGetRequest (abRecord: ABRecHandle; async: BOOLEAN) :
          OSErr;  [Not in ROM]
```

ABusRecord

<--	abOpcode	{always tATPGetRequest}
<--	abResult	{result code}
-->	abUserReference	{for your use}
-->	atpSocket	{listening socket number}
<--	atpAddress	{source socket address}
-->	atpReqCount	{buffer size in bytes}
-->	atpDataPtr	{pointer to buffer}
<--	atpBitMap	{transaction bit map}
<--	atpTransID	{transaction ID}
<--	atpActCount	{number of bytes actually received}
<--	atpUserData	{user bytes}
<--	atpXO	{exactly-once flag}

ATPGetRequest sets up the mechanism to receive a request sent by either an ATPSndRequest or an ATPRequest call. ATPSocket contains the socket number of the socket that should listen for a request; this socket must already have been opened by calling ATPOpenSocket. The address of the socket from which the request was sent is returned in atpAddress. ATPDataPtr specifies a buffer to store the incoming request; atpReqCount indicates the size of the buffer in bytes. The number of bytes actually received in the request is returned in atpActCount. ATPUserData contains the user bytes from the ATP header. The transaction bit map is returned in atpBitMap. The transaction ID is returned in atpTransID. ATPXO will be TRUE if the request is part of an exactly-once transaction.

ATPGetRequest completes when a request is received. To cancel an asynchronous ATPGetRequest call, you must call ATPCloseSocket, but this cancels all pending calls involving that socket.

Result codes	noErr	No error
	badATPSkt	Bad responding socket
	sktClosed	Socket closed by a cancel call

```
FUNCTION ATPSndRsp (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
          [Not in ROM]
```

ABusRecord

<--	abOpcode	{always tATPSndRsp}
<--	abResult	{result code}
-->	abUserReference	{for your use}
-->	atpSocket	{responding socket number}
-->	atpAddress	{destination socket address}
-->	atpRspBDSPtr	{pointer to response BDS}
-->	atpTransID	{transaction ID}
-->	atpEOM	{end-of-message flag}
-->	atpNumBufs	{number of response packets being } { sent}
-->	atpBDSSize	{number of elements in response BDS}

ATPSndRsp sends a response to another socket. ATPSocket contains the socket number from which the response should be sent and atpAddress contains the internet address of the socket to which the response should be sent. ATPTransID must contain the transaction ID. ATPEOM is TRUE if the response BDS contains the final packet in a transaction composed of a group of packets and the number of packets in the response is less than expected. ATPRspBDSPtr points to the buffer data structure containing the responses to be sent. ATPBDSSize indicates the number of elements in the response BDS, and must be in the range 1 to 8. ATPNumBufs indicates the number of response packets being sent with this call, and must be in the range Ø to 8.

(note)

> In some situations, you may want to send only part (or possibly none) of your response message back immediately. For instance, you might be requested to send back seven disk blocks, but have only enough internal memory to store one block. In this case, set atpBDSSize to 7 (total number of response packets), atpNumBufs to Ø (number of response packets currently being sent), and call ATPSndRsp. Then as you read in one block at a time, call ATPAddRsp until all seven response datagrams have been sent.

During exactly-once transactions, ATPSndRsp won't complete until the release packet is received or the release timer expires.

Result codes		
	noErr	No error
	badATPSkt	Bad responding socket
	noRelErr	No release received
	sktClosed	Socket closed by a cancel call
	noDataArea	Too many outstanding ATP calls

FUNCTION ATPAddRsp (abRecord: ABRecHandle) : OSErr; [Not in ROM]

ABusRecord

<--	abOpcode	{always tATPAddRsp}
<--	abResult	{result code}
-->	abUserReference	{for your use}
-->	atpSocket	{responding socket number}
-->	atpAddress	{destination socket address}
-->	atpReqCount	{buffer size in bytes}
-->	atpDataPtr	{pointer to buffer}
-->	atpTransID	{transaction ID}
-->	atpUserData	{user bytes}
-->	atpEOM	{end-of-message flag}
-->	atpNumRsp	{sequence number}

ATPAddRsp sends one additional response packet to a socket that has already been sent the initial part of a response via ATPSndRsp. ATPSocket contains the socket number from which the response should be sent and atpAddress contains the internet address of the socket to which the response should be sent. ATPTransID must contain the transaction ID. ATPDataPtr and atpReqCount specify the location and size of a buffer that contains the information to send; atpNumRsp is the sequence number of the response. ATPEOM is TRUE if this response datagram is the final packet in a transaction composed of a group of packets. ATPUserData contains the user bytes to be sent in this response datagram's ATP header.

(note)

No BDS is needed with ATPAddRsp because all pertinent information is passed within the record.

Result codes		
	noErr	No error
	badATPSkt	Bad responding socket
	badBuffNum	Bad sequence number
	noSendResp	ATPAddRsp issued before ATPSndRsp
	noDataArea	Too many outstanding ATP calls

```
FUNCTION ATPResponse (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
          [Not in ROM]
```

```
      ABusRecord
         <--   abOpcode            {always tATPResponse}
         <--   abResult            {result code}
         -->   abUserReference     {for your use}
         -->   atpSocket           {responding socket number}
         -->   atpAddress          {destination socket address}
         -->   atpRspUData         {user bytes to be sent in }
                                   { transaction response}
         -->   atpRspBuf           {pointer to response message buffer}
         -->   atpRspSize          {size of response message buffer}
```

ATPResponse is functionally analogous to ATPSndRsp. It sends a response to a socket, but doesn't require the caller to provide a BDS. ATPAddress must contain the complete network address of the socket to which the response should be sent (taken from the data provided by an ATPGetRequest call). ATPSocket indicates the socket from which the response should be sent (the socket on which the corresponding ATPGetRequest was issued). ATPRspBuf points to the buffer containing the response message; the size of this buffer must be passed in atpRspSize. The four user bytes to be sent in the ATP header of the first response packet are passed in atpRspUData. The last packet of the transaction response is sent with the EOM flag set.

Although you don't provide a BDS, ATPResponse in fact creates one and calls the .ATP driver (as in an ATPSndRsp call). For this reason, the abRRecord fields atpRspBDSPtr and atpNumBufs are used by ATPResponse; you should not expect these fields to remain unaltered during or after the function's execution.

During exactly-once transactions ATPResponse won't complete until the release packet is received or the release timer expires.

(warning)

> The maximum permissible size of the response message is 4624 bytes.

Result codes		
	noErr	No error
	badATPSkt	Bad responding socket
	noRelErr	No release received
	atpLenErr	Response too big
	sktClosed	Socket closed by a cancel call
	noDataArea	Too many outstanding ATP calls

```
FUNCTION ATPRspCancel (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
          [Not in ROM]
```

Given the handle to the ABusRecord of a previously made ATPSndRsp or ATPResponse call, ATPRspCancel dequeues the ATPSndRsp or ATPResponse call, provided that the call hasn't already completed. ATPRspCancel returns noErr if the ATPSndRsp or ATPResponse call is successfully removed from the queue. If it returns cbNotFound, check the abResult field of abRecord to verify that the ATPSndRsp or ATPResponse call has been completed and determine its outcome.

Result codes		
	noErr	No error
	cbNotFound	ATP control block not found
	noDataArea	Too many outstanding ATP calls

Example

This example shows the requesting side of an ATP transaction that asks for a 512-byte disk block from the responding end. The block number of the file is a byte and is contained in myBuffer[Ø].

```
VAR myABRecord: ABRecHandle;
    myBDSPtr: BDSPtr;
    myBuffer: PACKED ARRAY[Ø..511] OF CHAR;
    errCode: INTEGER;
    async: BOOLEAN;

BEGIN
errCode := ATPLoad;
IF errCode <> noErr
  THEN
    WRITELN('Error in opening AppleTalk')
    {Maybe serial port B isn't available for use by AppleTalk}
  ELSE
    BEGIN
    {Prepare the BDS; allocate space for a one-element BDS}
    myBDSPtr := BDSPtr(NewPtr(SIZEOF(BDSElement)));
    WITH myBDSPtr^[Ø] DO
      BEGIN
      buffSize := 512;      {size of our buffer used in reception}
      buffPtr := @myBuffer;   {pointer to the buffer}
      END;
    {Prepare the ABusRecord}
    myBuffer[Ø] := CHR(1);      {requesting disk block number 1}
    myABRecord := ABRecHandle(NewHandle(atpSize));
    WITH myABRecord^^ DO
      BEGIN
      atpAddress.aNet := Ø;
      atpAddress.aNode := 3Ø;  {we probably got this from an NBP call}
      atpAddress.aSocket := 15;  {socket to send request to}
      atpReqCount := 1;   {size of request data field (disk block #)}
      atpDataPtr := @myBuffer;   {ptr to request to be sent}
      atpRspBDSPtr := @myBDSPtr;
      atpUserData := Ø;  {for your use}
      atpXO := FALSE;    {at-least-once service}
      atpTimeOut := 5;   {5-second timeout}
      atpRetries := 3;   {3 retries; request will be sent 4 times max}
      atpNumBufs := 1;   {we're only expecting 1 block to be returned}
      END;
    async := FALSE;
    {Send the request and wait for the response}
    errCode := ATPSndRequest(myABRecord,async);
    IF errCode <> noErr
      THEN
        WRITELN('An error occurred in the ATPSndRequest call')
      ELSE
        BEGIN
        {The disk block requested is now in myBuffer.  We can verify }
        { that atpNumRsp contains 1, meaning one response received.}
        ...
        END;
    END;
END.
```

Name-Binding Protocol

Data Structures

NBP calls use the following fields:

```
nbpProto:
  (nbpEntityPtr:       EntityPtr;       {pointer to entity name}
   nbpBufPtr:          Ptr;             {pointer to buffer}
   nbpBufSize:         INTEGER;         {buffer size in bytes}
   nbpDataField:       INTEGER;         {number of addresses }
                                        { or socket number}
   nbpAddress:         AddrBlock;       {socket address}
   nbpRetransmitInfo: RetransType);     {retransmission }
                                        { information}
```

When data is sent via NBP, nbpBufSize indicates the size of the data in bytes and nbpBufPtr points to a buffer containing the data. When data is received via NBP, nbpBufPtr points to a buffer in which the incoming data can be stored and nbpBufSize indicates the size of the buffer in bytes. NBPAddress is used in some calls to give the internet address of a named entity. The AddrBlock data type is described above under "Datagram Delivery Protocol".

NBPEntityPtr points to a variable of type EntityName, which has the following data structure:

```
       TYPE EntityName = RECORD
                           objStr:  Str32; {object}
                           typeStr: Str32; {type}
                           zoneStr: Str32  {zone}
                         END;

            EntityPtr = ^EntityName;

            Str32 = STRING[32];
```

NBPRetransmitInfo contains information about the number of times a packet should be transmitted and the interval between retransmissions:

```
       TYPE RetransType =
         PACKED RECORD
           retransInterval: Byte;  {retransmit interval in 8-tick units}
           retransCount:    Byte   {number of attempts}
         END;
```

Using NBP

On a Macintosh 128K, the AppleTalk Manager's NBP code is read into the application heap when any one of the NBP (Pascal) routines is called; you can call the NBPLoad function yourself if you want to load the NBP code explicitly. When you're finished with the NBP code and want to reclaim the space it occupies, call NBPUnload. On a Macintosh 512K or XL, the NBP code is read in when the .MPP driver is loaded.

When an entity wants to communicate via an AppleTalk network, it should call NBPRegister to place its name and internet address in the names table. When an entity no longer wants to communicate on the network, or is being shut down, it should call NBPRemove to remove its entry from the names table.

To determine the address of an entity you know only by name, call NBPLookup, which returns a list of all entities with the name you specify. Call NBPExtract to extract entity names from the list.

If you already know the address of an entity, and want only to confirm that it still exists, call NBPConfirm. NBPConfirm is more efficient than NBPLookup in terms of network traffic.

NBP Routines

```
FUNCTION NBPRegister (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
            [Not in ROM]
```

ABusRecord		
<--	abOpcode	{always tNBPRegister}
<--	abResult	{result code}
-->	abUserReference	{for your use}
-->	nbpEntityPtr	{pointer to entity name}
-->	nbpBufPtr	{pointer to buffer}
-->	nbpBufSize	{buffer size in bytes}
-->	nbpAddress.aSocket	{socket address}
-->	nbpRetransmitInfo	{retransmission information}

NBPRegister adds the name and address of an entity to the node's names table. NBPEntityPtr points to a variable of type EntityName containing the entity's name. If the name is already registered, NBPRegister returns the result code nbpDuplicate. NBPAddress indicates the socket for which the name should be registered. NBPBufPtr and nbpBufSize specify the location and size of a buffer for NBP to use internally. The buffer must contain at least 12 bytes plus the length of the entity name.

(warning)

> This buffer must not be altered or released until the name is removed from the names table via an NBPRemove call. If you allocate the buffer through a NewHandle call, the handle must be locked as long as the name is registered.

Result codes		
	noErr	No error
	nbpDuplicate	Duplicate name already exists

```
FUNCTION NBPLookup (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
            [Not in ROM]
```

ABusRecord

<--	abOpcode	{always tNBPLookup}
<--	abResult	{result code}
-->	abUserReference	{for your use}
-->	nbpEntityPtr	{pointer to entity name}
-->	nbpBufPtr	{pointer to buffer}
-->	nbpBufSize	{buffer size in bytes}
<->	nbpDataField	{number of addresses received}
-->	nbpRetransmitInfo	{retransmission information}

NBPLookup returns the addresses of all entities with a specified name. NBPEntityPtr points to a variable of type EntityName containing the name of the entity whose address should be returned. (Meta-characters are allowed in the entity name.) NBPBufPtr and NBPBufSize contain the location and size of an area of memory in which the entities' addresses should be returned. NBPDataField indicates the maximum number of matching names to find addresses for; the actual number of addresses found is returned in NBPDataField. NBPRetransmitInfo contains the retry interval and the retry count.

Result codes	noErr	No error
	nbpBuffOvr	Buffer overflow

```
FUNCTION NBPExtract (theBuffer: Ptr; numInBuf: INTEGER; whichOne:
            INTEGER; VAR abEntity: EntityName; VAR address: AddrBlock)
            : OSErr;  [Not in ROM]
```

NBPExtract returns one address from the list of addresses returned by NBPLookup. TheBuffer and numInBuf indicate the location and number of tuples in the buffer. WhichOne specifies which one of the tuples in the buffer should be returned in the abEntity and address parameters.

Result codes	noErr	No error
	extractErr	Can't find tuple in buffer

FUNCTION NBPConfirm (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
[Not in ROM]

ABusRecord

<--	abOpcode	{always tNBPConfirm}
<--	abResult	{result code}
-->	abUserReference	{for your use}
-->	nbpEntityPtr	{pointer to entity name}
<--	nbpDataField	{socket number}
-->	nbpAddress	{socket address}
-->	nbpRetransmitInfo	{retransmission information}

NBPConfirm confirms that an entity known by name and address still exists (is still entered in the names directory). NBPEntityPtr points to a variable of type EntityName that contains the name to confirm, and nbpAddress specifies the address to be confirmed. (No meta-characters are allowed in the entity name.) NBPRetransmitInfo contains the retry interval and the retry count. The correct socket number of the entity is returned in nbpDataField. NBPConfirm is more efficient than NBPLookup in terms of network traffic.

Result codes	noErr	No error
	nbpConfDiff	Name confirmed for different socket
	nbpNoConfirm	Name not confirmed

FUNCTION NBPRemove (abEntity: EntityPtr) : OSErr; [Not in ROM]

NBPRemove removes an entity name from the names table of the caller's node.

Result codes	noErr	No error
	nbpNotFound	Name not found

FUNCTION NBPLoad : OSErr; [Not in ROM]

On a Macintosh 128K, NBPLoad reads the AppleTalk Manager's NBP code from the system resource file into the application heap. On a Macintosh 512K or XL, NBPLoad has no effect since the NBP code should have already been loaded when the .MPP driver was opened. Normally you'll never need to call NBPLoad because the AppleTalk Manager calls it when necessary.

Result codes	noErr	No error

```
FUNCTION NBPUnload : OSErr;  [Not in ROM]
```

NBPUnload makes the NBP code purgeable; the space isn't actually released by the Memory Manager until necessary.

(note)

> This call applies only to a Macintosh 128K; on a Macintosh 512K or Macintosh XL, NBPUnload has no effect.

Result codes	noErr	No error

Example

This example of NBP registers our node as a print spooler, searches for any print spoolers registered on the network, and then extracts the information for the first one found.

```
CONST nbpNameBufSize = 33;
      mySocket = 2Ø;

VAR  myABRecord: ABRecHandle;
     myEntity: EntityName;
     entityAddr: AddrBlock;
     nbpNamePtr: Ptr;
     myBuffer: PACKED ARRAY[Ø..999] OF CHAR;
     errCode: INTEGER;
     async: BOOLEAN;

BEGIN
errCode := MPPOpen;
IF errCode <> noErr
  THEN
    WRITELN('Error in opening AppleTalk')
    {Maybe serial port B isn't available for use by AppleTalk}
  ELSE
    BEGIN
    {Call Memory Manager to allocate ABusRecord}
    myABRecord := ABRecHandle(NewHandle(nbpSize));
    {Set up our entity name to register}
    WITH myEntity DO
      BEGIN
      objStr := 'Gene Station';     {we are called 'Gene Station'  }
      typeStr := 'PrintSpooler';    { and are of type 'PrintSpooler'}
      zoneStr := '*';
      END;
    {Allocate data space for the entity name (used by NBP)}
    nbpNamePtr := NewPtr(nbpNameBufSize);
    {Set up the ABusRecord for the NBPRegister call}
    WITH myABRecord^^ DO
      BEGIN
      nbpEntityPtr := @myEntity;
      nbpBufPtr := nbpNamePtr;    {buffer used by NBP internally}
```

```
   nbpBufSize := nbpNameBufSize;
   nbpAddress.aSocket := mySocket; {socket to register us on}
   nbpRetransmitInfo.retransInterval := 8; {retransmit every 64 }
   nbpRetransmitInfo.retransCount := 3;    { ticks and try 3 times}
   END;
  async := FALSE;
  errCode := NBPRegister(myABRecord,async);
  IF errCode <> noErr
   THEN
    WRITELN('Error occurred in the NBPRegister call')
    {Maybe the name is already registered somewhere else }
    { on the network.}
   ELSE
    BEGIN
    {Now that we've registered our name, find others of }
    { type 'PrintSpooler'}
    WITH myABRecord^^ DO
     BEGIN
     nbpEntityPtr := @myEntity;
     nbpBufPtr := @myBuffer;  {buffer to place responses in}
     nbpBufSize := SIZEOF(myBuffer);
     {The field nbpDataField, before the NBPLookup call, }
     { represents an approximate number of responses. }
     { After the call, nbpDataField contains the actual }
     { number of responses received.}
     nbpDataField := 100;  {we want about 100 responses back}
     END;
    errCode := NBPLookUp(myABRecord,async);  {make sync call}
    IF errCode <> noErr
     THEN
      WRITELN('An error occurred in the NBPLookup')
      {Did the buffer overflow?}
     ELSE
      BEGIN
      {Get the first reply}
      errCode := NBPExtract(@mybuffer,myABRecord^^.nbpDataField,1,
                             myEntity,entityAddr);
      {The socket address and name of the entity are returned }
      { here.  If we want all of them, we'll have to loop }
      { for each one in the buffer.}
      IF errCode <> noErr
       THEN
        WRITELN('Error in NBPExtract');
        {Maybe the one we wanted wasn't in the buffer}
      END;
    END;
  END;
END.
```

Miscellaneous Routines

FUNCTION GetNodeAddress (VAR myNode,myNet: INTEGER) : OSErr; [Not in ROM]

GetNodeAddress returns the current node ID and network number of the caller. If the .MPP driver isn't installed, it returns noMPPErr. If myNet contains Ø, this means that a bridge hasn't yet been found.

Result codes	noErr	No error
	noMPPErr	MPP driver not installed

FUNCTION IsMPPOpen : BOOLEAN; [Not in ROM]

IsMPPOpen returns TRUE if the .MPP driver is loaded and running.

FUNCTION IsATPOpen : BOOLEAN; [Not in ROM]

IsATPOpen returns TRUE if the .ATP driver is loaded and running.

CALLING THE APPLETALK MANAGER FROM ASSEMBLY LANGUAGE

This section discusses how to use the AppleTalk Manager from assembly language. Equivalent Pascal information is given in the preceding section.

All routines make Device Manager Control calls; the description for each routine includes a list of the fields needed. Some of these fields are part of the parameter block described in the Device Manager manual; additional fields are provided for the AppleTalk Manager.

The number next to each field name indicates the byte offset of the field from the start of the parameter block pointed to by AØ. An arrow next to each parameter name indicates whether it's an input, output, or input/output parameter:

Arrow	Meaning
-->	Parameter must be passed to the routine
<--	Parameter will be returned by the routine
<->	Parameter must be passed to and will be returned by the routine

All Device Manager Control calls return a result code of type OSErr in the ioResult field. Each routine description lists all of the applicable result codes generated by the AppleTalk Manager, along with a short description of what the result code means. Lengthier explanations of all the result codes can be found in the summary at the end of this manual. Result codes from other parts of the Operating System may also be returned. (See the Operating System Utilities manual for a list of all result codes.)

Opening AppleTalk

The .MPP driver is opened at system startup. Two tests to determine whether serial port B is configured for AppleTalk and is not being used. If either of these tests fail, (indicating that port B isn't available), the Device Manager Open call will fail. It's the application's responsibility to test the availability of port B before opening AppleTalk. Assembly-language programmers can use the Pascal calls MPPOpen and ATPLoad to open the .MPP and .ATP drivers.

A byte in parameter RAM is used for configuring the serial ports; it's copied into the global variable SPConfig. The low-order four bits of this variable contain the current configuration of port B. The following use types are provided as global constants for testing or setting the configuration of port B:

```
      useFree     .EQU    Ø    ;unconfigured
      useATalk    .EQU    1    ;configured for AppleTalk
      useAsync    .EQU    2    ;configured for the Serial Driver
```

The application shouldn't attempt to open AppleTalk unless SPConfig is equal to either useFree or useATalk.

A second test involves the global variable PortBUse; the low-order four bits of this variable are used to monitor the current use of port B. If PortBUse is negative, the program is free to open AppleTalk. If PortBUse is positive, the program should test to see if port B is already being used by AppleTalk; if it is, the low-order four bits of PortBUse will be equal to the use type useATalk.

The .MPP driver sets this byte to the correct value (useATalk) when it's opened and resets it to $FF when it's closed. ATP uses bit 4 from the driver-specific bits to indicate whether it's currently opened:

```
      atpLoadedBit   .EQU    4    ;set if ATP is opened
```

Example

The following code illustrates the use of the SPConfig and PortBUse variables.

```
          MOVE      #-<ATPUnitNum+1>,ATPRefNum(AØ)  ;save known ATP
                                            ; refnum in case ATP not opened
OpenAbus  SUB       #ioQElSize,SP           ;allocate queue element
          MOVE.L    SP,AØ                   ;AØ -> queue element
          CLR.B     ioPermssn(AØ)           ;make sure permission's clear
          MOVE.B   PortBUse,D1          ;is port B in use?
          BPL.S    @1Ø                  ;if so, make sure by AppleTalk
          MOVEQ    #portNotCf,DØ        ;assume port not configured for
                                        ; AppleTalk
          MOVE.B   SPConfig,D1          ;get configuration data
          AND.B    #$ØF,D1              ;mask it to low 4 bits
          SUBQ.B   #useATalk,D1         ;unconfigured or configured for
                                        ; AppleTalk
          BGT.S    @3Ø                  ;if not, return error
          LEA      MPPName,A1           ;A1 = address of driver name
          MOVE.L   A1,ioFileName(AØ)    ;set in queue element
          _Open                         ;open MPP
          BNE.S    @3Ø                  ;return error, if it can't load it
          BRA.S    @2Ø                  ;otherwise, go check ATP
@1Ø       MOVEQ    #portInUse,DØ        ;assume port in use error
          AND.B    #$ØF,D1              ;clear all but use bits
          SUBQ.B   #useATalk,D1         ;is AppleTalk using it?
          BNE.S    @3Ø                  ;if not, then error
@2Ø       MOVEQ    #Ø,DØ                ;assume no error
          BTST     #atpLoadedBit,PortBUse  ;ATP already open?
          BNE.S    @3Ø                  ;just return if so
          LEA      ATPName,A1           ;A1 = address of driver name
          MOVE.L   A1,ioFileName(AØ)    ;set in queue element
          _Open                         ;open ATP
@3Ø       ADD      #ioQElSize,SP        ;deallocate queue element
          RTS                           ;and return
MPPName   .BYTE    4                    ;length of .MPP driver name
          .ASCII   '.MPP'               ;name of .MPP driver
ATPName   .BYTE    4                    ;length of .ATP driver name
          .ASCII   '.ATP'               ;name of .ATP driver
```

AppleTalk Link Access Protocol

Data Structures

An ALAP frame is composed of a 3-byte header, up to 6ØØ bytes of data, and a 2-byte frame check sequence (Figure 5). You can use the following global constants to access the contents of an ALAP header:

```
lapDstAdr   .EQU   Ø   ;destination node ID
lapSrcAdr   .EQU   1   ;source node ID
lapType     .EQU   2   ;LAP protocol type
lapHdSz     .EQU   3   ;ALAP header size
```

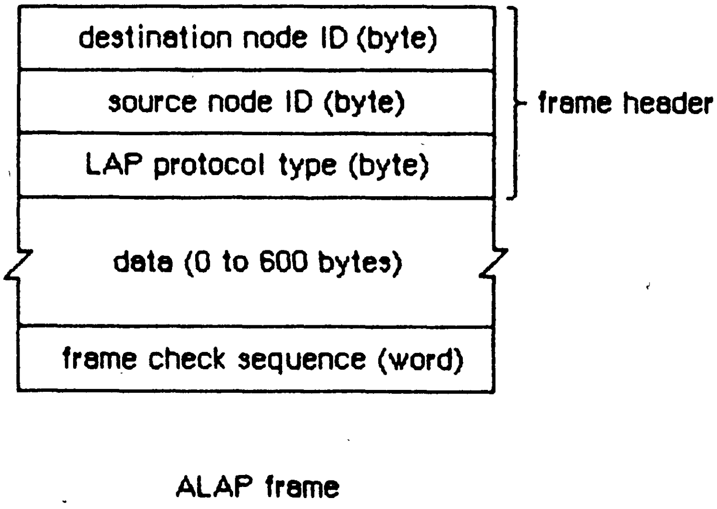


Figure 5. ALAP Frame

Two of the protocol handlers in every node are used by DDP. These protocol handlers service frames with LAP protocol types equal to the following global constants:

```
shortDDP    .EQU   1    ;short DDP header
longDDP     .EQU   2    ;long DDP header
```

When you call ALAP to send a frame, you pass it information about the frame in a write data structure, which has the format shown in Figure 6.

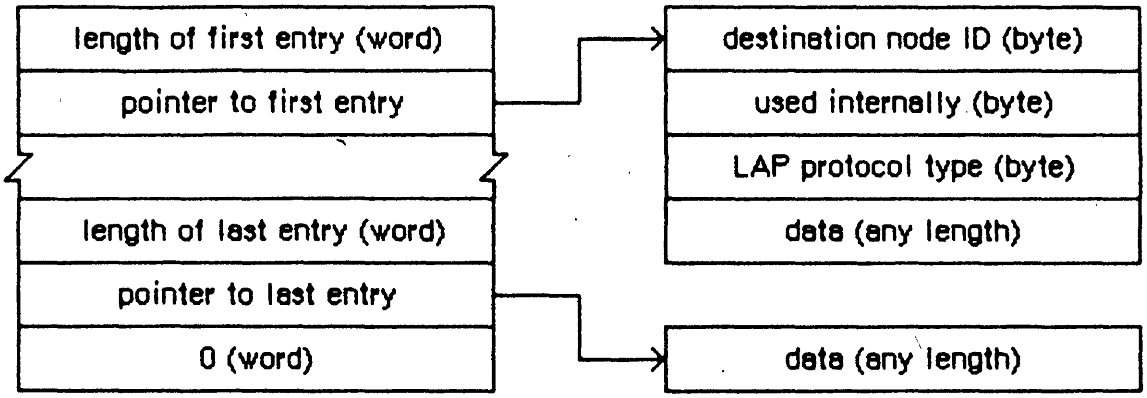


Figure 6. Write Data Structure for ALAP

If you specify a destination node ID of 255, the frame will be broadcast to all nodes. The byte that's "used internally" is used by the AppleTalk Manager to store the address of the node sending the frame.

Using ALAP

Most programs will never need to call ALAP, because higher-level protocols will automatically call ALAP as necessary. If you do want to send a frame directly via ALAP, call the WriteLAP function. There's no ReadLAP function in assembly language; if you want to read ALAP frames, you must call AttachPH to add your protocol handler to the node's protocol handler table. The ALAP module will examine every incoming frame and call your protocol handler for each frame received with the correct LAP protocol. When your program no longer wants to receive frames with a particular LAP protocol type value, it can call DetachPH to remove the corresponding protocol handler from the protocol handler table.

See the "Protocol Handlers and Socket Listeners" section for information on how to write a protocol handler.

ALAP Routines

WriteLAP function

Parameter block

-->	26	csCode	word	;always writeLAP
-->	3Ø	wdsPointer	pointer	;write data structure

WriteLAP sends a frame to another node. The frame data and destination of the frame are described by a write data structure pointed to by wdsPointer. The first two data bytes of an ALAP frame sent to another computer using the AppleTalk Manager must indicate the length of the frame in bytes.

Result codes		
	noErr	No error
	excessCollsns	No CTS received after 32 RTS's
	ddpLengthErr	Packet length exceeds maximum

AttachPH function

```
Parameter block
    -->  26   csCode       word        ;always attachPH
    -->  28   protType     byte        ;LAP protocol type
    -->  3Ø   handler      pointer     ;protocol handler
```

AttachPH adds the protocol handler pointed to by handler to the node's protocol table. ProtType specifies what kind of frame the protocol handler can service. After AttachPH is called, the protocol handler is called for each incoming frame whose LAP protocol type equals protType.

Result codes	noErr	No error
	lapProtErr	Error attaching protocol type

DetachPH function

```
Parameter block
    -->  26   csCode       word        ;always detachPH
    -->  28   protType     byte        ;LAP protocol type
```

DetachPH removes from the node's protocol table the specified LAP protocol type and corresponding protocol handler.

Result codes	noErr	No error
	lapProtErr	Error detaching protocol type

Datagram Delivery Protocol

Data Structures

A DDP datagram consists of a header followed by up to 586 bytes of actual data (Figure 5). The headers can be of two different lengths; they're identified by the following LAP protocol types:

```
shortDDP   .EQU  1   ;short DDP header
longDDP    .EQU  2   ;long DDP header
```

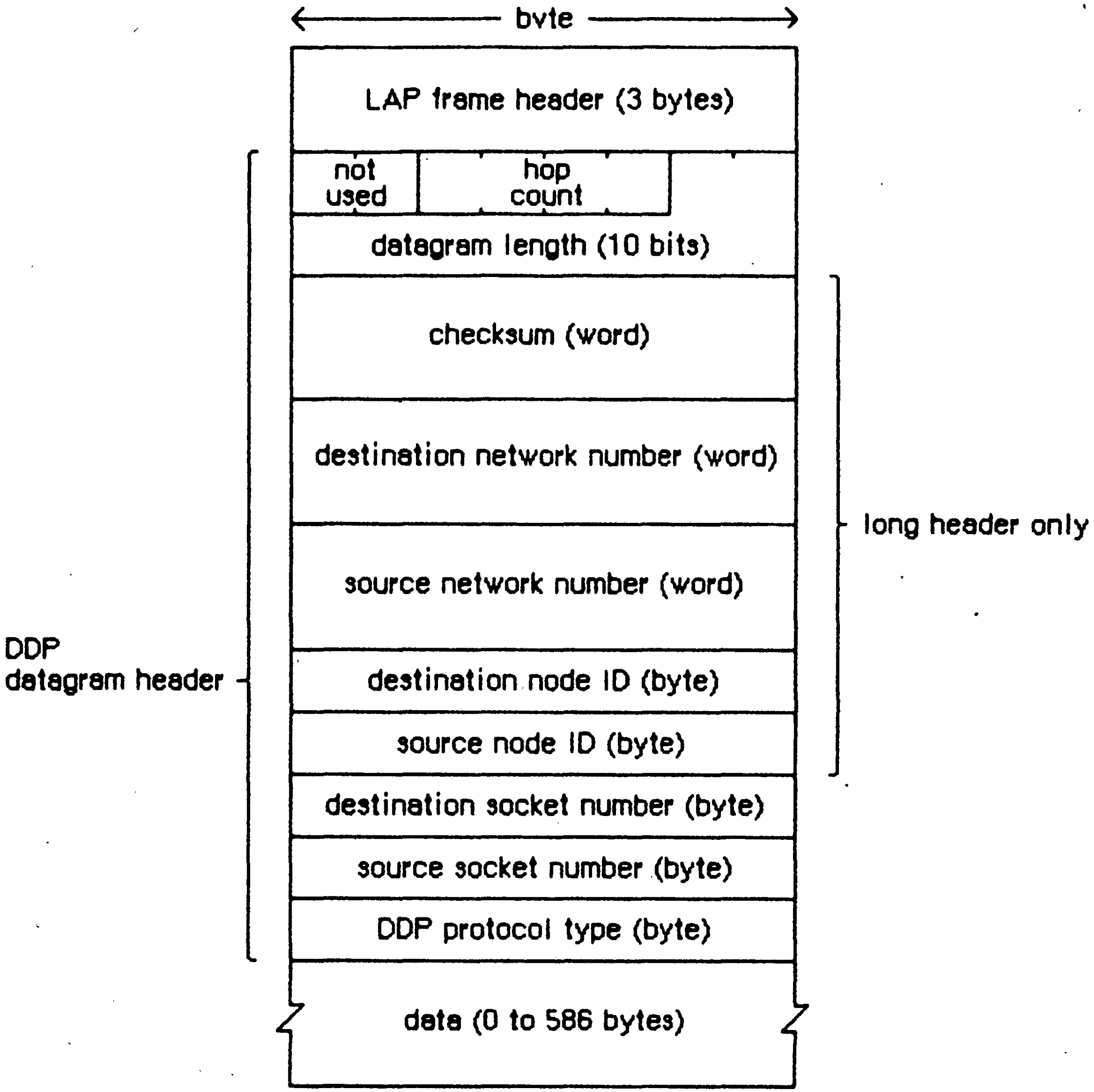

byte
LAP frame header (3 bytes)
not used
hop count
datagram length (10 bits)
checksum (word)
destination network number (word)
source network number (word)
destination node ID (byte)
source node ID (byte)
destination socket number (byte)
source socket number (byte)
DDP protocol type (byte)
data (0 to 586 bytes)
DDP datagram header
long header only

Figure 7. DDP Datagram

Long DDP headers (13 bytes) are used for sending datagrams between two or more different AppleTalk networks. You can use the following global constants to access the contents of a long DDP header:

```
ddpHopCnt      .EQU   Ø          ;hop count (4 bits)
ddpLength      .EQU   Ø          ;datagram length (1Ø bits)
ddpChecksum    .EQU   2          ;checksum
ddpDstNet      .EQU   4          ;destination network number
ddpSrcNet      .EQU   6          ;source network number
ddpDstNode     .EQU   8          ;destination node ID
ddpSrcNode     .EQU   9          ;source node ID
ddpDstSkt      .EQU   1Ø         ;destination socket number
ddpSrcSkt      .EQU   11         ;source socket number
ddpType        .EQU   12         ;DDP protocol type
```

The length of a DDP long header is given by the following constant:

```
ddpHSzLong     .EQU   ddpType+1   ;length of DDP long header
```

The short headers (five bytes) are used for datagrams sent to sockets within the same network as the source socket. You can use the following global constants to access the contents of a short DDP header:

```
ddpLength      .EQU   Ø              ;datagram length
sDDPDstSkt     .EQU   ddpChecksum    ;destination socket number
sDDPSrcSkt     .EQU   sDDPDstSkt+1   ;source socket number
sDDPType       .EQU   sDDPSrcSkt+1   ;DDP protocol type
```

The length of a DDP short header is given by the following constant:

```
ddpHSzShort    .EQU   sDDPType+1     ;length of DDP short header
```

The datagram length is a 1Ø-bit field. You can use the following global constant as a mask for these bits:

```
lengthMask   .EQU   $Ø3FF   ;mask for datagram length
```

The following constant indicates the maximum length of a DDP datagram:

```
ddpMaxData   .EQU     586   ;maximum length of DDP data
```

When you call DDP to send a datagram, you pass it information about the datagram in a write data structure with the format shown in Figure 8.

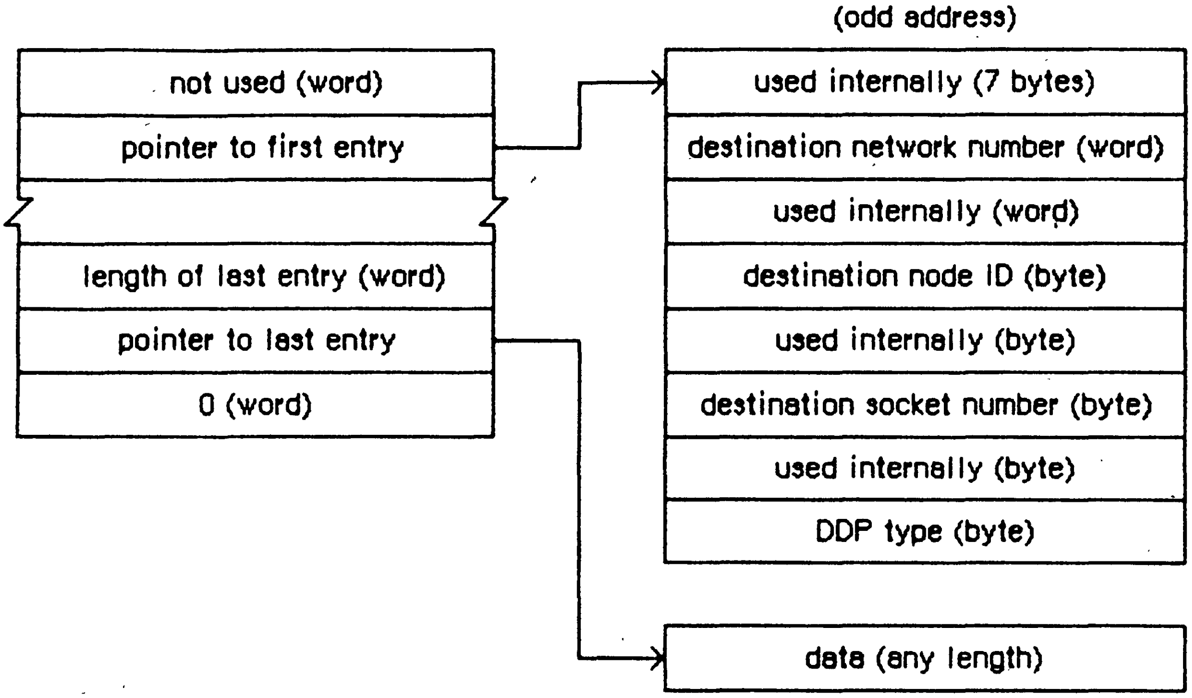


Figure 8. Write Data Structure for DDP

The first seven bytes are used internally for the ALAP header and the DDP datagram length and checksum. The other bytes used internally store the network number, node ID, and socket number of the socket client sending the datagram.

(warning)
The first entry in a DDP write data structure must begin at an odd address.

If you specify a node number of 255, the datagram will be broadcast to all nodes within the destination network. A network number of Ø means the local network to which the node is connected.

(warning)
DDP always destroys the high-order byte of the destination network number when it sends a datagram with a short header. Therefore, if you want to reuse the first entry of a DDP write data structure entry, you must restore the destination network number.

Using DDP

Before it can use a socket, the program must call OpenSkt, which adds a socket and its socket listener to the socket table. When a client is finished using a socket, call CloseSkt, which removes the socket's entry from the socket table. To send a datagram via DDP, call WriteDDP. If you want to read DDP datagrams, you must write your own socket listener. DDP will send every incoming datagram for that socket to your socket listener.

See the "Protocol Handlers and Socket Listeners" section for information on how to write a socket listener.

DDP Routines

OpenSkt function

Parameter block

-->	26	csCode	word	;always openSkt
<->	28	socket	byte	;socket number
-->	3Ø	listener	pointer	;socket listener

OpenSkt adds a socket and its socket listener to the socket table. If the socket parameter is nonzero, it specifies the socket's number (in the range 1 to 127); if socket is Ø, OpenSkt opens a socket with a socket number in the range 128 to 254, and returns it in the socket parameter. Listener contains a pointer to the socket listener.

OpenSkt will return ddpSktErr if you pass the number of an already opened socket, if you pass a socket number greater than 127, or if the socket table is full (the socket table can hold a maximum of 12 sockets).

Result codes	noErr	No error
	ddpSktErr	Socket error

CloseSkt function

Parameter block

-->	26	csCode	word	;always closeSkt
-->	28	socket	byte	;socket number

CloseSkt removes the entry of the specified socket from the socket table. If you pass a socket number of Ø, or if you attempt to close a socket that isn't open, CloseSkt will return ddpSktErr.

Result codes	noErr	No error
	ddpSktErr	Socket error

WriteDDP function

Parameter block				
-->	26	csCode	word	;always writeDDP
-->	28	socket	byte	;socket number
-->	29	checksumFlag	byte	;checksum flag
-->	3Ø	wdsPointer	pointer	;write data structure

WriteDDP sends a datagram to another socket. WDSPointer contains a pointer to a write data structure containing the datagram and the address of the destination socket. If checksumFlag is TRUE, WriteDDP will compute the checksum for all datagrams requiring long headers.

Result codes		
	noErr	No error
	ddpLenErr	Datagram length too big
	ddpSktErr	Socket error
	noBridgeErr	No bridge found

AppleTalk Transaction Protocol

Data Structures

An ATP packet consists of an ALAP header, DDP header, and ATP header, followed by actual data (Figure 9). You can use the following global constants to access the contents of an ATP header:

```
atpControl     .EQU   Ø    ;control information
atpBitMap      .EQU   1    ;bit map
atpRespNo      .EQU   1    ;sequence number
atpTransID     .EQU   2    ;transaction ID
atpUserData    .EQU   4    ;user bytes
```

The length of an ATP header is given by the following constant:

```
atpHdSz        .EQU   8    ;ATP header size
```

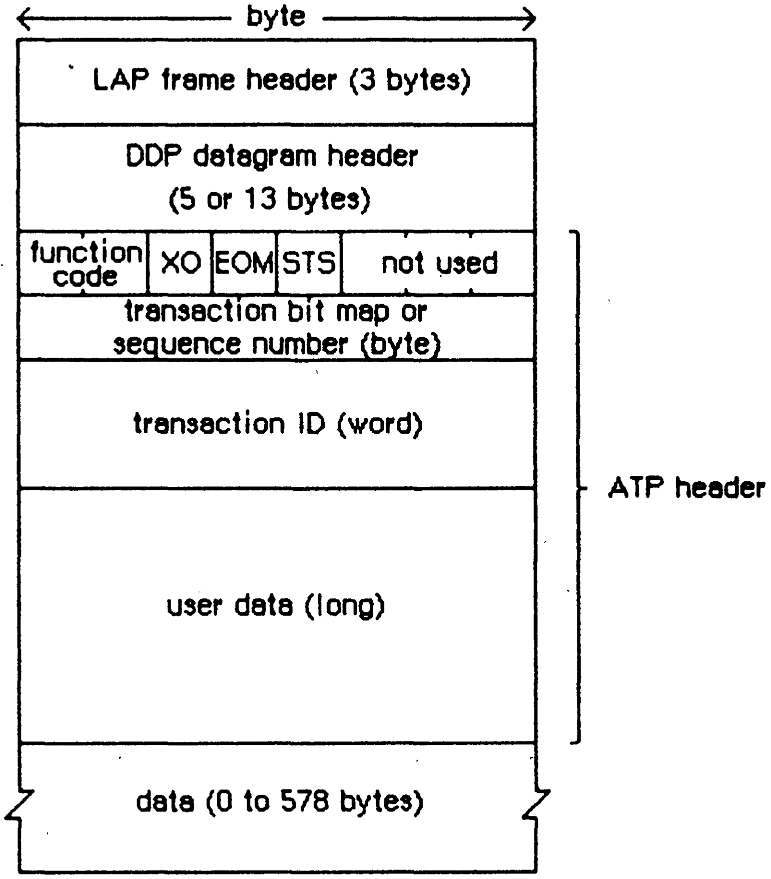


Figure 9. ATP Packet

ATP packets are identified by the following DDP protocol type:

```
       atp          .EQU   3      ;DDP protocol type for ATP packets
```

The control information contains a function code and various control bits. The function code identifies either a TReq, TResp, or TRel packet with one of the following global constants:

```
       atpReqCode    .EQU    $4Ø    ;TReq packet
       atpRspCode    .EQU    $8Ø    ;TResp packet
       atpRelCode    .EQU    $CØ    ;TRel packet
```

The send-transmission-status, end-of-message, and exactly-once bits in the control information are accessed via the following global constants:

```
       atpSTSBit    .EQU   3      ;send-transmission-status bit
       atpEOMBit    .EQU   4      ;end-of-message bit
       atpXOBit     .EQU   5      ;exactly-once bit
```

Many ATP calls require a field called atpFlags (Figure 1Ø), which contains the above three bits plus the following two bits:

```
sendChk     .EQU   Ø      ;checksum bit
tidValid    .EQU   1      ;transaction ID validity
```

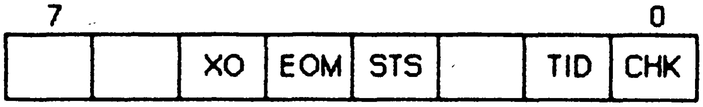


Figure 1Ø. ATPFlags Field

The maximum number of response packets in an ATP transaction is given by the following global constant:

```
atpMaxNum   .EQU   8   ;maximum number of response packets
```

When you call ATP to send responses, you pass the responses in a response BDS, which is a list of up to eight elements, each of which contains the following:

```
bdsBuffSz      .EQU   Ø    ;length of data to send
bdsBuffAddr    .EQU   2    ;pointer to data
bdsUserData    .EQU   8    ;user bytes
```

When you call ATP to receive responses, you pass it a response BDS with up to eight elements, each in the following format:

```
bdsBuffSz      .EQU   Ø    ;buffer size in bytes
bdsBuffAddr    .EQU   2    ;pointer to buffer
bdsDataSz      .EQU   6    ;number of bytes actually received
bdsUserData    .EQU   8    ;user bytes
```

The length of BDS element is given by the following constant:

```
bdsEntrySz     .EQU   12   ;response BDS element size
```

ATP clients are identified by internet addresses in the form shown in Figure 11.

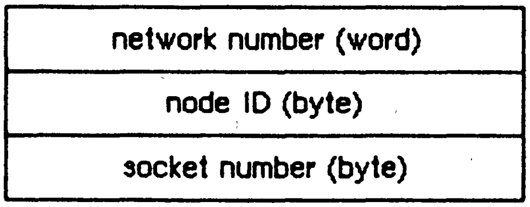


Figure 11. Internet Address

Using ATP

Before you can use ATP on a Macintosh 128K, the .ATP driver must be read from the system resource file via a Device Manager Open call. The name of the .ATP driver is '.ATP' and its reference number is -11. When the .ATP driver is opened, it reads its ATP code into the application heap and installs a task into the vertical retrace queue.

(warning)

> When another application is started up, the application heap is reinitialized; on a Macintosh 128K, this means that the ATP code is lost.

When you're through using ATP on a Macintosh 128K, call the Device Manager Close routine--the system will be returned to the state it was in before the .ATP driver was opened.

On a Macintosh 512K or XL, the .ATP driver will have been loaded into the system heap either at system startup or upon execution of a Device Manager Open call loading MPP. You shouldn't close the .ATP driver on a Macintosh 512K or XL; AppleTalk expects it to remain on these systems.

To send a request to another socket and get a response, call SendRequest. The call terminates when either an entire response is received or a specified retry timeout interval elapses. To open a socket for the purpose of responding to requests, call OpenATPSkt. Then call GetRequest to receive a request; when a request is received, the call is completed. After receiving and servicing a request, call SendResponse to return response information. If you cannot or do not want to send the entire response all at once, make a SendResponse call to send some of the response, and then call AddResponse later to send the remainder of the response. To close a socket opened for the purpose of sending responses, call CloseATPSkt.

During exactly-once transactions, SendResponse doesn't terminate until the transaction is completed via a TRel packet, or the retry count is exceeded.

(warning)

> Don't modify the parameter block passed to an ATP call until the call is completed.

ATP Routines

OpenATPSkt function

```
        Parameter block
            -->   26   csCode       word          ;always openATPSkt
            <->   28   atpSocket    byte          ;socket number
            -->   3Ø   addrBlock    long word     ;socket specification
                                                  ; for requests
```

OpenATPSkt opens a socket for the purpose of receiving requests. ATPSocket contains the socket number of the socket to open. If it's Ø, a number is dynamically assigned and returned in atpSocket. AddrBlock contains a specification of the socket addresses from which requests will be accepted. A Ø in the network number, node ID, or socket number field of addrBlock means that requests will be accepted from every network, node, or socket, respectively.

Result codes		
	noErr	No error
	tooManySkts	Too many responding sockets
	noDataArea	Too many outstanding ATP calls

CloseATPSkt function

```
        Parameter block
            -->   26   csCode       word      ;always closeATPSkt
            -->   28   atpSocket    byte      ;socket number
```

CloseATPSkt closes the socket whose number is specified by atpSocket, for the purpose of receiving requests.

Result codes		
	noErr	No error
	noDataArea	Too many outstanding ATP calls

SendRequest function

Parameter block

<--	16	reqTID	word	;transaction ID used ;in request
-->	18	userData	long word	;user bytes
-->	26	csCode	word	;always sendRequest
<--	28	currBitMap	byte	;transaction bit map ;of responses ;received so far
<->	29	atpFlags	byte	;control information
-->	30	addrBlock	long word	;destination socket ; address
-->	34	reqLength	word	;request size in bytes
-->	36	reqPointer	pointer	;pointer to request ; data
-->	40	bdsPointer	pointer	;pointer to response ; BDS
-->	44	numOfBuffs	byte	;number of responses ; expected
-->	45	timeOutVal	byte	;timeout interval
<--	46	numOfResps	byte	;number of responses ; received
<->	47	retryCount	byte	;number of retries

SendRequest sends a request to another socket and waits for a response. UserData contains the four user bytes. AddrBlock indicates the socket to which the request should be sent. ReqLength and reqPointer contain the length and location of the request to send. BDSPointer points to a response BDS where the responses are to be returned; numOfBuffs indicates the number of responses requested. The number of responses received is returned in numOfResps. If a nonzero value is returned in numOfResps, you can examine currBitMap to determine which packets of the transaction were actually received and to detect pieces for higher-level recovery, if desired.

TimeOutVal indicates the number of seconds that SendRequest should wait for a response before resending the request. RetryCount indicates the maximum number of retries SendRequest should attempt. The end-of-message flag of atpFlags will be set if the EOM bit is set in the last packet received in a valid response sequence. The exactly-once flag should be set if you want the request to be part of an exactly-once transaction.

To cancel a SendRequest call, you need the transaction ID; it's returned in reqTID. You can examine reqTID prior to the completion of the call, but its contents are valid only after the tidValid bit of ATPFlags has been set.

SendRequest completes when either an entire response is received or the retry count is exceeded.

(note)

The value provided in retryCount will be modified during SendRequest if any retries are made. This field is used to monitor the number of retries; for each retry, it's decremented by 1.

<u>Result codes</u>		
	noErr	No error
	reqFailed	Retry count exceeded
	tooManyReqs	Too many concurrent requests
	noDataArea	Too many outstanding ATP calls
	reqAborted	Request canceled by user

GetRequest function

<u>Parameter block</u>

<--	18	userData	long word	;user bytes
-->	26	csCode	word	;always getRequest
-->	28	atpSocket	byte	;socket number
<--	29	atpFlags	byte	;control information
<--	3Ø	addrBlock	long word	;source of request
<->	34	reqLength	word	;request buffer size in ; bytes
-->	36	reqPointer	pointer	;pointer to request ; buffer
<--	44	bitMap	byte	;bit map
<--	46	transID	word	;transaction ID

GetRequest sets up the mechanism to receive a request sent by a SendRequest call. UserData returns the four user bytes from the request. ATPSocket contains the socket number of the socket that should listen for a request. The internet address of the socket from which the request was sent is returned in addrBlock. ReqLength and reqPointer indicate the size and location of a buffer to store the incoming request. The actual size of the request is returned in reqLength. The transaction bit map and transaction ID will be returned in bitMap and transID. The exactly-once flag in atpFlags will be set if the request is part of an exactly-once transaction.

GetRequest completes when a request is received.

<u>Result codes</u>		
	noErr	No error
	badATPSkt	Bad responding socket

SendResponse function

Parameter	block			
<--	18	userData	long word	;user bytes from TRel
-->	26	csCode	word	;always sendResponse
-->	28	atpSocket	byte	;socket number
-->	29	atpFlags	byte	;control information
-->	30	addrBlock	long word	;response destination
-->	40	bdsPointer	pointer	;pointer to BDS
-->	44	numOfBuffs	byte	;number of packets ; being sent
-->	45	bdsSize	byte	;BDS size in elements
-->	46	transID	word	;transaction ID

SendResponse sends a response to a socket. If the response was part of an exactly-once transaction, userData will contain the user bytes from the TRel packet. ATPSocket contains the socket number from which the response should be sent. The end-of-message flag in atpFlags should be set if the response contains the final packet in a transaction composed of a group of packets and the number of responses is less than requested. AddrBlock indicates the address of the socket to which the response should be sent. BDSPointer points to a response BDS containing room for the maximum number of responses to be sent; bdsSize contains this maximum number. NumOfBuffs contains the number of response packets to be sent in this call; you may wish to make AddResponse calls to complete the response. TransID indicates the transaction ID of the associated request.

To cancel a SendResponse call, you need the transaction ID; it's returned in reqTID. You can examine reqTID prior to the completion of the call, but its contents are valid only after the tidValid bit of ATPFlags has been set.

During exactly-once transactions, SendResponse doesn't complete until either a TRel packet is received from the socket that made the request, or the retry count is exceeded.

Result codes	noErr	No error
	badATPSkt	Bad responding socket
	noRelErr	No release received
	noDataArea	Too many outstanding ATP calls
	badBuffNum	Sequence number out of range

AddResponse function

Parameter block

-->	18	userData	long word	;user bytes
-->	26	csCode	word	;always addResponse
-->	28	atpSocket	byte	;socket number
-->	29	atpFlags	byte	;control information
-->	3Ø	addrBlock	long word	;response destination
-->	34	reqLength	word	;response size in ; bytes
-->	36	reqPointer	pointer	;pointer to response
-->	44	rspNum	byte	;sequence number
-->	46	transID	word	;transaction ID

AddResponse sends an additional response packet to a socket that has already been sent the initial part of a response via SendResponse. UserData contains the four user bytes. ATPSocket contains the socket number from which the response should be sent. The end-of-message flag in atpFlags should be set if this response packet is the final packet in a transaction composed of a group of packets and the number of responses is less than requested. AddrBlock indicates the socket to which the response should be sent. ReqLength and reqPointer contain the length and location of the response to send; rspNum indicates the sequence number of the response (in the range Ø through 7). TransID must contain the transaction ID.

(warning)

> If the transaction is part of an exactly-once transaction, the buffer used in the AddResponse call must not be altered or released until the corresponding SendResponse call has completed.

Result codes	noErr	No error
	badATPSkt	Bad responding socket
	noSendResp	AddResponse issued before SendResponse
	badBuffNum	Sequence number out of range
	noDataArea	Too many outstanding ATP calls

RelTCB function

Parameter block

-->	26	csCode	word	;always relTCB
-->	3Ø	addrBlock	long word	;destination of request
<--	46	transID	word	;transaction ID of request

RelTCB dequeues the specified SendRequest call and returns the result code reqAborted for the aborted call. The transaction ID can be obtained from the reqTID field of the SendRequest queue element; see the description of SendRequest for details.

Result codes	noErr	No error
	cbNotFound	ATP control block not found

RelRspCB function

```
Parameter block
    -->   26   csCode      word          ;always relRspCB
    -->   28   atpSocket   byte          ;socket number that request
                                         ;was received on
    -->   3Ø   addrBlock   long word     ;source of request
    <--   46   transID     word          ;transaction ID of request
```

In an exactly-once transaction, RelRspCB cancels the specified SendResponse, without waiting for the release timer to expire or a TRel packet to be received. No error is returned for the SendResponse call; the call is said to have completed successfully. When called to cancel a transaction that isn't using exactly-once service, RelRspCB returns cbNotFound. The transaction ID can be obtained from the reqTID field of the SendResponse queue element; see the description of SendResponse for details.

Result codes	noErr	No error
	cbNotFound	ATP control block not found

Name-Binding Protocol

Data Structures

The first two bytes in the NBP header (Figure 12) indicate the type of the packet, the number of tuples in the packet, and an NBP packet identifier. You can use the following global constants to access these bytes:

```
nbpControl   .EQU   Ø   ;packet type
nbpTCount    .EQU   Ø   ;tuple count
nbpID        .EQU   1   ;NBP packet identifier
nbpTuple     .EQU   2   ;start of first tuple
```

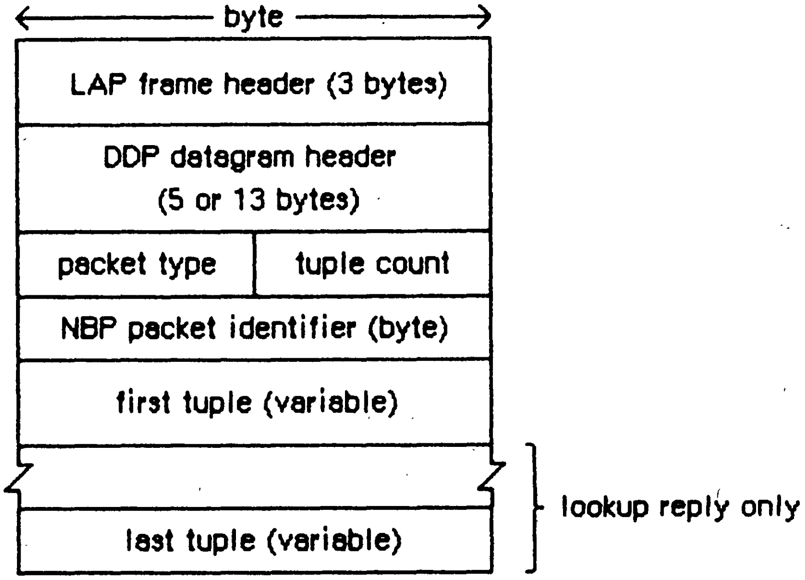

byte
LAP frame header (3 bytes)
DDP datagram header (5 or 13 bytes)
packet type
tuple count
NBP packet identifier (byte)
first tuple (variable)
last tuple (variable)
lookup reply only

Figure 12. NBP Packet

NBP packets are identified by the following DDP protocol type:

```
nbp    .EQU    2    ;DDP protocol type for NBP packets
```

NBP uses the following global constants in the nbpControl field to identify NBP packets:

```
brRq         .EQU   1    ;broadcast request
lkUp         .EQU   2    ;lookup request
lkUpReply    .EQU   3    ;lookup reply
```

NBP entities are identified by internet address in the form shown in Figure 13. Entities are also identified by tuples, which include both an internet address and an entity name. You can use the following global constants to access information in tuples:

```
tupleNet     .EQU   Ø    ;network number
tupleNode    .EQU   2    ;node ID
tupleSkt     .EQU   3    ;socket number
tupleEnum    .EQU   4    ;used internally
tupleName    .EQU   5    ;entity name
```

The meta-characters in an entity name can be identified with the following global constants:

```
equals      .EQU   '='   ;"wild-card" meta-character
star        .EQU   '*'   ;"this zone" meta-character
```

The maximum number of tuples in an NBP packet is given by the following global constant:

```
tupleMax    .EQU   15    ;number of tuples in a lookup reply
```

Entity names are mapped to sockets via the names table. Each entry in the names table has the structure shown in Figure 13.

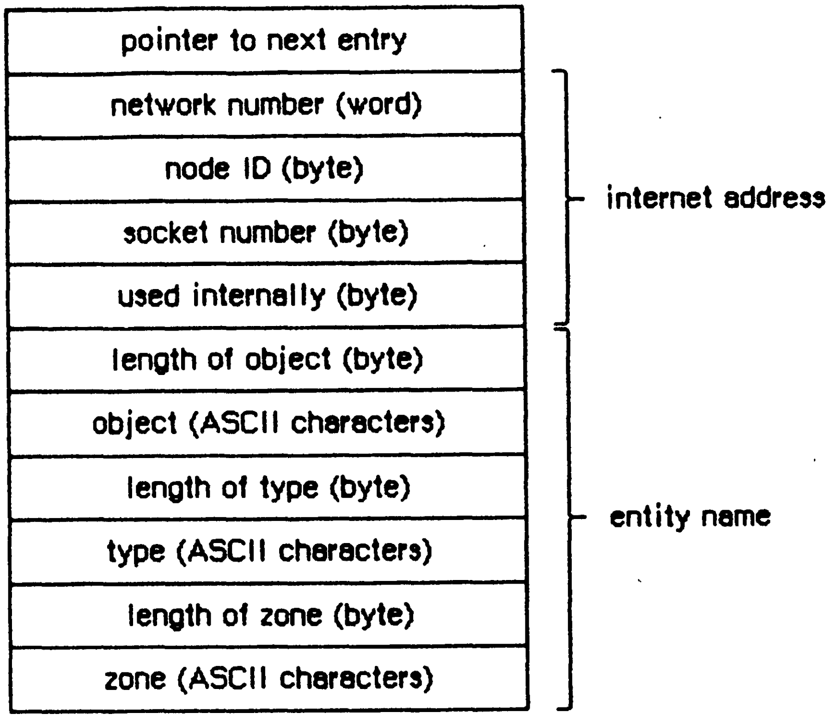


Figure 13. Names Table Entry

You can use the following global constants to access some of the elements of a names table entry:

```
ntLink      .EQU   Ø                    ;pointer to next entry
ntTuple     .EQU   ntLink+4             ;tuple
ntSocket    .EQU   ntTuple+tupleSkt     ;socket number
ntEntity    .EQU   ntTuple+tupleName    ;entity name
```

The socket number of the names information socket is given by the following global constant:

```
nis     .EQU   2      ;names information socket number
```

Using NBP

On a Macintosh 128K, before calling any other NBP routines, call the LoadNBP function, which reads the NBP code from the system resource file into the application heap. (The NBP code is part of the .MPP driver, which has a driver reference number of -1Ø.) When you're finished with NBP and want to reclaim the space its code occupies, call UnloadNBP. On a Macintosh 512K or XL, there shouldn't be any need to load or unload the NBP code.

(warning)

> When an application is started up, the application heap is reinitialized; on a Macintosh 128K the NBP code is lost.

When an entity wants to communicate via AppleTalk, it should call RegisterName to place its name and internet address in the names table. When an entity no longer wants to communicate on the network, or is being shut down, it should call RemoveName to remove its entry from the names table.

To determine the address of an entity you know only by name, call LookupName, which returns a list of all entities with the name you specify. If you already know the address of an entity, and want only to confirm that it still exists, call ConfirmName. ConfirmName is more efficient than LookupName in terms of network traffic.

NBP Routines

RegisterName function

Parameter block

-->	26	csCode	word	;always registerName
-->	28	interval	byte	;retry interval
<->	29	count	byte	;retry count
-->	30	ntQElPtr	pointer	;names table element ; pointer
-->	34	verifyFlag	byte	;set if verify needed

RegisterName adds the name and address of an entity to the node's names table. NTQElPtr points to a names table entry containing the entity's name and socket number (of the form shown in Figure 13). Metacharacters aren't allowed in the entity name. If verifyFlag is TRUE, RegisterName checks on the network to see if the name is already in use, and returns a result code of nbpDuplicate if so. Interval and count contain the retry interval in 8-tick units and the retry count. When a retry is made, the count field is modified.

(warning)

> The names table entry passed to RegisterName remains the property of NBP until removed from the names table. Don't attempt to remove or modify it. If you've allocated memory using the NewHandle call, you must lock it as long as the name is registered.

(warning)

> VerifyFlag should normally be set before calling RegisterName.

Result codes		
	noErr	No error
	nbpDuplicate	Duplicate name already exists
	nbpNISErr	Error opening names information socket

LookupName function

Parameter block

-->	26	csCode	word	;always lookupName
-->	28	interval	byte	;retry interval
<->	29	count	byte	;retry count
-->	3Ø	entityPtr	pointer	;entity name
<->	34	retBuffPtr	pointer	;pointer to buffer
-->	38	retBuffSize	word	;buffer size in bytes
-->	4Ø	maxToGet	word	;matches to get
<--	42	numGotten	word	;matches found

LookupName returns the addresses of all entities with a specified name. EntityPtr points to the entity's name (stored in the form shown in the bottom half of Figure 13 above). Meta-characters are allowed in the entity name. RetBuffPtr and RetBuffSize contain the location and size of an area of memory in which the tuples describing the entity's address should be returned. MaxToGet indicates the maximum number of matching names to find addresses for; the actual number of addresses found is returned in numGotten. Interval and count contain the retry interval and the retry count. LookupName completes when either the number of matches is equal to or greater than maxToGet, or the retry count has been exceeded. The count field is decremented for each retransmission.

(note)

> NumGotten is first set to Ø and then incremented with each match found. You can test the value in this field, and can start examining the received addresses in the buffer while the lookup continues.

Result codes		
	noErr	No error
	nbpBuffOvr	Buffer overflow

ConfirmName function

Parameter block

-->	26	csCode	word	;always confirmName
-->	28	interval	byte	;retry interval
<-->	29	count	byte	;retry count
-->	3Ø	entityPtr	pointer	;entity name
-->	34	confirmAddr	long word	;entity address
<--	38	newSocket	byte	;socket number

ConfirmName confirms that an entity known by name and address still exists (is still entered in the names directory). EntityPtr points to the entity's name (stored in the form shown in the bottom half of Figure 13 above). ConfirmAddr specifies the address to confirmed. No meta-characters are allowed in the entity name. Interval and count contain the retry interval and the retry count. The socket number of the entity is returned in newSocket. ConfirmName is more efficient than LookupName in terms of network traffic.

Result codes	noErr	No error
	nbpConfDiff	Name confirmed for different socket
	nbpNoConfirm	Name not confirmed

RemoveName function

Parameter block

-->	26	csCode	word	;always removeName
-->	3Ø	entityPtr	pointer	;entity name

RemoveName removes an entity name from the names table of the given entity's node.

Result codes	noErr	No error
	nbpNotFound	Name not found

LoadNBP function

Parameter block

-->	26	csCode	word	;always loadNBP

On a Macintosh 128K, LoadNBP reads the NBP code from the system resource file into memory; on a Macintosh 512K or XL it has no effect.

Result codes	noErr	No error

UnloadNBP function

```
       Parameter block
           -->  26   csCode         word        ;always unloadNBP
```

On a Macintosh 128K, UnloadNBP makes the NBP code purgeable; the space isn't actually released by the Memory Manager until necessary. On a Macintosh 512K or XL, UnloadNBP has no effect.

Result codes	noErr	No error

PROTOCOL HANDLERS AND SOCKET LISTENERS

This section describes how to write your own protocol handlers and socket listeners. If you're only interested in using the default protocol handlers and socket listeners provided by the Pascal interface, skip ahead to the summary. Protocol handlers and socket listeners must be written in assembly language because they will be called by the .MPP driver with parameters in various registers not directly accessible from Pascal.

The .MPP and .ATP drivers have been designed to maximize overall throughput while minimizing code size. Two principal sources of loss of throughput are unnecessary buffer copying and inefficient mechanisms for dispatching (routing) packets between the various layers of the network protocol architecture. The AppleTalk Manager completely eliminates buffer copying by using simple, efficient dispatching mechanisms at two important points of the data reception path: protocol handlers and socket listeners. To write your own you should understand the flow of control in this path.

Data Reception in the AppleTalk Manager

When the SCC detects a LAP frame addressed to the particular node (or a broadcast frame), it interrupts the Macintosh's MC68000. An interrupt handler built into the .MPP driver gets control and begins servicing the interrupt. Meanwhile, the frame's LAP header bytes are coming into the SCC's data reception buffer; this is a 3-byte FIFO buffer. The interrupt handler must remove these bytes from the SCC's buffer to make room for the bytes right behind; for this purpose, MPP has an internal buffer, known as the Read Header Area (RHA), into which it places these three bytes.

The third byte of the frame contains the LAP protocol type field. If the most significant bit of this field is set (that is, LAP protocol types 128 through 255), the frame is a LAP control frame. Since LAP control frames are only three bytes long (plus two CRC bytes), for such frames the interrupt handler simply confirms that the CRC bytes indicate an error-free frame and then performs the specified action.

If, however, the frame being received is a data frame (that is, LAP protocol types 1 through 127), intended for a higher layer of the protocol architecture implemented on that Macintosh, this means that additional data bytes are coming right behind. The interrupt handler must immediately pass control to the protocol handler corresponding to the protocol type specified in the third byte of the LAP frame for continued reception of the frame. To allow for such a dispatching mechanism, the LAP code in MPP maintains a protocol table. This consists of a list of currently used LAP protocol types with the memory addresses of their corresponding protocol handlers. To allow MPP to transfer control to a protocol handler you've written, you must make an appropriate entry in the protocol table with a valid LAP protocol type and the memory address of your code module.

To enter your protocol handler into the protocol table, issue the LAPOpenProtocol call from Pascal or an AttachPH call from assembly language. Thereafter, whenever a LAP header with your LAP protocol type is received, MPP will call your protocol handler. When you no longer wish to receive packets of that LAP protocol type, call LAPCloseProtocol from Pascal or DetachPH from assembly language.

(warning)

> Remember that LAP protocol types 1 and 2 are reserved by DDP for the default protocol handler and that types 128 through 255 are used by ALAP for its control frames.

A protocol handler is a piece of assembly-language code that controls the reception of AppleTalk packets of a given LAP protocol type. More specifically, a protocol handler must carry out the reception of the rest of the frame following the LAP header. The nature of a particular protocol handler depends on the characteristics of the protocol for which it was written. In the simplest case, the protocol handler simply reads the entire packet into an internal buffer. A more sophisticated protocol handler might read in the header of its protocol, and on the basis of information contained therein, decide where to put the rest of the packet's data. In certain cases, the protocol handler might, after examining the header corresponding to its own protocol, in turn transfer control to a similar piece of code at the next-higher level of the protocol architecture (for example, in the case of DDP, its protocol handler must call the socket listener of the datagram's destination socket).

In this way, protocol handlers are used to allow "on the fly" decisions as to the intended recipient of the packets's data, and thus avoid buffer copying. Using protocol handlers and their counterparts in higher layers (for instance, socket listeners), data sent over the AppleTalk network is read directly from the network into the destination's buffer.

Writing Protocol Handlers

When the .MPP driver calls your protocol handler, it has already read the first five bytes of the packet into the RHA. These are the three-byte LAP header and the next two bytes of the packet. The two bytes following the header must contain the length in bytes of the data in the packet, including these two bytes themselves, but excluding the LAP header.

(note)

> Since LAP packets can have at most 6ØØ data bytes, only the lower ten bits of this length value are significant.

After determining how many bytes to read and where to put them, the protocol handler must call one or both of two routines that perform all the low-level manipulation of the SCC required to read bytes from the network. ReadPacket can be called repeatedly to read in the packet piecemeal or ReadRest can be called to read the rest of the packet. Any number of ReadPacket calls can be used, as long as a ReadRest call is made to read the final piece of the packet. This is necessary because ReadRest restores state information and verifies that the hardware-generated CRC is correct. An error will be returned if the protocol handler attempts to use ReadPacket to read more bytes than remain in the packet.

When MPP passes control to your protocol handler, it passes various parameters and pointers in the processor's registers:

Register(s)	Contents
AØ-A1	SCC addresses used by MPP
A2	Pointer to MPP's local variables (discussed below)
A3	Pointer to next free byte in RHA
A4	Pointer to ReadPacket and ReadRest jump table
D1 (word)	Number of bytes left to read in packet

These registers, with the exception of A3, must be preserved until ReadRest is called. A3 is used as an input parameter to ReadPacket and ReadRest, so its contents may be changed. DØ, D2, and D3 are free for your use. In addition, register A5 has been saved by MPP and may be used by the protocol handler until ReadRest is called. When control returns to the protocol handler from ReadRest, MPP no longer needs the data in these registers. At that point, standard interrupt routine conventions apply and the protocol handler can freely use AØ-A3 and DØ-D3 (they are restored by the interrupt handler).

D1 contains the number of bytes left to be read in the packet as derived from the packet's length field. A transmission error could corrupt the length field or some bytes in the packet might be lost, but this won't be discovered until the end of the packet is reached and the CRC checked.

When the protocol handler is first called, the first five bytes of the packet (LAP destination node ID, source number, LAP protocol type, and length) can be read from the RHA. Since A3 is pointing to the next free position in the RHA, these bytes can be read using negative offsets from A3. For instance, the LAP source node ID is at -4(A3), the packet's data length (given in D1) is also pointed to by -2(A3), and so on. Alternatively, they can be accessed as positive offsets from the top of the RHA. The effective address of the top of the RHA is toRHA(A2), so the following code could be used to obtain the LAP type field:

```
          LEA       toRHA(A2),A5        ;A5 points to top of RHA
          MOVE.B    lapType(A5),D2      ;load D2 with type field
```

These methods are valid only as long as SCC interrupts remain locked out (which they are when the protocol handler is first called). If the protocol handler lowers the interrupt level, another packet could arrive over the network and invalidate the contents of the RHA.

You can call ReadPacket by jumping through the jump table in the following way:

```
      JSR  (A4)

      On entry   D3:  number of bytes to be read (word)
                 A3:  pointer to a buffer to hold the bytes

      On exit    DØ:  modified
                 D1:  number of bytes left to read in packet (word)
                 D2:  preserved
                 D3:  = Ø if requested number of bytes were read
                      <>Ø if error
                 AØ-A2:  preserved
                 A3:  pointer to one byte past the last byte read
```

ReadPacket reads the number of bytes specified in D3 into the buffer pointed to by A3. The number of bytes remaining to be read in the packet is returned in D1. A3 points to the byte following the last byte read.

You can call ReadRest by jumping through the jump table in the following way:

```
        JSR  2(A4)

        On entry     A3:  pointer to a buffer to hold the bytes
                     D3:  size of the buffer (word)

        On exit      DØ-D1:  modified
                     D2:  preserved
                     D3:  = Ø if packet was exactly the size of the buffer
                          < Ø if packet was (-D3) bytes too large to fit
                              in buffer and was truncated
                          > Ø if D3 bytes weren't read (packet is smaller
                              than buffer)
                     AØ-A2:  preserved
                     A3:  pointer to one byte past the last byte read
```

ReadRest reads the remaining bytes of the packet into the buffer whose size is given in D3 and whose location is pointed to by A3. The result of the operation is returned in D3.

ReadRest can be called with D3 set to a buffer size greater than the packet size; ReadPacket may not (and will return an error if it is).

(warning)

> Remember to always call ReadRest to read the last part of a packet; otherwise the system will eventually crash.

If at any point before it has read the last byte of a packet, the protocol handler wants to discard the remaining data, it should terminate by calling ReadRest as follows:

```
             MOVEQ    #Ø, D3     ;byte count of zero
             JSR      2(A4)      ;call ReadRest
             RTS
```

Or, equivalently:

```
             MOVEQ    #Ø, D3     ;byte count of zero
             JMP      2(A4)      ;JMP to ReadRest, not JSR
```

In all other cases, the protocol handler should end with an RTS, even if errors were detected. If MPP returns an error from a ReadPacket call, the protocol handler must quit via an RTS without calling ReadRest at all (in this case it has already been called by MPP).

The Zero bit of the condition codes is set upon return from these routines to indicate the presence of errors (CRC, overrun, and so on). Zero bit set means no error was detected; a nonzero condition code implies an error of some kind.

Up to 24 bytes of temporary storage are available in MPP's RHA. When the protocol handler is called, 19 of these bytes are free for its use.

It may read several bytes (at least four are suggested) into this area to empty the SCC's buffer and buy some time for further processing.

MPP's globals include some variables that you may find useful. They're allocated as a block of memory pointed to by the contents of the global variable ABusVars, but a protocol handler can access them by offsets from A2:

Name	Contents
sysLAPAddr	This node's node ID (byte)
toRHA	Top of the Read Header Area (24 bytes)
sysABridge	Node ID of a bridge (byte)
sysNetNum	This node's network number (word)
vSCCEnable	Status Register (SR) value to re-enable SCC interrupts (word)

(warning)

> Under no circumstances should your protocol handler modify these variables. Your protocol handler can read them to find the node's ID, network number, and the node ID of a bridge on the AppleTalk internet.

If, after reading the entire packet from the network and using the data in the RHA, the protocol handler needs to do extensive post-processing, it can load the value in vSCCEnable into the SR to enable interrupts. To allow your programs to run transparently on any Macintosh, use the value in vSCCEnable rather than directly manipulating the interrupt level by changing specific bits in the SR.

Additional information, such as the driver's version number or reference number and a pointer (or handle) to the driver itself, may be obtained from MPP's device control entry. This can be found by dereferencing the handle in the unit table's entry corresponding to unit number 9; for more information, see the section "The Structure of a Device Driver" in the Device Manager manual.

Timing Considerations

Once it's been called by MPP, your protocol handler has complete responsibility for receiving the rest of the packet. The operation of your protocol handler is time-critical. Since it's called just after MPP has emptied the SCC's 3-byte buffer, the protocol handler has approximately 95 microseconds (best case) before it must call ReadPacket or ReadRest. Failure to do so will result in an overrun of the SCC's buffer and loss of packet information. If, within that time, the protocol handler can't determine where to put the entire incoming packet, it should call ReadPacket to read at least four bytes into some private buffer (possibly the RHA). Doing this will again empty the SCC's buffer and buy another 95 microseconds. You can do this as often as necessary, as long as the processing time between successive calls to ReadPacket doesn't exceed 95 microseconds.

Writing Socket Listeners

A socket listener is a piece of assembly-language code that receives datagrams delivered by the DDP built-in protocol handler and delivers them to the client owning that socket.

When a datagram (a packet with LAP protocol type 1 or 2) is received by the LAP, DDP's built-in protocol handler is called. This handler reads the DDP header into the RHA, examines the destination socket number, and determines if this socket is open by searching DDP's socket table. This table lists the socket number and corresponding socket listener address for each open socket. If an entry is found matching the destination socket, the protocol handler immediately transfers control to the appropriate socket listener. (To allow DDP to recognize and branch to a socket listener you've written, call DDPOpenSocket from Pascal or OpenSkt from assembly language.)

At this point, the registers are set up as follows:

Register(s)	Contents
AØ-A1	SCC addresses used by MPP
A2	Pointer to MPP's local variables (discussed above)
A3	Pointer to next free byte in RHA
A4	Pointer to ReadPacket and ReadRest jump table
DØ	This packet's destination socket number
D1	Number of bytes left to read in packet (word)

The entire LAP and DDP headers are in the RHA; these are the only bytes of the packet that have been read in from the SCC's buffer. The socket listener can get the destination socket number from DØ to select a buffer into which the packet can be read. The listener then calls ReadPacket and ReadRest as described in the section "Writing Protocol Handlers" above. The timing considerations discussed in that section apply as well, as do the issues related to accessing the MPP local variables.

The socket listener may examine the LAP and DDP headers to extract the various fields relevant to its particular client's needs. To do so, it must first examine the LAP protocol type field (three bytes from the beginning of the RHA) to decide whether a short (LAP protocol type=1) or long (LAP protocol type=2) header has been received.

A long DDP header containing a nonzero checksum field implies that the datagram was checksummed at the source. In this case, the listener can recalculate the checksum using the received datagram, and compare it with the checksum value. The following subroutine can be used for this purpose:

```
DoChkSum    ;
            ;  D1 (word) = number of bytes to checksum
            ;  D3 (word) = current checksum
            ;  A1 points to the bytes to checksum
            ;
            CLR.W    DØ          ;clear high byte
            SUBQ.W   #1,D1       ;decrement count for DBRA
Loop        MOVE.B   (A1)+,DØ    ;read a byte into DØ
            ADD.W    DØ,D3       ;accumulate checksum
            ROL.W    #1,D3       ;rotate left one bit
            DBRA     D1,Loop     ;loop if more bytes
            RTS
```

(note)
DØ is modified by DoChkSum.

The checksum must be computed for all bytes starting with the DDP header byte following the checksum field up to the last data byte (not including the CRC bytes). The socket listener must start by first computing the checksum for the DDP header fields in the RHA. This is done as follows:

```
            CLR.W    D3          ;set checksum to Ø
            MOVEQ    #ddpHSzLong-ddpDstNet,D1
                                 ;length of header part
                                 ; to checksum
            LEA      toRHA+lapHdSz+ddpDstNet(A2),A1
                                 ;point to destination
                                 ; network number
            JSR      DoChkSum
            ;  D3 = accumulated checksum of DDP header part
```

The socket listener must now continue to set up D1 and A1 for each subsequent portion of the datagram, and call DoChkSum for each. It must not alter the value in D3.

The situation of the calculated checksum being equal to Ø requires special attention. For such packets, the source sends a value of -1 to distinguish them from unchecksummed packets. At the end of its checksum computation, the socket listener must examine the value in D3 to see if it's Ø. If so, it's converted to -1 and compared with the received checksum to determine if there was a checksum error:

```
            TST.W    D3          ;is calculated value zero?
            BNE.S    @1          ;no -- go and use it
            SUBQ.W   #1,D3       ;it is zero; make it -1
@1          CMP.W    toRHA+lapHdSz+ddpCheckSum(A2),D3
            BNE      ChkSumError
            ...
```

SUMMARY OF THE APPLETALK MANAGER

Constants

```
CONST lapSize = 2Ø; {ABusRecord size for ALAP}
      ddpSize = 26; {ABusRecord size for DDP}
      nbpSize = 26; {ABusRecord size for NBP}
      atpSize = 56; {ABusRecord size for ATP}
```

Data Types

```
TYPE ABProtoType = (lapProto,ddpProto,nbpProto,atpProto);

     ABRecHandle = ^ABRecPtr;
     ABRecPtr    = ^ABusRecord;
     ABusRecord  =
       RECORD
         abOpcode:         ABCallType; {type of call}
         abResult:         INTEGER;    {result code}
         abUserReference: LONGINT;     {for your use}
         CASE ABProtoType OF
          lapProto:
           (lapAddress:  LAPAdrBlock; {destination or source node ID}
            lapReqCount: INTEGER;     {length of frame data or }
                                      { buffer size in bytes}
            lapActCount: INTEGER;     {number of frame data bytes}
                                      { actually received}
            lapDataPtr:  Ptr);        {pointer to frame data or }
                                      { pointer to buffer}
          ddpProto:
           (ddpType:     Byte;        {DDP protocol type}
            ddpSocket:   Byte;        {source or listening socket }
                                      { number}
            ddpAddress:  AddrBlock;   {destination or source }
                                      { socket address}
            ddpReqCount: INTEGER;     {length of datagram data or }
                                      { buffer size in bytes}
            ddpActCount: INTEGER;     {number of bytes actually }
                                      { received}
            ddpDataPtr:  Ptr;         {pointer to buffer}
            ddpNodeID:   Byte);       {original destination node ID}
          nbpProto:
           (nbpEntityPtr:      EntityPtr;     {pointer to entity name}
            nbpBufPtr:         Ptr;           {pointer to buffer}
            nbpBufSize:        INTEGER;       {buffer size in bytes}
            nbpDataField:      INTEGER;       {number of addresses }
                                              { or socket number}
           nbpAddress:        AddrBlock;     {socket address}
           nbpRetransmitInfo: RetransType);  {retransmission }
                                             { information}
         atpProto:
          (atpSocket:    Byte;          {listening or responding socket }
                                        { number}
           atpAddress:   AddrBlock;     {destination or source }
                                        { socket address}
           atpReqCount:  INTEGER;       {request size or buffer }
                                        { size in bytes}
           atpDataPtr:   Ptr;           {pointer to request buffer}
           atpRspBDSPtr: BDSPtr;        {pointer to response BDS}
           atpBitMap:    BitMapType;    {transaction bit map}
           atpTransID:   INTEGER;       {transaction ID}
           atpActCount:  INTEGER;       {number of bytes actually }
                                        { received}
           atpUserData:  LONGINT;       {user bytes}
           atpXO:        BOOLEAN;       {exactly-once flag}
           atpEOM:       BOOLEAN;       {end-of-message flag}
           atpTimeOut:   Byte;          {retry timeout interval in seconds}
           atpRetries:   Byte;          {number of retries}
           atpNumBufs:   Byte;          {number of elements in response }
                                        { BDS or number of response }
                                        { packets sent}
           atpNumRsp:    Byte;          {number of response packets }
                                        { received or sequence number}
           atpBDSSize:   Byte;          {number of elements in }
                                        { response BDS}
           atpRspUData:  LONGINT;       {user bytes sent or received }
                                        { in transaction response}
           atpRspBuf:    Ptr;           {pointer to response message }
                                        { buffer}
           atpRspSize:   INTEGER);      {size in bytes of response }
                                        { message buffer}
     END;

  ABCallType = (tLAPRead,tLAPWrite,tDDPRead,tDDPWrite,
                tNBPLookup,tNBPConfirm,tNBPRegister,
                tATPSndRequest,tATPGetRequest,tATPSndRsp,
                tATPAddRsp,tATPRequest,tATPResponse);

  LAPAdrBlock = PACKED RECORD
                  dstNodeID:   Byte;   {destination node ID}
                  srcNodeID:   Byte;   {source node ID}
                  LAPProtType: ABByte  {LAP protocol type}
                END;

  ABByte = 1..127; {LAP protocol type}

  AddrBlock = PACKED RECORD
                aNet:    INTEGER; {network number}
                aNode:   Byte;    {node ID}
                aSocket: Byte     {socket number}
              END;
     BDSPtr     = ^BDSType;
     BDSType    = ARRAY[Ø..7] OF BDSElement; {response BDS}
     BDSElement = RECORD
                    buffSize:  INTEGER; {buffer size in bytes}
                    buffPtr:   Ptr;     {pointer to buffer}
                    dataSize:  INTEGER; {packet size}
                    userBytes: LONGINT  {user bytes}
                  END;

     BitMapType = PACKED ARRAY[Ø..7] OF BOOLEAN;

     EntityPtr  = ^EntityName;
     EntityName = RECORD
                    objStr:  Str32; {object}
                    typeStr: Str32; {type}
                    zoneStr: Str32  {zone}
                  END;

     Str32 = STRING[32];

     RetransType = PACKED RECORD
                     retransInterval: Byte; {retransmit interval }
                                            { in 8-tick units}
                     retransCount:    Byte  {number of attempts}
                   END;
```

Routines [Not in ROM]

Opening and Closing AppleTalk

```
FUNCTION MPPOpen :  OSErr;
FUNCTION MPPClose : OSErr;
```

AppleTalk Link Access Protocol

```
FUNCTION LAPOpenProtocol  (theLAPType: ABByte; protoPtr: Ptr) :
                           OSErr;
FUNCTION LAPCloseProtocol (theLAPType: ABByte) : OSErr;

FUNCTION LAPWrite (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
    <--   abOpcode                  {always tLapWrite}
    <--   abResult                  {result code}
    -->   abUserReference           {for your use}
    -->   lapAddress.dstNodeID      {destination node ID}
    -->   lapAddress.lapProtType    {LAP protocol type}
    -->   lapReqCount               {length of frame data}
    -->   lapDataPtr                {pointer to frame data}
```

```
FUNCTION LAPRead (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
    <--   abOpcode                 {always tLapRead}
    <--   abResult                 {result code}
    -->   abUserReference          {for your use}
    <--   lapAddress.dstNodeID     {destination node ID}
    <--   lapAddress.srcNodeID     {source node ID}
    -->   lapAddress.lapProtType   {LAP protocol type}
    -->   lapReqCount              {buffer size in bytes}
    <--   lapActCount              {number of frame data bytes actually }
                                   { received}
    -->   lapDataPtr               {pointer to buffer}

FUNCTION LAPRdCancel (abRecord: ABRecHandle) : OSErr;
```

Datagram Delivery Protocol

```
FUNCTION DDPOpenSocket  (VAR theSocket: Byte; sktListener: Ptr) :
                         OSErr;
FUNCTION DDPCloseSocket (theSocket: Byte) : OSErr;

FUNCTION DDPWrite (abRecord: ABRecHandle; doChecksum: BOOLEAN;
                   async: BOOLEAN) : OSErr;
    <--   abOpcode          {always tDDPWrite}
    <--   abResult          {result code}
    -->   abUserReference   {for your use}
    -->   ddpType           {DDP protocol type}
    -->   ddpSocket         {source socket number}
    -->   ddpAddress        {destination socket address}
    -->   ddpReqCount       {length of datagram data}
    -->   ddpDataPtr        {pointer to buffer}

FUNCTION DDPRead (abRecord: ABRecHandle; retCksumErrs: BOOLEAN;
                  async: BOOLEAN) : OSErr;
    <--   abOpcode          {always tDDPRead}
    <--   abResult          {result code}
    -->   abUserReference   {for your use}
    -->   ddpType           {DDP protocol type}
    -->   ddpSocket         {listening socket number}
    <--   ddpAddress        {source socket address}
    -->   ddpReqCount       {buffer size in bytes}
    <--   ddpActCount       {number of bytes actually received}
    -->   ddpDataPtr        {pointer to buffer}
    <--   ddpNodeID         {original destination node ID}

FUNCTION DDPRdCancel (abRecord: ABRecHandle) : OSErr;
```

AppleTalk Transaction Protocol

```
FUNCTION ATPLoad :       OSErr;
FUNCTION ATPUnload :     OSErr;
FUNCTION ATPOpenSocket  (addrRcvd: AddrBlock; VAR atpSocket: Byte) :
                         OSErr;
```

```
FUNCTION ATPCloseSocket (atpSocket: Byte) : OSErr;

FUNCTION ATPSndRequest (abRecord: ABRecHandle; async: BOOLEAN) :
                        OSErr;
    <--   abOpcode          {always tATPSndRequest}
    <--   abResult          {result code}
    -->   abUserReference   {for your use}
    -->   atpAddress        {destination socket address}
    -->   atpReqCount       {request size in bytes}
    -->   atpDataPtr        {pointer to buffer}
    -->   atpRspBDSPtr      {pointer to response BDS}
    -->   atpUserData       {user bytes}
    -->   atpXO             {exactly-once flag}
    <--   atpEOM            {end-of-message flag}
    -->   atpTimeOut        {retry timeout interval in seconds}
    -->   atpRetries        {maximum number of retries}
    -->   atpNumBufs        {number of elements in response BDS}
    <--   atpNumRsp         {number of response packets actually }
                            { received}

FUNCTION ATPRequest (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
    <--   abOpcode          {always tATPRequest}
    <--   abResult          {result code}
    -->   abUserReference   {for your use}
    -->   atpAddress        {destination socket address}
    -->   atpReqCount       {request size in bytes}
    -->   atpDataPtr        {pointer to buffer}
    <--   atpActCount       {number of bytes actually received }
    -->   atpUserData       {user bytes}
    -->   atpXO             {exactly-once flag}
    <--   atpEOM            {end-of-message flag}
    -->   atpTimeOut        {retry timeout interval in seconds}
    -->   atpRetries        {maximum number of retries}
    <--   atpRspUData       {user bytes received in transaction }
                            { response}
    -->   atpRspBuf         {pointer to response message buffer}
    -->   atpRspSize        {size of response message buffer}

FUNCTION ATPReqCancel (abRecord: ABRecHandle; async: BOOLEAN) :
                       OSErr;

FUNCTION ATPGetRequest (abRecord: ABRecHandle; async: BOOLEAN) :
                        OSErr;
    <--   abOpcode          {always tATPGetRequest}
    <--   abResult          {result code}
    -->   abUserReference   {for your use}
    -->   atpSocket         {listening socket number}
    <--   atpAddress        {source socket address}
    -->   atpReqCount       {buffer size in bytes}
    -->   atpDataPtr        {pointer to buffer}
    <--   atpBitMap         {transaction bit map}
    <--   atpTransID        {transaction ID}
    <--   atpActCount       {number of bytes actually received}
    <--   atpUserData       {user bytes}
    <--   atpXO             {exactly-once flag}

FUNCTION ATPSndRsp (abRecord: ABRecHandle; async: BOOLEAN) :
                   OSErr;
    <--   abOpcode          {always tATPSndRsp}
    <--   abResult          {result code}
    -->   abUserReference   {for your use}
    -->   atpSocket         {responding socket number}
    -->   atpAddress        {destination socket address}
    -->   atpRspBDSPtr      {pointer to response BDS}
    -->   atpTransID        {transaction ID}
    -->   atpEOM            {end-of-message flag}
    -->   atpNumBufs        {number of response packets being sent}
    -->   atpBDSSize        {number of elements in response BDS}

FUNCTION ATPAddRsp (abRecord: ABRecHandle) : OSErr;
    <--   abOpcode          {always tATPAddRsp}
    <--   abResult          {result code}
    -->   abUserReference   {for your use}
    -->   atpSocket         {responding socket number}
    -->   atpAddress        {destination socket address}
    -->   atpReqCount       {buffer size in bytes}
    -->   atpDataPtr        {pointer to buffer}
    -->   atpTransID        {transaction ID}
    -->   atpUserData       {user bytes}
    -->   atpEOM            {end-of-message flag}
    -->   atpNumRsp         {sequence number}

FUNCTION ATPResponse (abRecord: ABRecHandle; async: BOOLEAN) :
                     OSErr;
    <--   abOpcode          {always tATPResponse}
    <--   abResult          {result code}
    -->   abUserReference   {for your use}
    -->   atpSocket         {responding socket number}
    -->   atpAddress        {destination socket address}
    -->   atpRspUData       {user bytes sent in transaction }
                            { response}
    -->   atpRspBuf         {pointer to response message buffer}
    -->   atpRspSize        {size of response message buffer}

FUNCTION ATPRspCancel (abRecord: ABRecHandle; async: BOOLEAN) :
                      OSErr;
```

Name-Binding Protocol

```
FUNCTION NBPRegister (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
    <--   abOpcode            {always tNBPRegister}
    <--   abResult            {result code}
    -->   abUserReference     {for your use}
    -->   nbpEntityPtr        {pointer to entity name}
    -->   nbpBufPtr           {pointer to buffer}
    -->   nbpBufSize          {buffer size in bytes}
    -->   nbpAddress.aSocket  {socket address}
    -->   nbpRetransmitInfo    {retransmission information}

FUNCTION NBPLookup (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
    <--   abOpcode            {always tNBPLookup}
    <--   abResult            {result code}
    -->   abUserReference     {for your use}
    -->   nbpEntityPtr        {pointer to entity name}
    -->   nbpBufPtr           {pointer to buffer}
    -->   nbpBufSize          {buffer size in bytes}
    <->   nbpDataField        {number of addresses received}
    -->   nbpRetransmitInfo   {retransmission information}

FUNCTION NBPExtract (theBuffer: Ptr; numInBuf: INTEGER; whichOne:
                     INTEGER; VAR abEntity: EntityName; VAR address:
                     AddrBlock) : OSErr;

FUNCTION NBPConfirm (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
    <--   abOpcode            {always tNBPConfirm}
    <--   abResult            {result code}
    -->   abUserReference     {for your use}
    -->   nbpEntityPtr        {pointer to entity name}
    <--   nbpDataField        {socket number}
    -->   nbpAddress          {socket address}
    -->   nbpRetransmitInfo   {retransmission information}

FUNCTION NBPRemove  (abEntity: EntityPtr) : OSErr;
FUNCTION NBPLoad :   OSErr;
FUNCTION NBPUnload : OSErr;
```

Miscellaneous Routines

```
FUNCTION GetNodeAddress (VAR myNode,myNet: INTEGER) : OSErr;
FUNCTION IsMPPOpen :     BOOLEAN;
FUNCTION IsATPOpen :     BOOLEAN;
```

Assembly-Language Information

Constants

```
; Serial port use types

useFree      .EQU  Ø   ;use undefined
useATalk     .EQU  1   ;AppleTalk
useASync     .EQU  2   ;async

; Bit in PortBUse for .ATP driver status

atpLoadedBit  .EQU   4   ;set if .ATP driver is opened
```

```
; CsCode values for Control calls (MPP)

writeLAP      .EQU   242
detachPH      .EQU   243
attachPH      .EQU   245
writeDDP      .EQU   246
closeSkt      .EQU   247
openSkt       .EQU   248
loadNBP       .EQU   249
confirmName   .EQU   25Ø
lookupName    .EQU   251
removeName    .EQU   252
registerName  .EQU   253
killNBP       .EQU   254
unloadNBP     .EQU   255

; CsCode values for Control calls (ATP)

relRspCB      .EQU   249
closeATPSkt   .EQU   25Ø
addResponse   .EQU   251
sendResponse  .EQU   252
getRequest    .EQU   253
openATPSkt    .EQU   254
sendRequest   .EQU   255
relTCB        .EQU   256

; ALAP header

lapDstAdr   .EQU   Ø   ;destination node ID
lapSrcAdr   .EQU   1   ;source node ID
lapType     .EQU   2   ;LAP protocol type

; ALAP header size

lapHdSz     .EQU   3

; LAP protocol type values

shortDDP   .EQU   1   ;short DDP header
longDDP    .EQU   2   ;long DDP header

; Long DDP header

ddpHopCnt     .EQU   Ø           ;hop count (4 bits)
ddpLength     .EQU   Ø           ;datagram length (1Ø bits)
ddpChecksum   .EQU   2           ;checksum
ddpDstNet     .EQU   4           ;destination network number
ddpSrcNet     .EQU   6           ;source network number
ddpDstNode    .EQU   8           ;destination node ID
ddpSrcNode    .EQU   9           ;source node ID
ddpDstSkt     .EQU   1Ø          ;destination socket number
ddpSrcSkt     .EQU   11          ;source socket number
ddpType       .EQU   12          ;DDP protocol type
```

```
; DDP long header size

ddpHSzLong    .EQU   ddpType+1

; Short DDP header

ddpLength     .EQU   Ø              ;datagram length
sDDPDstSkt    .EQU   ddpChecksum    ;destination socket number
sDDPSrcSkt    .EQU   sDDPDstSkt+1   ;source socket number
sDDPType      .EQU   sDDPSrcSkt+1   ;DDP protocol type

;DDP short header size

ddpHSzShort   .EQU   sDDPType+1

; Mask for datagram length

lengthMask   .EQU   $Ø3FF

; Maximum size of DDP data

ddpMaxData     .EQU   586

; ATP header

atpControl    .EQU   Ø   ;control information
atpBitMap     .EQU   1   ;bit map
atpRespNo     .EQU   1   ;sequence number
atpTransID    .EQU   2   ;transaction ID
atpUserData   .EQU   4   ;user bytes

; ATP header size

atpHdSz       .EQU   8

; DDP protocol type for ATP packets

atp          .EQU   3

; ATP function code

atpReqCode   .EQU    $4Ø   ;TReq packet
atpRspCode   .EQU    $8Ø   ;TResp packet
atpRelCode   .EQU    $CØ   ;TRel packet

; ATPFlags control information bits

sendChk       .EQU   Ø     ;send-checksum bit
tidValid      .EQU   1     ;transaction ID validity bit
atpSTSBit     .EQU   3     ;send-transmission-status bit
atpEOMBit     .EQU   4     ;EOM bit of control information
atpXOBit      .EQU   5     ;exactly-once bit
```

```
; Maximum number of ATP request packets

atpMaxNum    .EQU   8

; ATP buffer data structure

bdsBuffSz      .EQU   Ø     ;length of data to send or buffer size
bdsBuffAddr    .EQU   2     ;pointer to data or buffer
bdsDataSz      .EQU   6     ;number of bytes actually received
bdsUserData    .EQU   8     ;for your use

; BDS element size

bdsEntrySz     .EQU   12

; NBP packet

nbpControl    .EQU   Ø   ;NBP call
nbpTCount     .EQU   Ø   ;tuple count
nbpID         .EQU   1   ;packet identifier
nbpTuple      .EQU   2   ;start of first tuple

; DDP protocol type for NBP packet

nbp   .EQU   2   ;DDP protocol type for NBP packets

; NBP packet types

brRq         .EQU   1   ;broadcast request
lkUp         .EQU   2   ;lookup request
lkUpReply    .EQU   3   ;lookup reply

; NBP tuple

tupleNet      .EQU    Ø    ;network number
tupleNode     .EQU    2    ;node ID
tupleSkt      .EQU    3    ;socket number
tupleEnum     .EQU    4    ;enumerator
tupleName     .EQU    5    ;entity name

;Maximum number of tuples in NBP packet

tupleMax      .EQU    15

; NBP meta-characters

equals      .EQU   '='   ;"wild-card" meta-character
star        .EQU   '*'   ;"this zone" meta-character
```

```
; NBP names table entry

ntLink      .EQU   Ø                     ;pointer to next entry
ntTuple     .EQU   ntLink+4              ;tuple
ntSocket    .EQU   ntTuple+tupleSkt      ;socket number
ntEntity    .EQU   ntTuple+tupleName     ;entity name

; NBP names information socket

nis         .EQU   2   ;names information socket number
```

Routines

Link Access Protocol

```
WriteLAP function
    -->   26   csCode        word         ;always writeLAP
    -->   3Ø   wdsPointer    pointer      ;write data structure

AttachPH function
    -->   26   csCode        word         ;always attachPH
    -->   28   protType      byte         ;LAP protocol type
    -->   3Ø   handler       pointer      ;protocol handler

DetachPH function
    -->   26   csCode        word         ;always detachPH
    -->   28   protType      byte         ;LAP protocol type
```

Datagram Delivery Protocol

```
OpenSkt function
    -->   26   csCode          word        ;always openSkt
    <->   28   socket          byte        ;socket number
    -->   3Ø   listener        pointer     ;socket listener

CloseSkt function
    -->   26   csCode          word        ;always closeSkt
    -->   28   socket          byte        ;socket number

WriteDDP function
    -->   26   csCode          word        ;always writeDDP
    -->   28   socket          byte        ;socket number
    -->   29   checksumFlag    byte        ;checksum flag
    -->   3Ø   wdsPointer      pointer     ;write data structure
```

AppleTalk Transaction Protocol

```
OpenATPSocket function
    -->   26   csCode       word      ;always openATPSocket
    <->   28   atpSocket    byte      ;socket number
    -->   3Ø   addrBlock    long word ;socket request specification

CloseATPSocket function
    -->   26   csCode       word      ;always closeATPSocket
    -->   28   atpSocket    byte      ;socket number

SendRequest function
    -->   18   userData      long word ;user bytes
    -->   26   csCode        word      ;always sendRequest
    <--   28   currBitMap    byte      ;bit map
    <->   29   atpFlags      byte      ;control information
    -->   3Ø   addrBlock     long word ;destination socket address
    -->   36   reqLength     word      ;request size in bytes
    -->   36   reqPointer    pointer   ;pointer to request data
    <->   4Ø   bdsPointer    pointer   ;response BDS
    -->   44   numOfBuffs    byte      ;number of responses
                                       ; expected
    -->   45   timeOutVal    byte      ;timeout interval
    <--   46   numOfResps    byte      ;number of responses
                                       ; actually received
    -->   47   retryCount    byte      ;number of retries

GetRequest function
    <--   18   user Data     long word ;user bytes
    -->   26   csCode        word      ;always getRequest
    -->   28   atpSocket     byte      ;socket number
    <--   29   atpFlags      byte      ;control information
    <--   3Ø   addrBlock     long word ;source of request
    <->   34   reqLength     word      ;request buffer size in
                                       ; bytes
    -->   36   reqPointer    pointer   ;pointer to request buffer
    <--   44   bitMap        byte      ;bit map
    <--   46   transID       word      ;transaction ID

SendResponse function
    <--   18   userData       long word  ;user bytes from TRel
    -->   26   csCode         word       ;always sendResponse
    -->   28   atpSocket      byte       ;socket number
    -->   29   atpFlags       byte       ;control information
    -->   3Ø   addrBlock      long word  ;response destination
    -->   4Ø   bdsPointer     pointer    ;pointer to response BDS
    -->   44   numOfBuffs     byte       ;number of responses
    -->   45   bdsSize        byte       ;BDS size in elements
    -->   46   transID        word       ;transaction ID
```

```
AddResponse function
    -->   18   userData      long word ;user bytes
    -->   26   csCode        word      ;always addResponse
    -->   28   atpSocket     byte      ;socket number
    -->   29   atpFlags      byte      ;control information
    -->   30   addrBlock     long word ;response destination
    -->   36   reqLength     word      ;response size in bytes
    -->   36   reqPointer    pointer   ;pointer to response
    -->   44   rspNum        byte      ;sequence number
    -->   46   transID       word      ;transaction ID
```

Name-Binding Protocol

```
RegisterName function
    -->   26   csCode         word        ;always registerName
    -->   28   interval       byte        ;retry interval
    -->   29   count          byte        ;retry count
    -->   30   ntQElPtr       pointer     ;names table element
                                          ; pointer
    -->   34   verifyFlag     byte        ;set if verify

LookupName function
    -->   26   csCode         word        ;always lookupName
    -->   28   interval       byte        ;retry interval
    -->   29   count          byte        ;retry count
    -->   30   entityPtr      pointer     ;entity name
    -->   34   retBuffPtr     pointer     ;pointer to buffer
    <--   38   retBuffSize    word        ;buffer size in bytes
    -->   40   maxToGet       word        ;matches to get
    <--   42   numGotten      word        ;matches found

ConfirmName function
    -->   26   csCode         word        ;always confirmName
    -->   28   interval       byte        ;retry interval
    -->   29   count          byte        ;retry count
    -->   30   entityPtr      pointer     ;entity name
    -->   34   confirmAddr    pointer     ;entity address
    <--   38   newSocket      byte        ;socket number

RemoveName function
    -->   26   csCode         word        ;always removeName
    -->   30   entityPtr      pointer     ;entity pointer

LoadNBP function
    -->   26   csCode         word        ;always loadNBP

UnloadNBP function
    -->   26   csCode         word        ;always unloadNBP
```

Variables

SPConfig	Configuration of serial ports (byte) (bits Ø-3: current configuration of serial port B bits 4-6: current configuration of serial port A)
PortBUse	Current availability of serial port B (byte) (bit 7: 1=not in use, Ø=in use bits Ø-3: current use of port bits 4-6: driver-specific)
ABusVars	Pointer to AppleTalk variables

Result Codes

Name	Value	Meaning
atpBadRsp	-31Ø7	Bad response from ATPRequest
atpLenErr	-31Ø6	ATP response message too large
badATPSkt	-1Ø99	ATP bad responding socket
badBuffNum	-11ØØ	ATP bad sequence number
buf2SmallErr	-31Ø1	ALAP frame too large for buffer DDP datagram too large for buffer
cbNotFound	-11Ø2	ATP control block not found
ckSumErr	-31Ø3	DDP bad checksum
ddpLenErr	-92	DDP datagram or LAP data length too big
ddpSktErr	-91	DDP socket error: socket already active; not a well-known socket; socket table full; all dynamic socket numbers in use
excessCollsns	-95	ALAP no CTS received after 32 RTS's, or line sensed in use 32 times (not necessarily caused by collisions)
extractErr	-31Ø4	NBP can't find tuple in buffer
lapProtErr	-94	ALAP error attaching/detaching LAP protocol type: attach error when LAP protocol type is negative, already in table, or when table is full; detach error when LAP protocol type isn't in table
nbpBuffOvr	-1Ø24	NBP buffer overflow
nbpConfDiff	-1Ø26	NBP name confirmed for different socket
nbpDuplicate	-1Ø27	NBP duplicate name already exists
nbpNISErr	-1Ø29	NBP names information socket error
nbpNoConfirm	-1Ø25	NBP name not confirmed
nbpNotFound	-1Ø28	NBP name not found
noBridgeErr	-93	No bridge found
noDataArea	-11Ø4	Too many outstanding ATP calls
noErr	Ø	No error
noMPPError	-31Ø2	MPP driver not installed
noRelErr	-11Ø1	ATP no release received
noSendResp	-11Ø3	ATPAddRsp issued before ATPSndRsp
portInUse	-97	Driver Open error, port already in use
portNotCf	-98	Driver Open error, port not configured for this connection
readQErr	-31Ø5	Socket or protocol type invalid or not found in table

recNotFnd	-31Ø8	ABRecord not found
reqAborted	-11Ø5	Request aborted
reqFailed	-1Ø96	ATPSndRequest failed: retry count exceeded
sktClosedErr	-31Ø9	Asynchronous call aborted because socket was closed before call was completed
tooManyReqs	-1Ø97	ATP too many concurrent requests
tooManySkts	-1Ø98	ATP too many responding sockets

GLOSSARY

ALAP: See AppleTalk Link Access Protocol.

ALAP frame: A packet of data transmitted and received by ALAP.

alias: A different name for the same entity.

AppleTalk address: A socket's number and its node ID number.

AppleTalk Link Access Protocol (ALAP): The lowest-level protocol in the AppleTalk architecture, managing node-to-node delivery of frames on a single AppleTalk network.

AppleTalk Transaction Protocol (ATP): An AppleTalk protocol that's a DDP client. It allows one ATP client to request another ATP client to perform some activity and report the activity's result as a response to the requesting socket with guaranteed delivery.

at-least-once transaction: An ATP transaction in which the requested operation is performed at least once, and possibly several times.

ATP: See AppleTalk Transaction Protocol.

bridge: An intelligent link between two or more AppleTalk networks.

broadcast service: An ALAP service wherein a frame is sent to all nodes on an AppleTalk network.

datagram: A packet of data transmitted by DDP.

Datagram Delivery Protocol (DDP): An AppleTalk protocol that is an ALAP client, managing socket-to-socket delivery of datagrams over AppleTalk internets.

DDP: See Datagram Delivery Protocol.

entity name: An identifier for an entity, of the form object:type@zone.

exactly-once transaction: An ATP transaction in which the requested operation is performed only once.

frame check sequence: Part of an ALAP frame trailer used by the AppleTalk header to check for transmission errors.

frame header: Information at the beginning of a packet.

frame trailer: Information at the end of an ALAP frame.

internet: An interconnected group of AppleTalk networks.

internet address: The AppleTalk address and network number of a socket.

LAP protocol type: An identifier used to match particular kinds of packets with a particular protocol handler.

Name-Binding Protocol (NBP): An AppleTalk protocol that's a DDP client, used to convert entity names to their internet socket addresses.

name lookup: An NBP operation that allows clients to obtain the internet addresses of entities from their names.

names directory: The union of all name tables in an internet.

names information socket: The socket in a node used to implement NBP (always socket number 2).

names table: A list of each entity's name and internet address in a node.

NBP: See Name-Binding Protocol.

NBP tuple: An entity name and an internet address.

network number: An identifier for an AppleTalk network.

network-visible entity: A named socket client on an internet.

node: A device that's attached to and communicates via an AppleTalk network.

node ID: A number, dynamically assigned, that identifies a node.

protocol: A well-defined set of communications rules.

protocol handler: A software process in a node that recognizes different kinds of frames by their ALAP type and services them.

protocol handler table: A list of the protocol handlers for a node.

release timer: A timer for determining when an exactly-once (XO) response buffer can be released.

response BDS: A data structure used to pass response information to the ATP module.

retry count: The maximum number of retransmissions for an NBP or ATP packet.

retry interval: The time between retransmissions of a packet by NBP or ATP.

routing table: A table in a bridge that contains routing information.

Routing Table Maintenance Protocol (RTMP): An AppleTalk protocol that is used internally by AppleTalk to maintain tables for routing datagrams through an internet.

RTMP: See Routing Table Maintenance Protocol.

RTMP socket: The socket in a node used to implement RTMP.

RTMP stub: The RTMP code in a nonbridge node.

sequence number: A number from Ø to 7, assigned to an ATP response datagram to indicate its ordering within the response.

socket: A logical entity within the node of a network.

socket client: A software process in a node that owns a socket.

socket listener: The portion of a socket client that receives and services datagrams addressed to that socket.

socket number: An identifier for a socket.

socket table: A listing of all the socket listeners for each active socket in a node.

transaction: A request-response communication between two ATP clients (see transaction request, transaction response).

transaction ID: An identifier assigned to a transaction.

transaction request: The initial part of a transaction in which one socket client asks another to perform an operation and return a response.

transaction response: The concluding part of a transaction in which one socket client returns requested information or simply confirms that a requested operation was performed.

user bytes: Four bytes in an ATP header provided for use by ATP's clients.

write data structure (WDS): A data structure used to pass information to the ALAP or DDP modules.

zone: An arbitrary subset of AppleTalk networks in an internet.

MACINTOSH USER EDUCATION

The Vertical Retrace Manager: A Programmer's Guide /VRMGR/TASK

See Also: The Macintosh User Interface Guidelines
The Memory Manager: A Programmer's Guide
The File Manager: A Programmer's Guide
The Device Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Desk Manager: A Programmer's Guide
Inside Macintosh: A Road Map
Programming Macintosh Applications in Assembly Language

Modification History: First Draft (ROM 7) Bradley Hacker 6/15/84

ABSTRACT

This manual describes the Vertical Retrace Manager, the part of the Macintosh Operating System that schedules and performs recurrent tasks during vertical retrace interrupts. It describes how your application can install and remove its own recurrent tasks.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Vertical Retrace Manager, the part of the Macintosh Operating System that schedules and performs recurrent tasks during vertical retrace interrupts. It describes how your application can install and remove its own recurrent tasks. *** Eventually it will become part of the comprehensive Inside Macintosh manual. ***

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal. You should also be familiar with the following:

- the Macintosh Operating System's Memory Manager
- interrupts, as described in the Macintosh Operating System's Device Manager manual
- queues, as described in the Operating System Utilities manual *** not yet; for now, see the appendix of the File 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 Vertical Retrace Manager and what you can do with it. It then introduces the routines of the Vertical Retrace Manager and tells how they fit into the flow of your application. This is followed by detailed descriptions of the routines themselves.

Finally, there's a summary of the Vertical Retrace Manager, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE VERTICAL RETRACE MANAGER

The Macintosh video circuitry generates a vertical retrace interrupt (also known as the vertical blanking or VBL interrupt) 6Ø 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 Operating System uses this interrupt as a convenient time to perform the following sequence of recurrent tasks:

1. Increment the number of ticks since system startup (every interrupt). (You can get this number by calling the Toolbox Event Manager function TickCount.)

2. Check whether the stack and heap have collided (every interrupt).

3. Handle cursor movement (every interrupt).

4. Post a mouse event if the state of the mouse button changed from its previous state and then remained unchanged for four interrupts (every other interrupt).

5. Post a disk inserted event if a disk has been inserted (every 3Ø interrupts).

These tasks must execute at regular intervals based on the "heartbeat" of the Macintosh, and shouldn't be changed.

An application can add any number of its own tasks for the Vertical Retrace Manager to execute. Application tasks can perform any desired actions as long as memory is neither allocated nor released, and can be set to execute at any frequency (up to once per vertical retrace interrupt). For example, a task within an electronic-mail application might check every tenth of a second to see if it has received any messages.

(note)

> Application tasks longer than about one-sixtieth of a second will affect other interrupt-driven parts of the Macintosh, such as the mouse position.

Information describing each application task is contained in the vertical retrace queue. The vertical retrace queue is a standard Macintosh Operating System queue, as described in the Operating System Utilities manual *** doesn't yet exist; for now, see the File Manager manual's appendix ***. Each entry in the vertical retrace queue has the following structure:

```
TYPE VBLTask = RECORD
                 qLink:     QElemPtr; {next queue entry}
                 qType:     INTEGER;  {queue type}
                 vblAddr:   ProcPtr;  {task address}
                 vblCount:  INTEGER;  {task frequency}
                 vblPhase:  INTEGER   {task phase}
               END;
```

As in all Operating System queue entries, qLink points to the next entry in the queue, and qType indicates the queue type. QType should always be ORD(vType) in the vertical retrace queue.

VBLAddr contains the address of the task. VBLCount specifies the number of ticks between successive calls to the task. This value is decremented each sixtieth of a second until it reaches Ø, at which point the task is called. The task must then reset vblCount, or its entry will be removed from the queue after it has been executed. VBLPhase contains an integer (smaller than vblCount) used to modify vblCount when the task is first added to the queue. This ensures that two or more routines added to the queue at the same time with the same vblCount value will be out of phase with each other, and won't be called during the same interrupt.

Assembly-language note: The Vertical Retrace Manager sets bit 6 of the queue flags whenever a task is being executed; assembly-programmers can use the global constant inVBL to test this bit.

USING THE VERTICAL RETRACE MANAGER

This section discusses how the Vertical Retrace Manager routines fit into the general flow of an application program. The routines themselves are described in detail in the next section.

The Vertical Retrace Manager is automatically initialized each time the system is started up. To add an application task to the vertical retrace queue, call VInstall. When your application no longer wants a task to be executed, it can remove the task from the vertical retrace queue by calling VRemove. An application task shouldn't call VRemove to remove its entry from the queue--either the application should call VRemove, or the task should simply not reset the vblCount field of the queue entry.

An application task cannot call routines that cause memory to be allocated or released. This severely limits the actions of tasks, so you might prefer using the Desk Manager procedure SystemTask to perform periodic actions. Or, since the very first thing the Vertical Retrace Manager does during a vertical retrace interrupt is increment the tick count, your application could call the Toolbox Event Manager function TickCount repeatedly and perform periodic actions whenever a specific number of ticks have elapsed.

Assembly-language note: Application tasks may use registers DØ through D3 and AØ through A3, and must save and restore any additional registers used. They must exit with an RTS instruction.

If you'd like to manipulate the contents of the vertical retrace queue directly, you can get a pointer to the vertical retrace queue by calling GetVBLQHdr.

VERTICAL RETRACE MANAGER ROUTINES

This section describes the Vertical Retrace Manager routines. Each routine is presented in its Pascal form; where applicable, it's followed by a box containing information needed to use the routine from assembly language. For general information on using the Vertical Retrace Manager from assembly language, see the manual Programming Macintosh Applications in Assembly Language.

FUNCTION VInstall (vblTaskPtr: QElemPtr) : OSErr;

Trap macro	_VInstall
On entry	AØ: vblTaskPtr (pointer)
On exit	DØ: result code (integer)

VInstall adds the task described by vblTaskPtr to the vertical retrace queue. Your application must fill in all fields of the task except qLink. VInstall returns one of the result codes listed below.

Result codes	noErr	No error
	vTypErr	QType field isn't ORD(vType)

FUNCTION VRemove (vblTaskPtr: QElemPtr) : OSErr;

Trap macro	_VRemove
On entry	AØ: vblTaskPtr (pointer)
On exit	DØ: result code (integer)

VRemove removes the task described by vblTaskPtr from the vertical retrace queue. It returns one of the result codes listed below.

Result codes	noErr	No error
	vTypErr	QType field isn't ORD(vType)
	qErr	Task entry isn't in the queue

FUNCTION GetVBLQHdr : QHdrPtr; [Pascal only]

GetVBLQHdr returns a pointer to the vertical retrace queue.

Assembly-language note: To access the contents of the vertical retrace queue from assembly language, assembly-language programmers can use offsets from the address of the global variable vblQueue.

SUMMARY OF THE VERTICAL RETRACE MANAGER

Constants

```
CONST { Result codes }

      noErr   =  Ø; {no error}
      qErr    = -1; {task entry isn't in the queue}
      vTypErr = -2; {qType field isn't ORD(vType)}
```

Data Types

```
TYPE VBLTask = RECORD
                 qLink:     QElemPtr; {next queue entry}
                 qType:     INTEGER;  {queue type}
                 vblAddr:   ProcPtr;  {task address}
                 vblCount:  INTEGER;  {task frequency}
                 vblPhase:  INTEGER   {task phase}
               END;
```

Routines

```
FUNCTION VInstall (vblTaskPtr: QElemPtr) : OSErr;
FUNCTION VRemove  (vblTaskPtr: QElemPtr) : OSErr;
FUNCTION GetVBLQHdr : QHdrPtr;  [Pascal only]
```

Assembly-Language Information

Constants

```
inVBL        .EQU     6        ;set if Vertical Retrace Manager
                               ; is executing

; Result codes

qErr         .EQU    -1        ;task entry isn't in the queue
vTypErr      .EQU    -2        ;qType field isn't vType
```

Vertical Retrace Queue Entry

qLink	Pointer to next queue entry
qType	Queue type
vblAddr	Task address
vblCount	Task frequency
vblPhase	Task phase

Variables

Name	Size	Contents
vblQueue	4 bytes	Vertical retrace queue

GLOSSARY

vertical retrace interrupt: The interrupt that occurs 6Ø 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.

vertical retrace queue: A list of the application tasks to be executed during the vertical retrace interrupt.

The System Error Handler: A Programmer's Guide /ERROR/SYS

See Also: Inside Macintosh: A Road Map
Macintosh Memory Management: An Introduction
Programming Macintosh Applications in Assembly Language
QuickDraw: A Programmer's Guide
The Control Manager: A Programmer's Guide
The Menu Manager: A Programmer's Guide
The Dialog Manager: A Programmer's Guide
The Package Manager: A Programmer's Guide
The File Manager: A Programmer's Guide
The Segment Loader: A Programmer's Guide
The Memory Manager: A Programmer's Guide
The Device Manager: A Programmer's Guide
The Resource Manager: A Programmer's Guide

Modification History: First Draft Bradley Hacker 9/26/84

ABSTRACT

The System Error Handler is the part of the Macintosh Operating System that assumes control when a fatal error (such as running out of memory) occurs. This manual introduces you to the System Error Handler and describes how your application can recover from system errors.

TABLE OF CONTENTS

ABOUT THIS MANUAL

The System Error Handler is the part of the Macintosh Operating System that assumes control when a fatal error occurs. This manual introduces you to the System Error Handler and describes how your application can recover from system errors. *** Eventually this will become part of the comprehensive Inside Macintosh manual. ***

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal and the information in the following manuals:

- Inside Macintosh: A Road Map
- Macintosh Memory Management: An Introduction
- Programming Macintosh Applications in Assembly Language, if you're using assembly language

You'll also need to be somewhat familiar with most of the User Interface Toolbox and the rest of the Operating System.

ABOUT THE SYSTEM ERROR HANDLER

The System Error Handler assumes control when a fatal system error occurs. Its main function is to display an alert box with a diagnostic error message (called a system error alert) and provide a mechanism for the application to resume execution.

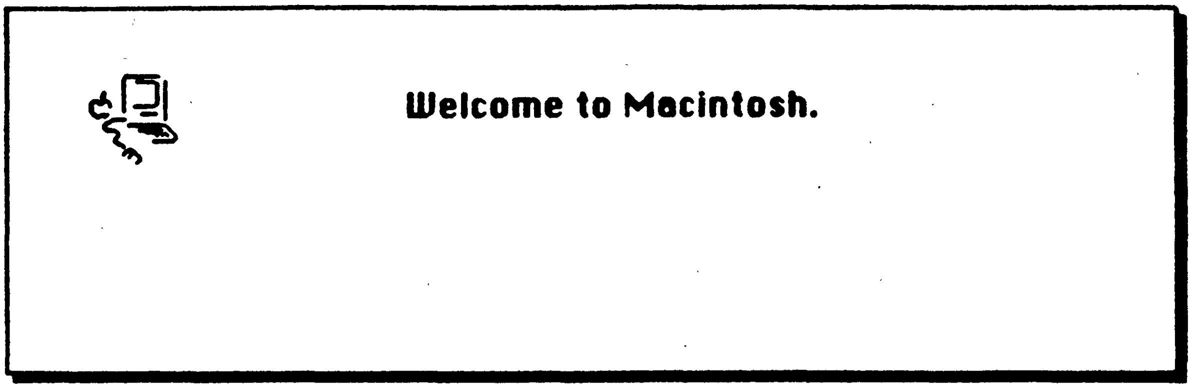
Because a system error usually indicates that a very low-level part of the system has failed, the System Error Handler performs its duties by using as little of the system as possible. It requires only the following:

- The trap dispatcher is operative.
- The Font Manager's InitFonts procedure has been called (it's called when the system starts up).
- Register A7 must point to a reasonable place in memory (for example, not to the main screen buffer).
- A few important system data structures aren't too badly damaged.

The System Error Handler doesn't require the Memory Manager to be operative.

The content of the alert box displayed is determined by a system error alert table, a resource stored in the system resource file. There are three different system error alert tables: a system startup alert table used when the system starts up, a user alert table used to inform the Macintosh user of system errors, and a programmer alert table used by programmers when debugging.

The system startup alerts include the "Welcome to Macintosh" box (Figure 1). It's displayed by the System Error Handler instead of the Dialog Manager because the System Error Handler needs very little of the system to operate.

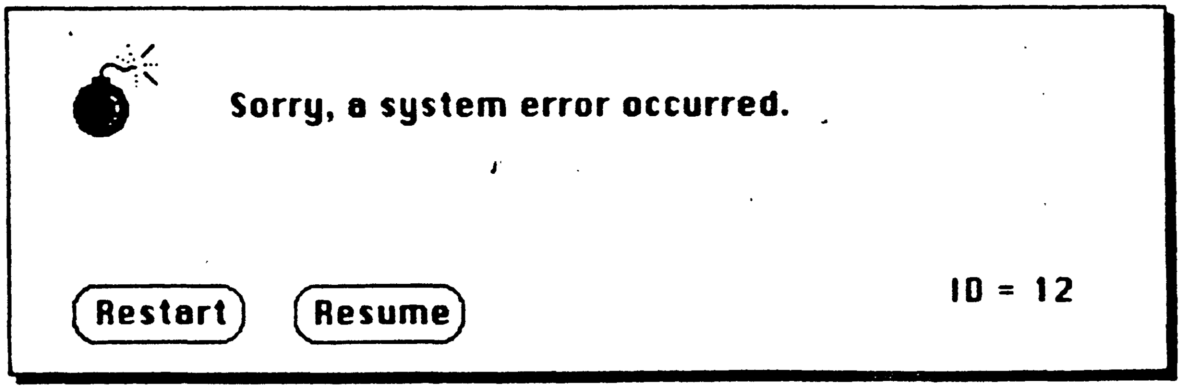


Welcome to Macintosh.

Figure 1. System Startup Alert

Only one of the system startup alerts actually interrupts execution: if the system can't find a disk with which to start up the system, a system startup alert containing an Eject and a Restart button will appear. The bad disk will be ejected if the user clicks the Eject button, and the Macintosh will attempt to restart if the user clicks the Restart button. The summary of this manual contains a complete list of the alert messages that can appear during system startup.

The user alerts (Figure 2) are used to notify the user of system errors in a friendly manner. The bottom right corner of a user alert contains a *system error ID* that identifies the error. Usually the message "Sorry, a system error occurred.", a Restart button, and a Resume button are also shown. If the Finder can't be found on a disk, the message "Can't load the finder" and a Restart button will be shown. The Macintosh will attempt to restart if the user clicks the Restart button, and the application will attempt to resume execution if the user clicks the Resume button.

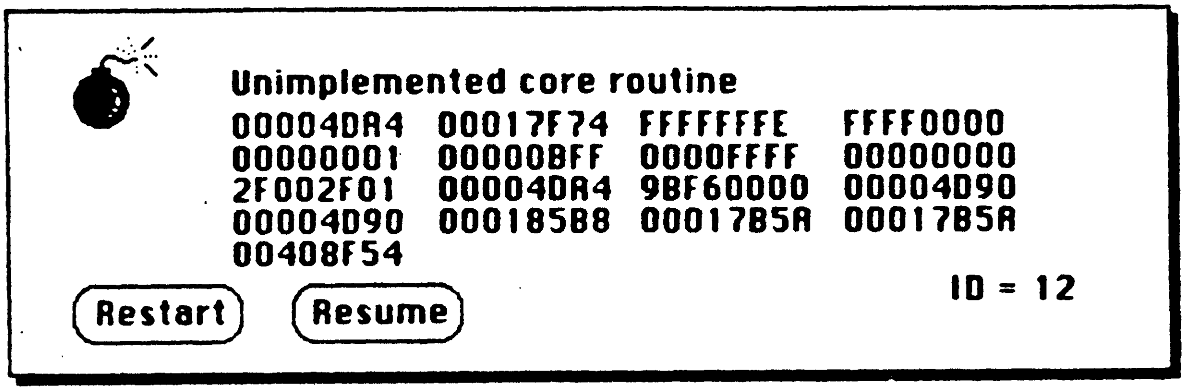


Sorry, a system error occurred.

Figure 2. User Alert

The "Please insert the disk:" alert displayed by the File Manager is also a user alert.

The programmer alerts (Figure 3) are used to provide programmers with information to diagnose the cause of system errors. They include all the alerts seen by users, but some of the alerts also display the contents of all registers. In addition to the Restart and Resume buttons, programmer alerts contain a Finder button, which if pressed launches the Finder. The summary contains a complete list of the programmer alert messages that can appear.



```
Unimplemented core routine
00004DA4  00017F74  FFFFFFFE  FFFF0000
00000001  00000BFF  0000FFFF  00000000
2F002F01  00004DA4  9BF60000  00004D90
00004D90  000185B8  00017B5A  00017B5A
00408F54
                                   ID = 12
```

Figure 3. Programmer Alert

Programmer alerts have been supplanted by user alerts (which provide the system error ID) and debuggers (which provides access to the contents of registers). Consequently, programmer alerts aren't normally part of the system.

RECOVERING FROM SYSTEM ERRORS

An application recovers from a system error by means of a resume procedure. You can pass a pointer to your resume procedure when you call the Dialog Manager procedure InitDialogs (if you don't have a resume procedure, you'll pass NIL). When the user clicks the Resume button in a system error alert, the System Error Handler attempts to restore the state of the system and then jumps to your resume procedure.

Assembly-language note: The System Error Handler actually restores the value of register A5 to what it was before the system error occurred, places the stack pointer at the bottom of the stack (throwing away the stack), and then jumps to your resume procedure.

SYSTEM ERROR HANDLER ALERT TABLES

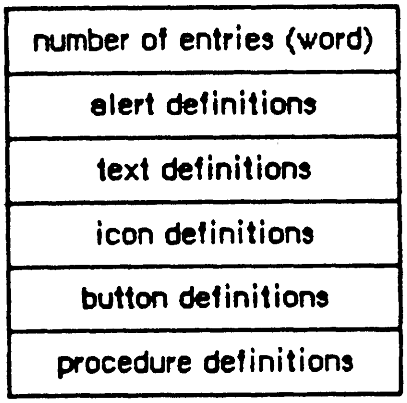
This section describes the data structures that define the alert boxes displayed by the System Error Handler. Most programmers won't need to know this background information; it pertains to the exact steps that the System Error Handler takes to generate a system error.

In the system resource file, the system error alerts have the following resource types and IDs:

Table	Resource type	Resource ID
System startup alert table	'DSAT'	Ø
Programmer alert table	'INIT'	1
User alert table	'INIT'	2

Assembly-language note: The global variable DSAlertTab contains a pointer to the current system error alert table. DSAlertTab points to the system startup alert table when the system is starting up. After that, DSAlertTab is changed to point to one of the other system error alert tables.

The format of a system error alert table is shown in Figure 4. It consists of a word indicating the length of the table, followed by alert, text, icon, button, and procedure definitions, all of which are explained below.



number of entries (word)

alert definitions

text definitions

icon definitions

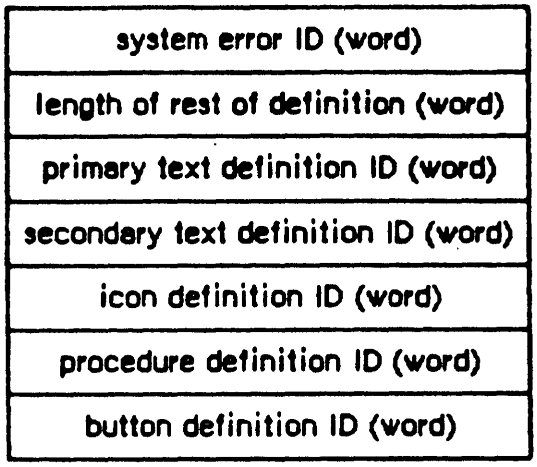
button definitions

procedure definitions

Figure 4. System Error Alert Table

The definitions within the alert table needn't be in the order shown, and definitions of one type needn't all be grouped together as shown. The first two words in every definition are used for the same purpose: the first contains an ID identifying the definition, and the second specifies the length of the definition.

An alert definition specifies the appearance and operation of the alert box that will be drawn when a particular system error occurs (Figure 5). The first word in an alert definition specifies which system error the alert pertains to.

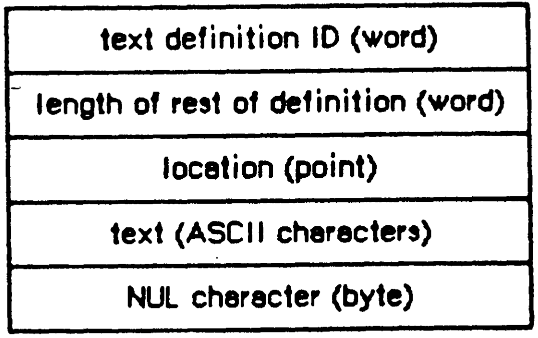


system error ID (word)
length of rest of definition (word)
primary text definition ID (word)
secondary text definition ID (word)
icon definition ID (word)
procedure definition ID (word)
button definition ID (word)

Figure 5. Alert Definition

The first alert definition in a system error alert table applies to all system errors that don't have their own alert definition.

A text definition specifies the text that will be drawn in a particular system error alert (Figure 6). The first word in the definition indicates the system error ID to which the text pertains. Note that each alert definition refers to two text definitions. The location (in global coordinates) where the text should be drawn is given as a point. The actual characters that comprise the text are suffixed by one NUL character.



text definition ID (word)
length of rest of definition (word)
location (point)
text (ASCII characters)
NUL character (byte)

Figure 6. Text Definition

An icon definition specifies the icon that will be drawn in a particular system error alert (Figure 7). The first word in the definition indicates the system error ID to which the icon pertains. The location (in global coordinates) where the icon should be drawn is given as a rectangle. The 128 bytes that comprise the icon complete the definition.

icon definition ID (word)
length of rest of definition (word)
location (rectangle)
icon data (128 bytes)

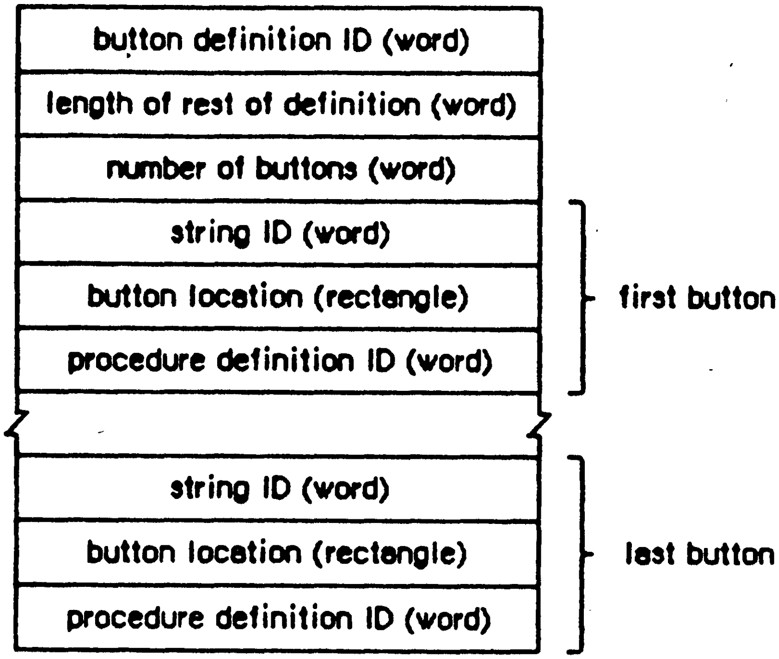
Figure 7. Icon Definition

A procedure definition specifies the procedure that will be executed whenever a particular system error alert box is drawn (Figure 8). The first word in the definition indicates the system error ID to which the procedure pertains. Most of a procedure definition is simply the code comprising the procedure.

procedure definition ID (word)
length of rest of definition (word)
procedure code

Figure 8. Procedure Definition

A button definition specifies the button(s) that will be drawn in a particular system error alert (Figure 9). The first word in the definition indicates the system error ID to which the button(s) pertain. The next word indicates the number of buttons that will be drawn. The rest of the button definition is composed of eight-word groups, each of which specifies the text, location, and operation of a button.



button definition ID (word)

length of rest of definition (word)

number of buttons (word)

string ID (word)

button location (rectangle)

procedure definition ID (word)

string ID (word)

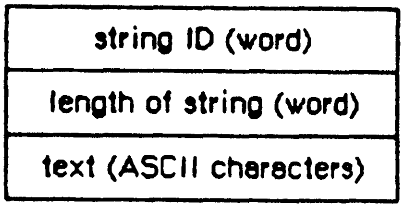
button location (rectangle)

procedure definition ID (word)

Figure 9. Button Definition

The first word contains a string ID (explained below), specifying the text that will be drawn inside the button. The location (in global coordinates) where the button should be drawn is given as a rectangle. The last word contains a procedure definition ID, identifying the code to be executed when the button is clicked.

The text that will be drawn inside each button is specified by the data structure shown in Figure 1Ø. The first word contains a string ID number identifying the string and the second indicates the length of the string in bytes. The actual characters of the string follow.



string ID (word)

length of string (word)

text (ASCII characters)

Figure 1Ø. Strings Drawn in Buttons

(warning)

> If a resume procedure was specified by a call to the Dialog Manager's InitDialogs procedure, the System Error Handler automatically adds 1 to the button definition ID in the alert definition. For this reason, button definitions must always occur in pairs, and the button definition IDs must differ by 1.

(note)
Every definition within a system error alert table must be word-aligned and have a unique ID.

SYSTEM ERROR HANDLER ROUTINE

The System Error Handler has only one routine, SysError, described in this section. Most application programs won't have any reason to call it. The system itself calls SysError whenever a system error occurs, and most applications need only be concerned with recovering from the error and resuming execution.

PROCEDURE SysError (errorCode: INTEGER);

Trap macro	_SysError
On entry	DØ: errorCode (integer)
On exit	all registers changed

SysError generates a system error with the ID specified by the errorCode parameter.

It does the following precise steps:

1. Saves all registers and the stack pointer.

Assembly-language note: Actually, the following instructions are executed:

```
        MOVEM.L  AØ-A7/DØ-D7,$7FC8Ø
        MOVE.L   (SP),$7FCCØ
```

Note that $7FC8Ø is address $1FC8Ø on a 128K Macintosh.

2. Stores the system error ID in a global variable (named DSErrCode).

3. Checks to see whether there's a system error alert table in memory (by testing whether the global variable DSAlertTab is Ø). If there's no system error alert table, it draws the unhappy

Macintosh icon with a 16-bit number and a 32-bit number. The 16-bit number is always $ØF for system errors, and the 32-bit number is the system error ID.

4. Allocates memory for QuickDraw globals on the stack, initializes QuickDraw, and initializes a grafPort in which the alert box will be drawn.

5. Checks the system error ID. If the system error ID is negative, the alert box isn't redrawn (this is used for system startup alerts, which can display a sequence of consecutive messages in the same box). If the system error ID doesn't correspond to an entry in the system error alert table, the system error alert will display the message "Sorry, a system error has occurred.".

6. Draws an alert box (in the rectangle specified by the global variable DSAlertRect).

7. If the text definition IDs aren't Ø, it draws both strings.

8. If the icon definition ID isn't Ø, it draws the icon.

9. If the button definition ID ir Ø, it returns control to the procedure that called it (this is used during the disk-switch alert to return control to the File Manager after the "Please insert the disk:" message has been displayed).

1Ø. If there's a resume procedure, it increments the button definition ID by 1.

11. Draws the buttons.

12. Hit-tests the buttons and calls the corresponding procedure code when a button is pressed. If there's no procedure code, it returns to the procedure that called it (normally this shouldn't happen).

SUMMARY OF THE SYSTEM ERROR HANDLER

Programmer and User Alerts

ID	Explanation
1	Bus error: Never happens on a Macintosh
2	Address error: Word or long-word reference made to an odd address
3	Illegal instruction: The 68ØØØ received an instruction it didn't recognize.
4	Zero divide: Signed Divide (DIVS) or Unsigned Divide (DIVU) instruction with a divisor of Ø was executed.
5	Check exception: Check Register Against Bounds (CHK) instruction was executed and failed.
6	TrapV exception: Trap On Overflow (TRAPV) instruction was executed and failed.
7	Privilege violation: Macintosh always runs in privilege mode; perhaps an erroneous RTE instruction was executed.
8	Trace exception: The trace bit in the status register is set.
9	Line 1Ø1Ø exception: The 1Ø1Ø trap dispatcher is broken.
1Ø	Line 1111 exception: Usually a breakpoint
11	Miscellaneous exception: All other 68ØØØ exceptions
12	Unimplemented core routine: An unimplemented trap number was encountered.
13	Spurious interrupt: The interrupt vector table entry for a particular level of interrupt is NIL; usually occurs with level 4, 5, 6, or 7 interrupts.
14	I/O system error: The File Manager is attempting to dequeue an element from the I/O request queue that has a bad queue type field; perhaps the queue element is unlocked. Or, the dCtlQHead field was NIL during a Fetch or Stash call. Or, a needed device control entry has been purged.
15	Segment Loader error: A GetResource call to read a segment into memory failed.
16	Floating point error: The halt bit in the floating-point environment word was set.
17-24	Can't load package: A GetResource call to read a package into memory failed.
25	Out of memory!
26	Segment Loader error: A GetResource call to read segment Ø into memory failed; usually indicates a nonexecutable file.
27	File map trashed: A logical block number was found that is greater than the number of the last logical block on the volume or less than the logical block number of the first allocation block on the volume.
28	Stack overflow error: The stack and heap have collided.
3Ø	"Please insert the disk:" File Manager alert
32-53	Memory Manager error
41	The file named "Finder" can't be found on the disk
1ØØ	Can't mount system startup volume. The system couldn't read the system resource file into memory.

32767 Sorry, a system error has occurred: Undifferentiated error

System Startup Alerts

ID	Explanation
-12	RAM-based Operating System installed
-11	Disassembler installed
-1Ø	MacsBug installed
4Ø	"Welcome to Macintosh" box
42	Can't mount system startup volume: An attempt to mount the volume in the internal drive failed, or the system couldn't read the system resource file into memory
43	"Warning--this startup disk is not usable"

Routines

```
PROCEDURE SysError (errorCode: INTEGER);
```

Assembly-Language Information

Constants

```
; System error IDs

dsBusErr        .EQU      1     ;Bus Error
dsAddressErr    .EQU      2     ;Address Error
dsIllInstErr    .EQU      3     ;Illegal Instruction
dsZeroDivErr    .EQU      4     ;Zero Divide
dsChkErr        .EQU      5     ;Check Exception
dsOvflowErr     .EQU      6     ;TrapV Exception
dsPrivErr       .EQU      7     ;Privilege Violation
dsTraceErr      .EQU      8     ;Trace Exception
dsLineAErr      .EQU      9     ;Line 1Ø1Ø Exception
dsLineFErr      .EQU     1Ø     ;Line 1111 Exception
dsMiscErr       .EQU     11     ;Miscellaneous Exception
dsCoreErr       .EQU     12     ;Unimplemented Core Routine
dsIrqErr        .EQU     13     ;Spurious Interrupt
dsIOCoreErr     .EQU     14     ;I/O System Error
dsLoadErr       .EQU     15     ;Segment Loader Error
dsFPErr         .EQU     16     ;Floating Point Error
dsNoPackErr     .EQU     17     ;Can't load package Ø
dsNoPk1         .EQU     18     ;Can't load package 1
dsNoPk2         .EQU     19     ;Can't load package 2
dsNoPk3         .EQU     2Ø     ;Can't load package 3
dsNoPk4         .EQU     21     ;Can't load package 4
dsNoPk5         .EQU     22     ;Can't load package 5
dsNoPk6         .EQU     23     ;Can't load package 6
dsNoPk7         .EQU     24     ;Can't load package 7
dsMemFullErr    .EQU     25     ;Out of memory!
```

```
dsBadLaunch     .EQU      26    ;Segment Loader Error
dsFSErr         .EQU      27    ;File Map trashed
dsStkNHeap      .EQU      28    ;Stack overflow error
dsReinsert      .EQU      3Ø    ;Please insert the disk
dsNotThe1       .EQU      31    ;This is not the correct disk
memTrbBase      .EQU      32    ;Memory Manager failed
dsSysErr        .EQU   32767    ;System Error
```

Routines

_SysError

On entry DØ: errorCode (integer)

On exit all registers changed

Variables

Name	Size	Contents
DSErrCode	2 bytes	Current system error ID
DSAlertTab	4 bytes	Address of system error alert table in use
DSAlertRect	8 bytes	Location of system error alert

GLOSSARY

resume procedure: A procedure within an application that allows the application to recover from system errors.

system error alert: An alert box displayed by the System Error Handler.

system error alert table: A resource that determines the appearance and function of system error alerts.

system error ID: An ID number that appears in a system error alert to identify the error.

MACINTOSH USER EDUCATION

The Operating System Utilities: A Programmer's Guide /OSUTIL/UTIL

See Also: Inside Macintosh: A Road Map
Macintosh Memory Management: An Introduction
Programming Macintosh Applications in Assembly Language
Macintosh Packages: A Programmer's Guide
The Structure of a Macintosh Application
The Font Manager: A Programmer's Guide
The Device Manager: A Programmer's Guide
The Serial Drivers: A Programmer's Guide
Index to Technical Documentation

Modification History:			
	First Draft	Brent Davis & Caroline Rose	8/7/84
	Second Draft	Caroline Rose	1/11/85

ABSTRACT

This manual describes the Operating System Utilities, a set of routines and data types in the Operating System that perform generally useful operations such as manipulating pointers and handles, comparing strings, and reading the date and time.

Summary of significant changes and additons since last draft:

- Corrections have been made to the descriptions of EqualString (page 12), UprString (page 13), and SysBeep (page 23).

- The description of Date2Secs has been expanded (page 15).

- An appendix has been added that lists all result codes, in numerical order (31).

- The appendix containing the system traps has been expanded to include a numerically ordered list (page 34).

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Operating System Utilities, a set of routines and data types in the Operating System that perform generally useful operations such as manipulating pointers and handles, comparing strings, and reading the date and time. *** Eventually it will become part of the comprehensive Inside Macintosh manual. ***

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal and the information in the following manuals:

- Inside Macintosh: A Road Map

- Macintosh Memory Management: An Introduction

- Programming Macintosh Applications in Assembly Language, if you're using assembly language

Depending on which Operating System Utilities you're interested in using, you may also need to be familiar with other parts of the Toolbox or Operating System; where that's necessary, you're referred to the appropriate manuals.

PARAMETER RAM

Various settings, such as those specified by the user by means of the Control Panel desk accessory, need to be preserved when the Macintosh is off so they will still be present at the next system startup. This information is kept in parameter RAM, 2Ø bytes that are stored in the clock chip together with the current settings for the date and time. The clock chip is powered by a battery when the system is off, thereby preserving all the settings stored in it.

You may find it necessary to read the values in parameter RAM or even change them (for example, if you create a desk accessory like the Control Panel). Since the clock chip itself is difficult to access, its contents are copied into low memory at system startup. You read and change parameter RAM through this low-memory copy.

(note)

> Certain values from parameter RAM are used so frequently that special routines have been designed to return them (for example, the Toolbox Event Manager function GetDblTime). These routines are discussed in other manuals where appropriate.

> Assembly-language note: The low-memory copy of parameter RAM begins at the address SysParam; the various portions of the copy can be accessed through individual global variables, listed in

the summary at the end of this manual. Some of these are copied into other global variables at system startup for even easier access: for example, the auto-key threshold and rate, which are contained in the variable SPKbd in the copy of parameter RAM, are copied into the variables KeyThresh and KeyRepThresh. Each such variable is discussed in its appropriate manual.

The date and time is also copied at system startup from the clock chip into its own low-memory location. It's stored as a number of seconds since midnight, January 1, 1904, and is updated every second. The maximum value, $FFFFFFFF, corresponds to 6:28:15 AM, February 6, 2040; after that, it wraps around to midnight, January 1, 1904.

Assembly-language note: The low-memory location containing the date and time is the global variable Time.

The structure of parameter RAM is represented by the following data type:

```
TYPE SysParmType =
       RECORD
         valid:     LONGINT;  {validity status}
         portA:     INTEGER;  {modem port configuration}
         portB:     INTEGER;  {printer port configuration}
         alarm:     LONGINT;  {alarm setting}
         font:      INTEGER;  {default application font number minus 1}
         kbdPrint:  INTEGER;  {auto-key threshold and rate, printer }
                              { connection}
         volClik:   INTEGER;  {speaker volume, double-click and caret- }
                              { blink times}
         misc:      INTEGER   {mouse scaling, system startup disk, menu }
                              { blink}
       END;

     SysPPtr = ^SysParmType;
```

Only the high-order byte of the valid field is used (Figure 1). It contains the validity status of the clock chip: whenever you successfully write to the clock chip, $A8 is stored in this byte.

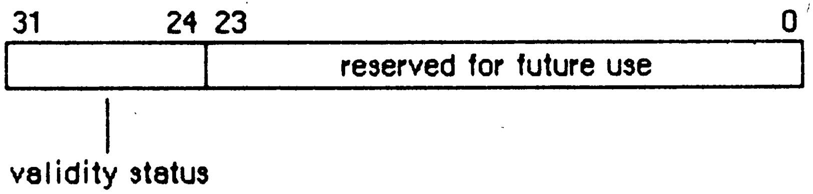


Figure 1. The Valid Field

The validity status is examined when the clock chip is read at system startup. It won't be $A8 if a hardware problem prevented the values from being written; in this case, the low-memory copy of parameter RAM is set to the default values shown in the table below, and these values are then written to the clock chip itself. (The meanings of the parameters are explained in the descriptions of the various fields, following the table.)

Parameter	Default value
Validity status	$A8
Modem port configuration	96ØØ baud, 8 data bits, 2 stop bits, no parity
Printer port configuration	Same as for modem port
Alarm setting	Ø (midnight, January 1, 19Ø4)
Default application font - 1	2 (Geneva)
Auto-key threshold	6 (24 ticks)
Auto-key rate	3 (6 ticks)
Printer connection	Ø (printer port)
Speaker volume	3 (medium)
Double-click time	8 (32 ticks)
Caret-blink time	8 (32 ticks)
Mouse scaling	1 (on)
Preferred system startup disk	Ø (internal drive)
Menu blink	3

(warning)

> Your program must not use bits indicated as "reserved for future use" in parameter RAM, since future Macintosh software features will use them.

The portA and portB fields contain the baud rates, data bits, stop bits, and parity for the device drivers using the modem port ("port A") and printer port ("port B"). An explanation of these terms and the exact format of the information are given in the Serial Drivers manual.

The alarm field contains the alarm setting in seconds since midnight, January 1, 19Ø4.

The font field contains 1 less than the number of the default application font. A list of font numbers can be found in the Font Manager manual.

Bit Ø of the kbdPrint field (Figure 2) designates whether the printer (if any) is connected to the printer port (Ø) or the modem port (1). Bits 8 through 11 of this field contain the auto-key rate, the rate of the repeat when a character key is held down; this value is stored in two-tick units (where one tick is a sixtieth of a second). Bits 12 through 15 contain the auto-key threshold, the length of time the key must be held down before it begins to repeat; it's stored in four-tick units.

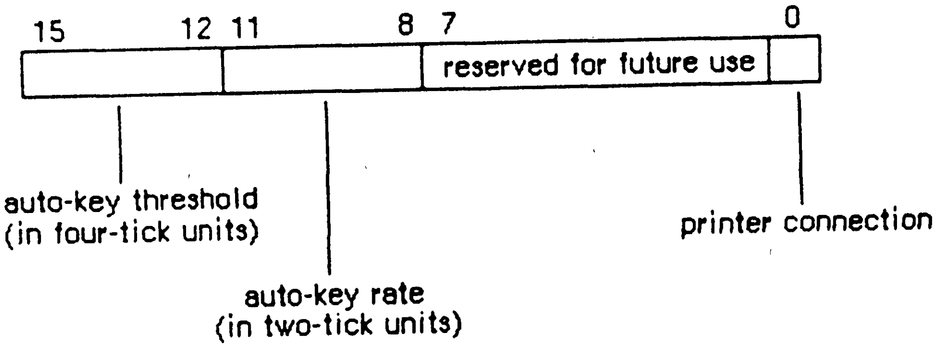


Figure 2. The KbdPrint Field

Bits Ø through 3 of the volClik field (Figure 3) contain the caret-blink time, and bits 4 through 7 contain the double-click time; both values are stored in four-tick units. The caret-blink time is the interval between blinks of the caret that marks an insertion point. The double-click time is the greatest interval between a mouse-up and mouse-down event that would qualify two mouse clicks as a double-click. Bits 8 through 1Ø of the volClick field contain the speaker volume, which has eight settings from silent (Ø) to loud (7).

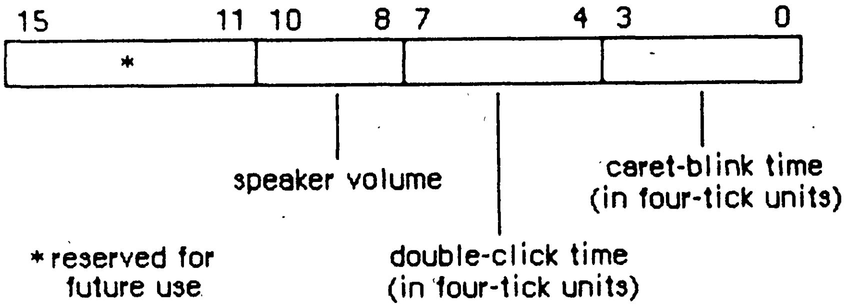


Figure 3. The VolClik Field

Bits 2 and 3 of the misc field (Figure 4) contain a value from Ø to 3 designating how many times a menu item will blink when it's chosen. Bit 4 of this field indicates whether the preferred disk to use to start up the system is in the internal (Ø) or the external (1) drive; if there's any problem using the disk in the specified drive, the other drive will be used.

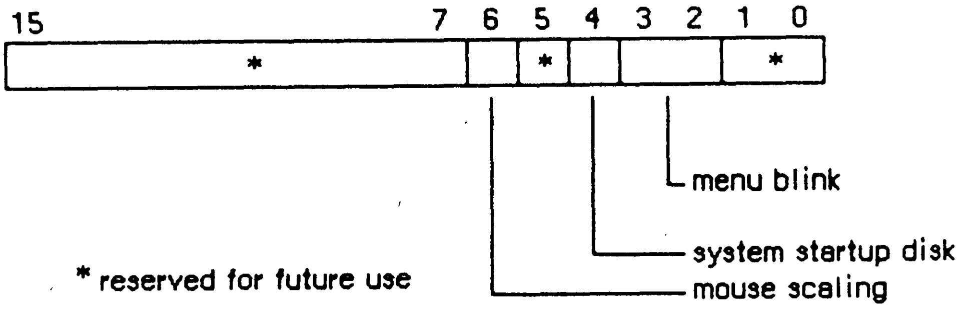


Figure 4. The Misc Field

Finally, bit 6 of the misc field designates whether mouse scaling is on (1) or off (Ø). If mouse scaling is on, the system looks every sixtieth of a second at whether the mouse has moved; if in that time the sum of the mouse's horizontal and vertical changes in position is greater than the mouse-scaling threshold (normally six pixels), then the cursor will move twice as far horizontally and vertically as it would if mouse scaling were off.

Assembly-language note: The mouse-scaling threshold is contained in the global variable CrsrThresh.

OPERATING SYSTEM QUEUES

Some of the information used by the Operating System is stored in data structures called queues. A queue is a list of identically structured entries linked together by pointers. Queues are used to keep track of vertical retrace tasks, I/O requests, events, mounted volumes, and disk drives (or other block-formatted devices).

A standard Operating System queue has a header with the following structure:

```
TYPE QHdr    = RECORD
                 qFlags: INTEGER;  {queue flags}
                 qHead:  QElemPtr; {first queue entry}
                 qTail:  QElemPtr  {last queue entry}
               END;

          QHdrPtr = ^QHdr;
```

QFlags contains information (usually flags) that's different for each queue type. QHead points to the first entry in the queue, and qTail points to the last entry in the queue. The entries within each type of queue are different; the Operating System uses the following variant record to access them:

```
TYPE QTypes   = (dummyType,
                 vType,      {vertical retrace queue type}
                 ioQType,    {file I/O or driver I/O queue type}
                 drvQType,   {drive queue type}
                 evType,     {event queue type}
                 fsQType);   {volume-control-block queue type}

     QElem    = RECORD
                  CASE QTypes OF
                    vType:    (vblQElem: VBLTask);
                    ioQType:  (ioQElem:  ParamBlockRec);
                    drvQType: (drvQElem: DrvQEl);
                    evType:   (evQElem:  EvQEl);
                    fsQType:  (vcbQElem: VCB)
                END;

     QElemPtr = ^QElem;
```

The exact structure of the entries in each type of Operating System queue is described in the manual that discusses that queue in detail; for more information, look up the corresponding data type in the index *** currently the manual _Index to Technical Documentation_ ***. All entries in queues, regardless of the queue type, begin with a pointer to the next queue entry and an integer designating the queue type (for example, ORD(evType) for the event queue).

Assembly-language note: The queue types are available to assembly-language programmers as global constants.

GENERAL OPERATING SYSTEM DATA TYPES

This section describes two data types of interest to users of the Operating System.

There are several places in the Operating System where you specify a four-character sequence for something, such as for file types and application signatures (as described in _The Structure of a Macintosh Application_). The Pascal data type for such sequences is

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

Another data type that's used frequently in the Operating System is

```
TYPE OSErr = INTEGER;
```

This is the data type for a result code, which many Operating System routines (including those described in this manual) return in addition to their normal results. A result code is an integer indicating whether the routine completed its task successfully or was prevented by some error condition (or other special condition, such as reaching the end of a file). In the normal case that no error is detected, the result code is

```
CONST noErr = Ø;  {no error}
```

A nonzero result code (usually negative) signals an error. A list of all result codes is provided in Appendix A.

OPERATING SYSTEM UTILITY ROUTINES

Pointer and Handle Manipulation

*** The notation "[No trap macro]" (formerly "[Pascal only]") has been changed to "[Not in ROM]" ***

These functions would be easy to duplicate with Memory Manager calls; they're included in the Operating System Utilities as a convenience because the operations they perform are so common.

```
FUNCTION HandToHand (VAR theHndl: Handle) : OSErr;
```

Trap macro	_HandToHand
On entry	AØ: theHndl (handle)
On exit	AØ: theHndl (handle) DØ: result code (word)

HandToHand copies the information to which theHndl is a handle and returns a new handle to the copy in theHndl. Since HandToHand replaces the input parameter with a new handle, you should retain the original value of the input parameter somewhere else, or you won't be able to access it. For example:

```
VAR x,y: Handle;
    err: OSErr;

y := x;
err := HandToHand(y)
```

The original handle remains in x while y becomes a different handle to identical data.

Result codes		
	noErr	No error
	memFullErr	Not enough room in heap
	nilHandleErr	NIL master pointer
	memWZErr	Attempt to operate on a free block

```
FUNCTION PtrToHand (srcPtr: Ptr; VAR dstHndl: Handle; size: LONGINT) :
        OSErr;
```

Trap macro	_PtrToHand	
On entry	AØ:	srcPtr (pointer)
	DØ:	size (long word)
On exit	AØ:	dstHndl (handle)
	DØ:	result code (word)

PtrToHand returns in dstHndl a newly created handle to a copy of the number of bytes specified by the size parameter, beginning at the location specified by srcPtr.

Result codes		
	noErr	No error
	memFullErr	Not enough room in heap

```
FUNCTION PtrToXHand (srcPtr: Ptr; dstHndl: Handle; size: LONGINT) :
        OSErr;
```

Trap macro	_PtrToXHand	
On entry	AØ:	srcPtr (pointer)
	A1:	dstHndl (handle)
	DØ:	size (long word)
On exit	A1:	dstHndl (handle)
	DØ:	result code (word)

PtrToXHand takes the existing handle specified by dstHndl and makes it a handle to a copy of the number of bytes specified by the size parameter, beginning at the location specified by srcPtr.

Result codes	noErr	No error
	memFullErr	Not enough room in heap
	nilHandleErr	NIL master pointer
	memWZErr	Attempt to operate on a free block

FUNCTION HandAndHand (aHndl,bHndl: Handle) : OSErr;

Trap macro	_HandAndHand	
On entry	AØ:	aHndl (handle)
	A1:	bHndl (handle)
On exit	A1:	bHndl (handle)
	DØ:	result code (word)

HandAndHand concatenates the information to which aHndl is a handle onto the end of the information to which bHndl is a handle.

Result codes	noErr	No error
	memFullErr	Not enough room in heap
	nilHandleErr	NIL master pointer
	memWZErr	Attempt to operate on a free block

FUNCTION PtrAndHand (pntr: Ptr; hndl: Handle; size: LONGINT) : OSErr;

Trap macro	_PtrAndHand	
On entry	AØ:	pntr (pointer)
	A1:	hndl (handle)
	DØ:	size (long word)
On exit	A1:	hndl (handle)
	DØ:	result code (word)

PtrAndHand takes the number of bytes specified by the size parameter, beginning at the location specified by pntr, and concatenates them onto the end of the information to which hndl is a handle.

Result codes		
	noErr	No error
	memFullErr	Not enough room in heap
	nilHandleErr	NIL master pointer
	memWZErr	Attempt to operate on a free block

String Comparison

Assembly-language note: The trap macros for these utility routines have optional arguments corresponding to the Pascal flags associated with the routines. When present, such an argument sets a certain bit of the routine trap word; this is equivalent to setting the corresponding Pascal flag to either TRUE or FALSE, depending on the flag. The trap macros for these routines are listed below, together with all the possible permutations of arguments. Whichever permutation you use, you must type it exactly as shown.

```
FUNCTION EqualString (aStr,bStr: Str255; caseSens,diacSens: BOOLEAN) :
         BOOLEAN;
```

Trap macro	_CmpString		
	_CmpString	,MARKS	(sets bit 9, for diacSens=FALSE)
	_CmpString	,CASE	(sets bit 1Ø, for caseSens=TRUE)
	_CmpString	,MARKS,CASE	(sets bits 9 and 1Ø)

On entry	AØ:	pointer to first character of first string
	A1:	pointer to first character of second string
	DØ:	high-order word: length of first string low-order word: length of second string
On exit	DØ:	Ø if strings equal, 1 if strings not equal (long word)

EqualString compares the two given strings for equality on the basis of their ASCII values. If caseSens is TRUE, uppercase characters are distinguished from the corresponding lowercase characters. If diacSens is FALSE, diacritical marks are ignored during the comparison. The function returns TRUE if the strings are equal.

(note)

See also the International Utilities Package function IUEqualString, as described in the Macintosh Packages manual.

```
PROCEDURE UprString (VAR theString: Str255; diacSens: BOOLEAN);
```

Trap macro	_UprString _UprString ,MARKS	 (sets bit 9, for diacSens=FALSE)
On entry	AØ: DØ:	pointer to first character of string length of string (word)
On exit	AØ:	pointer to first character of string

UprString converts any lowercase letters in the given string to uppercase, returning the converted string in theString. In addition, diacritical marks are stripped from the string if diacSens is FALSE.

Date and Time Operations

The following utilities are for reading and setting the date and time stored in the clock chip. Reading the date and time is a fairly common operation; setting it is somewhat rarer, but could be necessary for implementing a desk accessory like the Control Panel.

The date and time is stored as an unsigned number of seconds since midnight, January 1, 19Ø4; you can use a utility routine to convert this to a date/time record. Date/time records are defined as follows:

```
TYPE DateTimeRec =
       RECORD
         year:      INTEGER;  {19Ø4 to 2Ø4Ø}
         month:     INTEGER;  {1 to 12 for January to December}
         day:       INTEGER;  {1 to 31}
         hour:      INTEGER;  {Ø to 23}
         minute:    INTEGER;  {Ø to 59}
         second:    INTEGER;  {Ø to 59}
         dayOfWeek: INTEGER   {1 to 7 for Sunday to Saturday}
       END;
```

FUNCTION ReadDateTime (VAR secs: LONGINT) : OSErr;

Trap macro	_ReadDateTime
On entry	A0: pointer to long word secs
On exit	A0: pointer to long word secs D0: result code (word)

ReadDateTime copies the date and time stored in the clock chip to a low-memory location and returns it in the secs parameter. This routine is called at system startup; you'll probably never need to call it yourself. Instead you'll call GetDateTime (see below).

> Assembly-language note: The low-memory location to which ReadDateTime copies the date and time is the global variable Time.

Result codes	noErr	No error
	clkRdErr	Unable to read clock

PROCEDURE GetDateTime (VAR secs: LONGINT); [Not in ROM]

GetDateTime returns in the secs parameter the contents of the low-memory location in which the date and time is stored; if the date and time is properly set, secs will contain the number of seconds between midnight, January 1, 1904 and the time that the function was called.

(note)

> If your application disables interrupts for longer than a second, the number of seconds returned will not be exact.

> Assembly-language note: Assembly-language programmers can just access the global variable Time.

If you wish, you can convert the value returned by GetDateTime to a date/time record by calling the Secs2Date procedure.

(note)

Passing the value returned by GetDateTime to the International Utilities Package procedure IUDateString or IUTimeString will yield a string representing the corresponding date or time of day, respectively.

FUNCTION SetDateTime (secs: LONGINT) : OSErr;

Trap macro	_SetDateTime
On entry	DØ: secs (long word)
On exit	DØ: result code (word)

SetDateTime takes a number of seconds since midnight, January 1, 19Ø4 as specified by the secs parameter and writes it to the clock chip as the current date and time. It then attempts to read the value just written and verify it by comparing it to the secs parameter.

Assembly-language note: SetDateTime updates the global variable Time to the value of the secs parameter.

Result codes	noErr	No error
	clkWrErr	Time written did not verify
	clkRdErr	Unable to read clock

PROCEDURE Date2Secs (date: DateTimeRec; VAR secs: LONGINT);

Trap macro	_Date2Secs
On entry	AØ: pointer to date/time record
On exit	DØ: secs (long word)

Date2Secs takes the given date/time record, converts it to the corresponding number of seconds elapsed since midnight, January 1, 19Ø4, and returns the result in the secs parameter. The dayOfWeek field of the date/time record is ignored. The values passed in the year and month fields should be within their allowable ranges, or unpredictable results may occur. The remaining four fields of the date/time record may contain any value. For example, September 35 will be interpreted as October 4, and you could specify the 3ØØth day of the year as January 3ØØ.

PROCEDURE Secs2Date (secs: LONGINT; VAR date: DateTimeRec);

Trap macro	_Secs2Date
On entry	DØ: secs (long word)
On exit	AØ: pointer to date/time record

Secs2Date takes a number of seconds elapsed since midnight, January 1, 19Ø4 as specified by the secs parameter, converts it to the corresponding date and time, and returns the corresponding date/time record in the date parameter.

PROCEDURE GetTime (VAR date: DateTimeRec); [Not in ROM]

GetTime takes the number of seconds elapsed since midnight, January 1, 19Ø4 (obtained by calling GetDateTime), converts that value into a date and time (by calling Secs2Date), and returns the result in the date parameter.

Assembly-language note: From assembly language, you can pass the value of the global variable Time to Secs2Date.

PROCEDURE SetTime (date: DateTimeRec); [Not in ROM]

SetTime takes the date and time specified by the date parameter, converts it into the corresponding number of seconds elapsed since midnight, January 1, 19Ø4 (by calling Date2Secs), and then writes that value to the clock chip as the current date and time (by calling SetDateTime).

Assembly-language note: From assembly language, you can just call Date2Secs and SetDateTime directly.

Parameter RAM Operations

The following three utilities are used for reading from and writing to parameter RAM. Figure 5 illustrates the function of these three utilities; further details are given below and earlier in the "Parameter RAM" section.

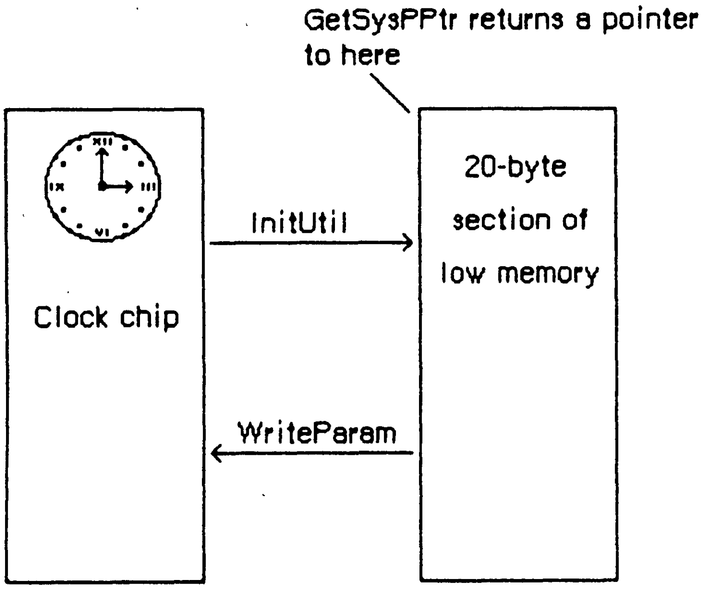

GetSysPPtr returns a pointer to here
Clock chip
InitUtil
WriteParam
20-byte section of low memory

Figure 5. Parameter RAM Routines

FUNCTION InitUtil : OSErr;

Trap macro _InitUtil

On exit D∅: result code (word)

InitUtil copies the contents of parameter RAM into 2∅ bytes of low memory and copies the date and time from the clock chip into its own low-memory location. This routine is called at system startup; you'll probably never need to call it yourself.

Assembly-language note: InitUtil copies parameter RAM into 2Ø bytes starting at the address SysParam and copies the date and time into the global variable Time.

If the validity status in parameter RAM is not $A8 when InitUtil is called, an error is returned as the result code, and the default values (given earlier in the "Parameter RAM" section) are read into the low-memory copy of parameter RAM; these values are then written to the clock chip itself.

Result codes		
	noErr	No error
	prInitErr	Validity status not $A8

FUNCTION GetSysPPtr : SysPPtr; [Not in ROM]

GetSysPPtr returns a pointer to the low-memory copy of parameter RAM. You can examine the values stored in its various fields, or to change them before calling WriteParam (below).

Assembly-language note: Assembly-language programmers can simply access the global variables corresponding to the low-memory copy of parameter RAM. These variables, which begin at the address SysParam, are listed in the summary.

FUNCTION WriteParam : OSErr;

Trap macro _WriteParam

On entry AØ: SysParam (pointer)
DØ: MinusOne (long word)

(You have to pass the values of these global variables for historical reasons.)

On exit DØ: result code (word)

WriteParam writes the low-memory copy of parameter RAM to the clock chip. You should previously have called GetSysPPtr and changed selected values as desired.

WriteParam also attempts to verify the values written by reading them back in and comparing them to the values in the low-memory copy.

(note)

> If you've accidentally write incorrect values into parameter RAM, the system may not be able to start up. If this happens, you can reset parameter RAM by removing the battery, letting the Macintosh sit turned off for about five minutes, and then putting the battery back in.

<u>Result codes</u>	noErr	No error
	prWrErr	Parameter RAM written did not verify

Queue Manipulation

This section describes utilities that advanced programmers may want to use for adding entries to or deleting entries from an Operating System queue. Normally you won't need to use these utilities, since queues are manipulated for you as necessary by routines that need to deal with them.

PROCEDURE Enqueue (qElement: QElemPtr; theQueue: QHdrPtr);

<u>Trap macro</u>	_Enqueue
<u>On entry</u>	AØ: qElement (pointer) A1: theQueue (pointer)
On exit	A1: theQueue (pointer)

Enqueue adds the queue entry pointed to by qElement to the end of the queue specified by theQueue.

(note)

> Interrupts are disabled for a short time while the queue is updated.

```
FUNCTION Dequeue (qElement: QElemPtr; theQueue: QHdrPtr) : OSErr;
```

Trap macro	_Dequeue	
On entry	AØ:	qElement (pointer)
	A1:	theQueue (pointer)
On exit	A1:	theQueue (pointer)
	DØ:	result code (word)

Dequeue removes the queue entry pointed to by qElement from the queue specified by theQueue (without deallocating the entry) and adjusts other entries in the queue accordingly.

(note)

> The note under Enqueue above also applies here. In this case, the amount of time interrupts are disabled depends on the length of the queue and the position of the entry in the queue.

(note)

> To remove all entries from a queue, you can just clear all the fields of the queue's header.

Result codes	noErr	No error
	qErr	Entry not in specified queue

Trap Dispatch Table Utilities

The Operating System Utilities include two routines for manipulating the trap dispatch table, which is described in detail in the manual Programming Macintosh Applications in Assembly Language. Using these routines, you can intercept calls to an Operating System or Toolbox routine and do some pre- or post-processing of your own: call GetTrapAddress to get the address of the original routine, save that address for later use, and call SetTrapAddress to install your own version of the routine in the dispatch table. Before or after its own processing, the new version of the routine can use the saved address to call the original version.

(warning)

> You can replace as well as intercept existing routines; in any case, you should be absolutely sure you know what you're doing. Remember that some calls that aren't in the ROM do some processing of their own before invoking a trap macro (for example, FSOpen eventually invokes _Open, and IUCompString invokes the macro for IUMagString). Also, a number of ROM routines have been patched with corrected versions in RAM; if you intercept a patched

routine, be sure to preserve the registers and the stack, or the system will not work properly.

Assembly-language note: You can tell whether a routine is patched by comparing its address to the global variable ROMBase; if the address is less than ROMBase, the routine is patched.

In addition, you can use GetTrapAddress to save time in critical sections of your program by calling an Operating System or Toolbox routine directly, avoiding the overhead of a normal trap dispatch.

FUNCTION GetTrapAddress (trapNum: INTEGER) : LONGINT;

Trap macro	_GetTrapAddress
On entry	DØ: trapNum (word)
On exit	AØ: address of routine

GetTrapAddress returns the address of a routine currently installed in the trap dispatch table under the trap number designated by trapNum. To find out the trap number for a particular routine, see Appendix B.

Assembly-language note: When you use this technique to bypass the trap dispatcher, you don't get the extra level of register saving. The routine itself will follow Lisa Pascal conventions and preserve A2-A6 and D3-D7, but if you want any other registers preserved across the call you have to save and restore them yourself.

PROCEDURE SetTrapAddress (trapAddr: LONGINT; trapNum: INTEGER);

Trap macro	_SetTrapAddress	
On entry	AØ:	trapAddr (address)
	DØ:	trapNum (word)

SetTrapAddress installs in the trap dispatch table a routine whose address is trapAddr; this routine is installed under the trap number designated by trapNum.

(note)

> Remember, the trap dispatch table can address locations within a range of 64K bytes from the base address of ROM or RAM.

Miscellaneous Utilities

PROCEDURE Delay (numTicks: LONGINT; VAR finalTicks: LONGINT);

Trap macro	_Delay	
On entry	AØ:	numTicks (long word)
On exit	DØ:	finalTicks (long word)

Delay causes the system to wait for the number of ticks (sixtieths of a second) specified by numTicks, and returns in finalTicks the total number of ticks from system startup to the end of the delay.

(warning)

> Do not rely on the duration of the delay being exact; it will usually be accurate within one tick, but may be off more than that. The Delay procedure enables all interrupts and checks the tick count that's incremented during the vertical retrace interrupt; however, it's possible for this interrupt to be disabled by other interrupts, in which case the duration of the delay will not be exactly what you requested.

Assembly-language note: On exit from this procedure, register DØ contains the value of the global variable Ticks as measured at the end of the delay.

PROCEDURE SysBeep (duration: INTEGER);

SysBeep causes the system to beep for approximately the number of ticks specified by the duration parameter. The sound decays from loud to soft; after about five seconds it's inaudible. The initial volume of the beep depends on the current speaker volume setting, which the user can adjust with the Control Panel desk accessory. If the speaker volume has been set to Ø (silent), SysBeep instead causes the menu bar to blink once.

Assembly-language note: Unlike all other Operating System Utilities, this procedure is stack-based.

SUMMARY OF THE OPERATING SYSTEM UTILITIES

Constants

```
CONST { Result codes }

      clkRdErr     = -85;    {unable to read clock}
      clkWrErr     = -86;    {time written did not verify}
      memFullErr   = -108;   {not enough room in heap}
      memWZErr     = -111;   {attempt to operate on a free block}
      nilHandleErr = -109;   {NIL master pointer}
      noErr        =  0;     {no error}
      prInitErr    = -88;    {validity status is not $A8}
      prWrErr      = -87;    {parameter RAM written did not verify}
      qErr         = -1;     {entry not in specified queue}
```

Data Types

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

     OSErr = INTEGER;

     SysPPtr     = ^SysParmType;
     SysParmType =
       RECORD
         valid:     LONGINT;  {validity status}
         portA:     INTEGER;  {modem port configuration}
         portB:     INTEGER;  {printer port configuration}
         alarm:     LONGINT;  {alarm setting}
         font:      INTEGER;  {default application font number minus 1}
         kbdPrint:  INTEGER;  {auto-key threshold and rate, printer }
                              { connection}
         volClik:   INTEGER;  {speaker volume, double-click and caret- }
                              { blink times}
         misc:      INTEGER   {mouse scaling, system startup disk, menu }
                              { blink}
       END;

     QHdrPtr = ^QHdr;
     QHdr    = RECORD
                 qFlags: INTEGER;  {queue flags}
                 qHead:  QElemPtr; {first queue entry}
                 qTail:  QElemPtr  {last queue entry}
               END;
```

```
QTypes    = (dummyType,
             vType,     {vertical retrace queue type}
             ioQType,   {file I/O or driver I/O queue type}
             drvQType,  {drive queue type}
             evType,    {event queue type}
             fsQType);  {volume-control-block queue type}

QElemPtr = ^QElem;
QElem    = RECORD
              CASE QTypes OF
                vType:    (vblQElem: VBLTask);
                ioQType:  (ioQElem:  ParamBlockRec);
                drvQType: (drvQElem: DrvQEl);
                evType:   (evQElem:  EvQEl);
                fsQType:  (vcbQElem: VCB)
             END;

DateTimeRec =
  RECORD
    year:      INTEGER;  {19ø4 to 2ø4ø}
    month:     INTEGER;  {1 to 12 for January to December}
    day:       INTEGER;  {1 to 31}
    hour:      INTEGER;  {ø to 23}
    minute:    INTEGER;  {ø to 59}
    second:    INTEGER;  {ø to 59}
    dayOfWeek: INTEGER   {1 to 7 for Sunday to Saturday}
  END;
```

Routines

Pointer and Handle Manipulation

```
FUNCTION  HandToHand  (VAR theHndl: Handle) : OSErr;
FUNCTION  PtrToHand   (srcPtr: Ptr; VAR dstHndl: Handle; size:
                       LONGINT) : OSErr;
FUNCTION  PtrToXHand  (srcPtr: Ptr; dstHndl: Handle; size: LONGINT) :
                       OSErr;
FUNCTION  HandAndHand (aHndl,bHndl: Handle) : OSErr;
FUNCTION  PtrAndHand  (pntr: Ptr; hndl: Handle; size: LONGINT) : OSErr;
```

String Comparison

```
FUNCTION  EqualString (aStr,bStr: Str255; caseSens,diacSens: BOOLEAN) :
                       BOOLEAN;
PROCEDURE UprString   (VAR theString: Str255; diacSens: BOOLEAN);
```

Date and Time Operations

```
FUNCTION  ReadDateTime (VAR secs: LONGINT) : OSErr;
PROCEDURE GetDateTime  (VAR secs: LONGINT);  [Not in ROM]
FUNCTION  SetDateTime  (secs: LONGINT) : OSErr;
PROCEDURE Date2Secs    (date: DateTimeRec; VAR secs: LONGINT);
PROCEDURE Secs2Date    (secs: LONGINT; VAR date: DateTimeRec);
PROCEDURE GetTime      (VAR date: DateTimeRec);  [Not in ROM]
PROCEDURE SetTime      (date: DateTimeRec);  [Not in ROM]
```

Parameter RAM Operations

```
FUNCTION InitUtil :   OSErr;
FUNCTION GetSysPPtr : SysPPtr;  [Not in ROM]
FUNCTION WriteParam : OSErr;
```

Queue Manipulation

```
PROCEDURE Enqueue (qElement: QElemPtr; theQueue: QHdrPtr);
FUNCTION  Dequeue (qElement: QElemPtr; theQueue: QHdrPtr) : OSErr;
```

Trap Dispatch Table Utilities

```
PROCEDURE SetTrapAddress (trapAddr: LONGINT; trapNum: INTEGER);
FUNCTION  GetTrapAddress (trapNum: INTEGER) : LONGINT;
```

Miscellaneous Utilities

```
PROCEDURE Delay   (numTicks: LONGINT; VAR finalTicks: LONGINT);
PROCEDURE SysBeep (duration: INTEGER);
```

Default Parameter RAM Values

Parameter	Default value
Validity status	$A8
Modem port configuration	96ØØ baud, 8 data bits, 2 stop bits, no parity
Printer port configuration	Same as for modem port
Alarm setting	Ø (midnight, January 1, 19Ø4)
Default application font - 1	2 (Geneva)
Auto-key threshold	6 (24 ticks)
Auto-key rate	3 (6 ticks)
Printer connection	Ø (printer port)
Speaker volume	3 (medium)
Double-click time	8 (32 ticks)
Caret-blink time	8 (32 ticks)
Mouse scaling	1 (on)
Preferred system startup disk	Ø (internal drive)
Menu blink	3

Assembly-Language Information

Constants

```
; Result codes

clkRdErr       .EQU  -85    ;unable to read clock
clkWrErr       .EQU  -86    ;time written did not verify
memFullErr     .EQU  -1Ø8   ;not enough room in heap
memWZErr       .EQU  -111   ;attempt to operate on a free block
nilHandleErr   .EQU  -1Ø9   ;NIL master pointer
noErr          .EQU   Ø     ;no error
prInitErr      .EQU  -88    ;validity status is not $A8
prWrErr        .EQU  -87    ;parameter RAM written did not verify
qErr           .EQU  -1     ;entry not in specified queue

; Queue types

vType          .EQU   1     ;vertical retrace queue type
ioQType        .EQU   2     ;file I/O or driver I/O queue type
drvQType       .EQU   3     ;drive queue type
evType         .EQU   4     ;event queue type
fsQType        .EQU   5     ;volume-control-block queue type
```

Queue Data Structure

qFlags	Queue flags (word)
qHead	Pointer to first queue entry
qTail	Pointer to last queue entry

Date/Time Record Data Structure

dtYear	19Ø4 to 2Ø4Ø (word)
dtMonth	1 to 12 for January to December (word)
dtDay	1 to 31 (word)
dtHour	Ø to 23 (word)
dtMinute	Ø to 59 (word)
dtSecond	Ø to 59 (word)
dtDayOfWeek	1 to 7 for Sunday to Saturday (word)

Routines

Name	On entry	On exit
HandToHand	AØ: theHndl (handle)	AØ: theHndl (handle) DØ: result code (word)
PtrToHand	AØ: srcPtr (ptr) DØ: size (long)	AØ: dstHndl (handle) DØ: result code (word)
PtrToXHand	AØ: srcPtr (ptr) A1: dstHndl (handle) DØ: size (long)	A1: dstHndl (handle) DØ: result code (word)
HandAndHand	AØ: aHndl (handle) A1: bHndl (handle)	A1: bHndl (handle) DØ: result code (word)
PtrAndHand	AØ: pntr (ptr) A1: hndl (handle) DØ: size (long)	A1: hndl (handle) DØ: result code (word)
CmpString	_CmpString ,MARKS sets bit 9, for diacSens=FALSE _CmpString ,CASE sets bit 1Ø, for caseSens=TRUE _CmpString ,MARKS,CASE sets bits 9 and 1Ø	
	AØ: ptr to first string A1: ptr to second string DØ: high word: length of first string low word: length of second string	DØ: Ø if equal, 1 if not equal (long)
UprString	_UprString ,MARKS sets bit 9, for diacSens=FALSE	
	AØ: ptr to string DØ: length of string (word)	AØ: ptr to string
ReadDateTime	AØ: ptr to long word secs	AØ: ptr to long word secs DØ: result code (word)
SetDateTime	DØ: secs (long)	DØ: result code (word)
Date2Secs	AØ: ptr to date/time record DØ: secs (long)	
Secs2Date	DØ: secs (long)	AØ: ptr to date/time record

InitUtil	DØ: result code (word)	
WriteParam	AØ: SysParam (ptr) DØ: MinusOne (long)	DØ: result code (word)
Enqueue	AØ: qElement (ptr)	A1: theQueue (ptr)
Dequeue	AØ: qElement (ptr) A1: theQueue (ptr)	A1: theQueue (ptr) DØ: result code (word)
GetTrapAddress	DØ: trapNum (word)	DØ: address of routine
SetTrapAddress	AØ: trapAddr (address) DØ: trapNum (word)	
Delay	AØ: numTicks (long)	DØ: finalTicks (long)
SysBeep	Push duration (word) onto stack	

Variables

SysParam	Low-memory copy of parameter RAM (2Ø bytes)
SPValid	Validity status (byte)
SPPortA	Modem port configuration (word)
SPPortB	Printer port configuration (word)
SPAlarm	Alarm setting (long)
SPFont	Default application font number minus 1 (word)
SPKbd	Auto-key threshold and rate (byte)
SPPrint	Printer connection (byte)
SPVolCtl	Speaker volume (byte)
SPClikCaret	Double-click and caret-blink times (byte)
SPMisc2	Mouse scaling, system startup disk, menu blink (byte)
CrsrThresh	Mouse-scaling threshold (word)
Time	Seconds since midnight, January 1, 19Ø4 (long)

GLOSSARY

auto-key rate: The rate at which a key repeats after it's begun to do so.

auto-key threshold: The length of time a key must be held down before it begins to repeat.

caret-blink time: The interval between blinks of the caret that marks an insertion point.

clock chip: A special chip in which are stored parameter RAM and the current settings for the date and time. This chip runs on a battery when the system is off, thus preserving the information.

date/time record: An alternate representation of the date and time (which is stored on the clock chip in seconds since midnight, January 1, 19Ø4).

double-click time: The greatest interval between a mouse-up and mouse-down event that would qualify two mouse clicks as a double-click.

mouse scaling: A feature that causes the cursor to move twice as far during a mouse stroke than it would have otherwise, provided the change in the mouse's position in a sixtieth of a second exceeds the mouse-scaling threshold.

mouse-scaling threshold: A number of pixels which, if exceeded by the sum of the horizontal and vertical changes in the mouse position during a sixtieth of a second, causes mouse scaling to occur (if that feature is turned on); normally six pixels.

parameter RAM: In the clock chip, 2Ø bytes where settings such as those made with the Control Panel desk accessory are preserved.

queue: A list of identically structured entries linked together by pointers.

result code: An integer indicating whether a routine completed its task successfully or was prevented by some error condition.

validity status: A number stored in parameter RAM designating whether the last attempt to write there was successful. (The number is $A8 if so.)

APPENDIX A: RESULT CODES

This appendix lists all the result codes returned by routines in the Macintosh Operating System, as well as a few that are returned by the Resource Manager and Scrap Manager of the User Interface Toolbox. They're ordered by value, for convenience when debugging; the names you should actually use in your program are also listed.

The result codes are grouped roughly according to the lowest level at which the error may occur. This doesn't mean that only routines at that level may cause those errors; higher-level software may yield the same result codes. For example, an Operating System Utility routine that calls the Memory Manager may return one of the Memory Manager result codes. Where a different or more specific meaning is appropriate in a different context, that meaning is also listed.

Value	Name	Meaning
Ø	noErr	No error
Operating System Event Manager Error		
1	evtNotEnb	Event type not designated in system event mask
Queuing Errors		
-1	qErr	Entry not in queue
-2	vTypErr	QType field of entry in vertical retrace queue isn't vType (in Pascal, ORD(vType))
Device Manager Errors		
-17	controlErr	Driver can't respond to this Control call
-18	statusErr	Driver can't respond to this Status call
-19	readErr	Driver can't respond to Read calls
-2Ø	writErr	Driver can't respond to Write calls
-21	badUnitErr	Driver reference number doesn't match unit table
-22	unitEmptyErr	Driver reference number specifies NIL handle in unit table
-23	openErr	Requested read/write permission doesn't match driver's open permission Attempt to open RAM Serial Driver failed
-25	dRemovErr	Attempt to remove an open device driver
-26	dInstErr	Couldn't find driver in resource file
-27	abortErr	I/O request aborted by KillIO
-28	notOpenErr	Driver isn't open
File Manager Errors		
-33	dirFulErr	File directory full
-34	dskFulErr	All allocation blocks on the volume are full
-35	nsvErr	Specified volume doesn't exist

-36	ioErr	I/O error
-37	bdNamErr	Bad file name or volume name (perhaps zero-length)
-38	fnOpnErr	File not open
-39	eofErr	Logical end-of-file reached during read operation
-4Ø	posErr	Attempt to position before start of file
-41	mFulErr	Memory full
-42	tmfoErr	Too many files open; only 12 files can be open simultaneously
-43	fnfErr	File not found
-44	wPrErr	Volume is locked by a hardware setting
-45	fLckdErr	File is locked
-46	vLckdErr	Volume is locked by a software flag
-47	fBsyErr	File is busy; one or more files are open
-48	dupFNErr	File with specified name and version number already exists
-49	opWrErr	The read/write permission of only one access path to a file can allow writing
-5Ø	paramErr	Error in parameter list Parameters don't specify an existing volume, and there's no default volume (File Manager) Bad positioning information (Disk Driver) Bad drive number (Disk Initialization Package)
-51	rfNumErr	Path reference number specifies nonexistent access path
-52	gfpErr	Error during GetFPos *** will be in next draft of File Manager manual ***
-53	volOffLinErr	Volume not on-line
-54	permErr	Read/write permission doesn't allow writing
-55	volOnLinErr	Specified volume is already mounted and on-line
-56	nsDrvErr	No such drive; specified drive number doesn't match any number in the drive queue
-57	noMacDskErr	Not a Macintosh disk; volume lacks Macintosh-format directory
-58	extFSErr	External file system; file-system identifier is nonzero, or path reference number is greater than 1Ø24
-59	fsRnErr	Problem during rename
-6Ø	badMDBErr	Bad master directory block; must reinitialize volume
-61	wrPermErr	Read/write permission or open permission doesn't allow writing

Low-Level Disk Errors

-64	noDriveErr	Drive isn't connected
-65	offLinErr	No disk in drive
-66	noNybErr	Disk is probably blank
-67	noAdrMkErr	Can't find an address mark
-68	dataVerErr	Read-verify failed
-69	badCkSmErr	Bad address mark
-7Ø	badBtSlpErr	Bad address mark

-71	noDtaMkErr	Can't find a data mark
-72	badDCksum	Bad data mark
-73	badDBtSlp	Bad data mark
-75	cantStepErr	Hardware error
-76	tkOBadErr	Hardware error.
-77	initIWMErr	Hardware error
-78	twoSideErr	Tried to read side 2 of a disk in a single-sided drive
-79	spdAdjErr	Hardware error
-80	seekErr	Hardware error
-81	sectNFErr	Can't find sector

Also, to check for any low-level disk error:

-84	firstDskErr	First of the range of low-level disk errors
-64	lastDskErr	Last of the range of low-level disk errors

Clock Chip Errors

-85	clkRdErr	Unable to read clock
-86	clkWrErr	Time written did not verify
-87	prWrErr	Parameter RAM written did not verify
-88	prInitErr	Validity status is not $A8

Scrap Manager Errors

-100	noScrapErr	Desk scrap isn't initialized
-102	noTypeErr	No data of the requested type

Memory Manager Errors

-108	memFullErr	Not enough room in heap zone
-109	nilHandleErr	NIL master pointer
-111	memWZErr	Attempt to operate on a free block
-112	memPurErr	Attempt to purge a locked block

Resource Manager Errors

-192	resNotFound	Resource not found
-193	resFNotFound	Resource file not found
-194	addResFailed	AddResource failed
-195	addRefFailed	AddReference failed
-196	rmvResFailed	RmveResource failed
-197	rmvRefFailed	RmveReference failed

APPENDIX B: SYSTEM TRAPS

This appendix lists the trap macros for the Toolbox and Operating System routines and their corresponding trap word values in hexadecimal. The "Name" column gives the trap macro name (without its initial underscore character). In those cases where the name of the equivalent Pascal call is different, the Pascal name appears indented under the main entry. The routines in Macintosh packages are listed under the macros they invoke after pushing a routine selector onto the stack; the routine selector follows the Pascal routine name in parentheses.

There are two tables: The first is ordered alphabetically by name; the second is ordered numerically by trap number, for usefulness when debugging.

(note)

> The Operating System Utility routines GetTrapAddress and SetTrapAddress take a trap number as a parameter, not a trap word. You can get the trap number from the trap word as follows: If the trap word begins with AØ or A8, the last two digits are the trap number; if it begins with A9, the trap number is 1 followed by the last two digits of the trap word.

Name	Trap Word
AddDrive	AØ4E
AddPt	A87E
AddReference	A9AC
AddResMenu	A94D
AddResource	A9AB
Alert	A985
Allocate	AØ1Ø
PBAllocate	
AngleFromSlope	A8C4
AppendMenu	A933
BackColor	A863
BackPat	A87C
BeginUpdate	A922
BitAnd	A858
BitClr	A85F
BitNot	A85A
BitOr	A85B
BitSet	A85E
BitShift	A85C
BitTst	A85D
BitXor	A859
BlockMove	AØ2E
BringToFront	A92Ø
Button	A974
CalcMenuSize	A948
CalcVBehind	A9ØA
CalcVisBehind	
CalcVis	A9Ø9
CautionAlert	A988
Chain	A9F3
ChangedResource	A9AA
CharWidth	A88D
CheckItem	A945
CheckUpdate	A911
ClearMenuBar	A934
ClipAbove	A9ØB
ClipRect	A87B
Close	AØØ1
PBClose	
CloseDeskAcc	A9B7
CloseDialog	A982
ClosePgon	A8CC
ClosePoly	
ClosePicture	A8F4
ClosePort	A87D
CloseResFile	A99A
CloseRgn	A8DB
CloseWindow	A92D
CmpString	AØ3C
EqualString	

ColorBit	A864
CompactMem	AØ4C
Control	AØØ4
PBControl	
CopyBits	A8EC
CopyRgn	A8DC
CouldAlert	A989
CouldDialog	A979
CountMItems	A95Ø
CountResources	A99C
CountTypes	A99E
Create	AØØ8
PBCreate	
CreateResFile	A9B1
CurResFile	A994
Date2Secs	A9C7
Delay	AØ3B
Delete	AØØ9
PBDelete	
DeleteMenu	A936
DeltaPoint	A94F
Dequeue	A96E
DetachResource	A992
DialogSelect	A98Ø
DiffRgn	A8E6
DisableItem	A93A
DisposControl	A955
DisposeControl	
DisposDialog	A983
DisposHandle	AØ23
DisposMenu	A932
DisposeMenu	
DisposPtr	AØ1F
DisposRgn	A8D9
DisposeRgn	
DisposWindow	A914
DisposeWindow	
DragControl	A967
DragGrayRgn	A9Ø5
DragTheRgn	A926
DragWindow	A925
DrawChar	A883
DrawControls	A969
DrawDialog	A981
DrawGrowIcon	A9Ø4
DrawMenuBar	A937
DrawNew	A9ØF
DrawPicture	A8F6
DrawString	A884
DrawText	A885
DrvrInstall	AØ3D
(internal use only)	
DrvrRemove	AØ3E
(internal use only)	

Eject	AØ17
PBEject	
EmptyHandle	AØ2B
EmptyRect	A8AE
EmptyRgn	A8E2
EnableItem	A939
EndUpdate	A923
Enqueue	A96F
EqualPt	A881
EqualRect	A8A6
EqualRgn	A8E3
EraseArc	A8CØ
EraseOval	A8B9
ErasePoly	A8C8
EraseRect	A8A3
EraseRgn	A8D4
EraseRoundRect	A8B2
ErrorSound	A98C
EventAvail	A971
ExitToShell	A9F4
FillArc	A8C2
FillOval	A8BB
FillPoly	A8CA
FillRect	A8A5
FillRgn	A8D6
FillRoundRect	A8B4
FindControl	A96C
FindWindow	A92C
FixMul	A868
FixRatio	A869
FixRound	A86C
FlashMenuBar	A94C
FlushEvents	AØ32
FlushFile	AØ45
PBFlushFile	
FlushVol	AØ13
PBFlushVol	
FMSwapFont	A9Ø1
SwapFont	
ForeColor	A862
FrameArc	A8BE
FrameOval	A8B7
FramePoly	A8C6
FrameRect	A8A1
FrameRgn	A8D2
FrameRoundRect	A8BØ
FreeAlert	A98A
FreeDialog	A97A
FreeMem	AØ1C
FrontWindow	A924
GetAppParms	A9F5
GetClip	A87A
GetCRefCon	A95A
GetCTitle	A95E

GetCtlAction	A96A
GetCtlValue	A96Ø
GetCursor	A9B9
GetDItem	A98D
GetEOF	AØ11
PBGetEOF	
GetFileInfo	AØØC
PBGetFInfo	
GetFName	A8FF
GetFontName	
GetFNum	A9ØØ
GetFontInfo	A88B
GetFPos	AØ18
PBGetFPos	
GetHandleSize	AØ25
GetIcon	A9BB
GetIndResource	A99D
GetIndType	A99F
GetItem	A946
GetIText	A99Ø
GetItmIcon	A93F
GetItemIcon	
GetItmMark	A943
GetItemMark	
GetItmStyle	A941
GetItemStyle	
GetKeys	A976
GetMaxCtl	A962
GetCtlMax	
GetMenuBar	A93B
GetMHandle	A949
GetMinCtl	A961
GetCtlMin	
GetMouse	A972
GetNamedResource	A9A1
GetNewControl	A9BE
GetNewDialog	A97C
GetNewMBar	A9CØ
GetNewWindow	A9BD
GetNextEvent	A97Ø
GetOSEvent	AØ31
GetPattern	A9B8
GetPen	A89A
GetPenState	A898
GetPicture	A9BC
GetPixel	A865
GetPort	A874
GetPtrSize	AØ21
GetResAttrs	A9A6
GetResFileAttrs	A9F6
GetResInfo	A9A8
GetResource	A9AØ
GetRMenu	A9BF
GetMenu	
GetScrap	A9FD
GetString	A9BA
GetTrapAddress	AØ46
GetVol	AØ14
PBGetVol	
GetVolInfo	AØØ7
PBGetVInfo	
GetWindowPic	A92F
GetWMgrPort	A91Ø
GetWRefCon	A917
GetWTitle	A919
GetZone	AØ1A
GlobalToLocal	A871
GrafDevice	A872
GrowWindow	A92B
HandAndHand	A9E4
HandleZone	AØ26
HandToHand	A9E1
HideControl	A958
HideCursor	A852
HidePen	A896
HideWindow	A916
HiliteControl	A95D
HiliteMenu	A938
HiliteWindow	A91C
HiWord	A86A
HLock	AØ29
HNoPurge	AØ4A
HomeResFile	A9A4
HPurge	AØ49
HUnlock	AØ2A
InfoScrap	A9F9
InitAllPacks	A9E6
InitApplZone	AØ2C
InitCursor	A85Ø
InitDialogs	A97B
InitFonts	A8FE
InitGraf	A86E
InitMenus	A93Ø
InitPack	A9E5
InitPort	A86D
InitQueue	AØ16
FInitQueue *** File Mgr. InitQueue routine will be renamed this ***	
InitResources	A995
InitUtil	AØ3F
InitWindows	A912
InitZone	AØ19
InsertMenu	A935
InsertResMenu	A951
InsetRect	A8A9
InsetRgn	A8E1
InvalRect	A928

InvalRgn	A927
InverRect	A8A4
InvertRect	
InverRgn	A8D5
InvertRgn	
InverRoundRect	A8B3
InvertRoundRect	
InvertArc	A8C1
InvertOval	A8BA
InvertPoly	A8C9
IsDialogEvent	A97F
KillControls	A956
KillIO	AØØ6
PBKillIO	
KillPicture	A8F5
KillPoly	A8CD
Launch	A9F2
Line	A892
LineTo	A891
LoadResource	A9A2
LoadSeg	A9FØ
LocalToGlobal	A87Ø
LodeScrap	A9FB
LoadScrap	
LongMul	A867
LoWord	A86B
MapPoly	A8FC
MapPt	A8F9
MapRect	A8FA
MapRgn	A8FB
MaxMem	AØ1D
MenuKey	A93E
MenuSelect	A93D
ModalDialog	A991
MoreMasters	AØ36
MountVol	AØØF
PBMountVol	
Move	A894
MoveControl	A959
MovePortTo	A877
MoveTo	A893
MoveWindow	A91B
Munger	A9EØ
NewControl	A954
NewDialog	A97D
NewHandle	AØ22
NewMenu	A931
NewPtr	AØ1E
NewRgn	A8D8
NewString	A9Ø6
NewWindow	A913
NoteAlert	A987
ObscureCursor	A856
Offline	AØ35
PBOffline	
OffsetPoly	A8CE
OffsetRect	A8A8
OfsetRgn	A8EØ
OffsetRgn	
Open	AØØØ
PBOpen	
OpenDeskAcc	A9B6
OpenPicture	A8F3
OpenPoly	A8CB
OpenPort	A86F
OpenResFile	A997
OpenRF	AØØA
PBOpenRF	
OpenRgn	A8DA
OSEventAvail	AØ3Ø
PackØ (not used)	A9E7
Pack1 (not used)	A9E8
Pack2	A9E9
DIBadMount (Ø)	
DIFormat (6)	
DILoad (2)	
DIUnload (4)	
DIVerify (8)	
DIZero (1Ø)	
Pack3	A9EA
SFGetFile (2)	
SFPGetFile (4)	
SFPPutFile (3)	
SFPutFile (1)	
Pack4	A9EB
Pack5	A9EC
Pack6	A9ED
IUDatePString (14)	
IUDateString (Ø)	
IUGetIntl (6)	
IUMagIDString (12)	
IUMagString (1Ø)	
IUMetric (4)	
IUSetIntl (8)	
IUTimePString (16)	
IUTimeString (2)	
Pack7	A9EE
NumToString (Ø)	
StringToNum (1)	
PackBits	A8CF
PaintArc	A8BF
PaintBehind	A9ØD
PaintOne	A9ØC
PaintOval	A8B8
PaintPoly	A8C7
PaintRect	A8A2
PaintRgn	A8D3

PaintRoundRect	A8B1
ParamText	A98B
PenMode	A89C
PenNormal	A89E
PenPat	A89D
PenSize	A89B
PicComment	A8F2
PinRect	A94E
PlotIcon	A94B
PortSize	A876
PostEvent	AØ2F
Pt2Rect	A8AC
PtInRect	A8AD
PtInRgn	A8E8
PtrAndHand	A9EF
PtrToHand	A9E3
PtrToXHand	A9E2
PtrZone	AØ48
PtToAngle	A8C3
PurgeMem	AØ4D
PutScrap	A9FE
Random	A861
RDrvrInstall	AØ4F
Read	AØØ2
PBRead	
ReadDateTime	AØ39
RealFont	A9Ø2
ReallocHandle	AØ27
RecoverHandle	AØ28
RectInRgn	A8E9
RectRgn	A8DF
ReleaseResource	A9A3
Rename	AØØB
PBRename	
ResError	A9AF
ResrvMem	AØ4Ø
RmveReference	A9AE
RmveResource	A9AD
RsrcZoneInit	A996
RstFilLock	AØ42
PBRstFLock	
SaveOld	A9ØE
ScalePt	A8F8
ScrollRect	A8EF
Secs2Date	A9C6
SectRect	A8AA
SectRgn	A8E4
SelectWindow	A91F
SelIText	A97E
SendBehind	A921
SetAppBase	AØ57
SetApplBase	
SetApplLimit	AØ2D
SetClip	A879

SetCRefCon	A95B
SetCTitle	A95F
SetCtlAction	A96B
SetCtlValue	A963
SetCursor	A851
SetDateTime	AØ3A
SetDItem	A98E
SetEmptyRgn	A8DD
SetEOF	AØ12
PBSetEOF	
SetFileInfo	AØØD
PBSetFInfo	
SetFilLock	AØ41
PBSetFLock	
SetFilType	AØ43
PBSetFVers	
SetFontLock	A9Ø3
SetFPos	AØ44
PBSetFPos	
SetGrowZone	AØ4B
SetHandleSize	AØ24
SetItem	A947
SetIText	A98F
SetItmIcon	A94Ø
SetItemIcon	
SetItmMark	A944
SetItemMark	
SetItmStyle	A942
SetItemStyle	
SetMaxCtl	A965
SetCtlMax	
SetMenuBar	A93C
SetMFlash	A94A
SetMenuFlash	
SetMinCtl	A964
SetCtlMin	
SetOrigin	A878
SetPBits	A875
SetPortBits	
SetPenState	A899
SetPort	A873
SetPt	A88Ø
SetPtrSize	AØ2Ø
SetRecRgn	A8DE
SetRectRgn	
SetRect	A8A7
SetResAttrs	A9A7
SetResFileAttrs	A9F7
SetResInfo	A9A9
SetResLoad	A99B
SetResPurge	A993
SetStdProcs	A8EA
SetString	A9Ø7
SetTrapAddress	AØ47

SetVol	AØ15
PBSetVol	
SetWindowPic	A92E
SetWRefCon	A918
SetWTitle	A91A
SetZone	AØ1B
ShieldCursor	A855
ShowControl	A957
ShowCursor	A853
ShowHide	A9Ø8
ShowPen	A897
ShowWindow	A915
SizeControl	A95C
SizeRsrc	A9A5
SizeResource	
SizeWindow	A91D
SlopeFromAngle	A8BC
SpaceExtra	A88E
Status	AØØ5
PBStatus	
StdArc	A8BD
StdBits	A8EB
StdComment	A8F1
StdGetPic	A8EE
StdLine	A89Ø
StdOval	A8B6
StdPoly	A8C5
StdPutPic	A8FØ
StdRect	A8AØ
StdRgn	A8D1
StdRRect	A8AF
StdText	A882
StdTxMeas	A8ED
StillDown	A973
StopAlert	A986
StringWidth	A88C
StuffHex	A866
SubPt	A87F
SysBeep	A9C8
SysEdit	A9C2
SystemEdit	
SysError	A9C9
SystemClick	A9B3
SystemEvent	A9B2
SystemMenu	A9B5
SystemTask	A9B4
TEActivate	A9D8
TECalText	A9DØ
TEClick	A9D4
TECopy	A9D5
TECut	A9D6
TEDeactivate	A9D9
TEDelete	A9D7
TEDispose	A9CD
TEGetText	A9CB
TEIdle	A9DA
TEInit	A9CC
TEInsert	A9DE
TEKey	A9DC
TENew	A9D2
TEPaste	A9DB
TEScroll	A9DD
TESetJust	A9DF
TESetSelect	A9D1
TESetText	A9CF
TestControl	A966
TEUpdate	A9D3
TextBox	A9CE
TextFace	A888
TextFont	A887
TextMode	A889
TextSize	A88A
TextWidth	A886
TickCount	A975
TrackControl	A968
TrackGoAway	A91E
UnionRect	A8AB
UnionRgn	A8E5
UniqueID	A9C1
UnloadSeg	A9F1
UnlodeScrap	A9FA
UnloadScrap	
UnmountVol	AØØE
PBUnmountVol	
UnpackBits	A8DØ
UpdateResFile	A999
UprString	AØ54
UseResFile	A998
ValidRect	A92A
ValidRgn	A929
VInstall	AØ33
VRemove	AØ34
WaitMouseUp	A977
Write	AØØ3
PBWrite	
WriteParam	AØ38
WriteResource	A9BØ
XorRgn	A8E7
ZeroScrap	A9FC

Trap Word	Name
AØØØ	Open
	PBOpen
AØØ1	Close
	PBClose
AØØ2	Read
	PBRead
AØØ3	Write
	PBWrite
AØØ4	Control
	PBControl
AØØ5	Status
	PBStatus
AØØ6	KillIO
	PBKillIO
AØØ7	GetVolInfo
	PBGetVInfo
AØØ8	Create
	PBCreate
AØØ9	Delete
	PBDelete
AØØA	OpenRF
	PBOpenRF
AØØB	Rename
	PBRename
AØØC	GetFileInfo
	PBGetInfo
AØØD	SetFileInfo
	PBSetFInfo
AØØE	UnmountVol
	PBUnmountVol
AØØF	MountVol
	PBMountVol
AØ1Ø	Allocate
	PBAllocate
AØ11	GetEOF
	PBGetEOF
AØ12	SetEOF
	PBSetEOF
AØ13	FlushVol
	PBFlushVol
AØ14	GetVol
	PBGetVol
AØ15	SetVol
	PBSetVol
AØ16	InitQueue
	FInitQueue *** File Mgr. InitQueue routine will be renamed this ***
AØ17	Eject
	PBEject
AØ18	GetFPos
	PBGetFPos
AØ19	InitZone
AØ1A	GetZone
AØ1B	SetZone
AØ1C	FreeMem
AØ1D	MaxMem
AØ1E	NewPtr
AØ1F	DisposPtr
AØ2Ø	SetPtrSize
AØ21	GetPtrSize
AØ22	NewHandle
AØ23	DisposHandle
AØ24	SetHandleSize
AØ25	GetHandleSize
AØ26	HandleZone
AØ27	ReallocHandle
AØ28	RecoverHandle
AØ29	HLock
AØ2A	HUnlock
AØ2B	EmptyHandle
AØ2C	InitApplZone
AØ2D	SetApplLimit
AØ2E	BlockMove
AØ2F	PostEvent
AØ3Ø	OSEventAvail
AØ31	GetOSEvent
AØ32	FlushEvents
AØ33	VInstall
AØ34	VRemove
AØ35	Offline
	PBOffline
AØ36	MoreMasters
AØ38	WriteParam
AØ39	ReadDateTime
AØ3A	SetDateTime
AØ3B	Delay
AØ3C	CmpString
	EqualString
AØ3D	DrvrInstall
	(internal use only)
AØ3E	DrvrRemove
	(internal use only)
AØ3F	InitUtil
AØ4Ø	ResrvMem
AØ41	SetFilLock
	PBSetFLock
AØ42	RstFilLock
	PBRstFLock
AØ43	SetFilType
	PBSetFVers
AØ44	SetFPos
	PBSetFPos
AØ45	FlushFile
	PBFlushFile
AØ46	GetTrapAddress
AØ47	SetTrapAddress
AØ48	PtrZone

AØ49	HPurge
AØ4A	HNoPurge
AØ4B	SetGrowZone
AØ4C	CompactMem
AØ4D	PurgeMem
AØ4E	AddDrive
AØ4F	RDrvrInstall
AØ54	UprString
AØ57	SetAppBase
	SetApplBase
A85Ø	InitCursor
A851	SetCursor
A852	HideCursor
A853	ShowCursor
A855	ShieldCursor
A856	ObscureCursor
A858	BitAnd
A859	BitXor
A85A	BitNot
A85B	BitOr
A85C	BitShift
A85D	BitTst
A85E	BitSet
A85F	BitClr
A861	Random
A862	ForeColor
A863	BackColor
A864	ColorBit
A865	GetPixel
A866	StuffHex
A867	LongMul
A868	FixMul
A869	FixRatio
A86A	HiWord
A86B	LoWord
A86C	FixRound
A86D	InitPort
A86E	InitGraf
A86F	OpenPort
A87Ø	LocalToGlobal
A871	GlobalToLocal
A872	GrafDevice
A873	SetPort
A874	GetPort
A875	SetPBits
	SetPortBits
A876	PortSize
A877	MovePortTo
A878	SetOrigin
A879	SetClip
A87A	GetClip
A87B	ClipRect
A87C	BackPat
A87D	ClosePort
A87E	AddPt
A87F	SubPt
A88Ø	SetPt
A881	EqualPt
A882	StdText
A883	DrawChar
A884	DrawString
A885	DrawText
A886	TextWidth
A887	TextFont
A888	TextFace
A889	TextMode
A88A	TextSize
A88B	GetFontInfo
A88C	StringWidth
A88D	CharWidth
A88E	SpaceExtra
A89Ø	StdLine
A891	LineTo
A892	Line
A893	MoveTo
A894	Move
A896	HidePen
A897	ShowPen
A898	GetPenState
A899	SetPenState
A89A	GetPen
A89B	PenSize
A89C	PenMode
A89D	PenPat
A89E	PenNormal
A8AØ	StdRect
A8A1	FrameRect
A8A2	PaintRect
A8A3	EraseRect
A8A4	InverRect
	InvertRect
A8A5	FillRect
A8A6	EqualRect
A8A7	SetRect
A8A8	OffsetRect
A8A9	InsetRect
A8AA	SectRect
A8AB	UnionRect
A8AC	Pt2Rect
A8AD	PtInRect
A8AE	EmptyRect
A8AF	StdRRect
A8BØ	FrameRoundRect
A8B1	PaintRoundRect
A8B2	EraseRoundRect
A8B3	InverRoundRect
	InvertRoundRect
A8B4	FillRoundRect

A8B6	StdOval
A8B7	FrameOval
A8B8	PaintOval
A8B9	EraseOval
A8BA	InvertOval
A8BB	FillOval
A8BC	SlopeFromAngle
A8BD	StdArc
A8BE	FrameArc
A8BF	PaintArc
A8CØ	EraseArc
A8C1	InvertArc
A8C2	FillArc
A8C3	PtToAngle
A8C4	AngleFromSlope
A8C5	StdPoly
A8C6	FramePoly
A8C7	PaintPoly
A8C8	ErasePoly
A8C9	InvertPoly
A8CA	FillPoly
A8CB	OpenPoly
A8CC	ClosePgon ClosePoly
A8CD	KillPoly
A8CE	OffsetPoly
A8CF	PackBits
A8DØ	UnpackBits
A8D1	StdRgn
A8D2	FrameRgn
A8D3	PaintRgn
A8D4	EraseRgn
A8D5	InverRgn InvertRgn
A8D6	FillRgn
A8D8	NewRgn
A8D9	DisposRgn DisposeRgn
A8DA	OpenRgn
A8DB	CloseRgn
A8DC	CopyRgn
A8DD	SetEmptyRgn
A8DE	SetRecRgn SetRectRgn
A8DF	RectRgn
A8EØ	OfsetRgn OffsetRgn
A8E1	InsetRgn
A8E2	EmptyRgn
A8E3	EqualRgn
A8E4	SectRgn
A8E5	UnionRgn
A8E6	DiffRgn
A8E7	XorRgn
A8E8	PtInRgn
A8E9	RectInRgn
A8EA	SetStdProcs
A8EB	StdBits
A8EC	CopyBits
A8ED	StdTxMeas
A8EE	StdGetPic
A8EF	ScrollRect
A8FØ	StdPutPic
A8F1	StdComment
A8F2	PicComment
A8F3	OpenPicture
A8F4	ClosePicture
A8F5	KillPicture
A8F6	DrawPicture
A8F8	ScalePt
A8F9	MapPt
A8FA	MapRect
A8FB	MapRgn
A8FC	MapPoly
A8FE	InitFonts
A8FF	GetFName GetFontName
A9ØØ	GetFNum
A9Ø1	FMSwapFont SwapFont
A9Ø2	RealFont
A9Ø3	SetFontLock
A9Ø4	DrawGrowIcon
A9Ø5	DragGrayRgn
A9Ø6	NewString
A9Ø7	SetString
A9Ø8	ShowHide
A9Ø9	CalcVis
A9ØA	CalcVBehind CalcVisBehind
A9ØB	ClipAbove
A9ØC	PaintOne
A9ØD	PaintBehind
A9ØE	SaveOld
A9ØF	DrawNew
A91Ø	GetWMgrPort
A911	CheckUpdate
A912	InitWindows
A913	NewWindow
A914	DisposWindow DisposeWindow
A915	ShowWindow
A916	HideWindow
A917	GetWRefCon
A918	SetWRefCon
A919	GetWTitle
A91A	SetWTitle
A91B	MoveWindow

A91C	HiliteWindow
A91D	SizeWindow
A91E	TrackGoAway
A91F	SelectWindow
A92Ø	BringToFront
A921	SendBehind
A922	BeginUpdate
A923	EndUpdate
A924	FrontWindow
A925	DragWindow
A926	DragTheRgn
A927	InvalRgn
A928	InvalRect
A929	ValidRgn
A92A	ValidRect
A92B	GrowWindow
A92C	FindWindow
A92D	CloseWindow
A92E	SetWindowPic
A92F	GetWindowPic
A93Ø	InitMenus
A931	NewMenu
A932	DisposMenu
	DisposeMenu
A933	AppendMenu
A934	ClearMenuBar
A935	InsertMenu
A936	DeleteMenu
A937	DrawMenuBar
A938	HiliteMenu
A939	EnableItem
A93A	DisableItem
A93B	GetMenuBar
A93C	SetMenuBar
A93D	MenuSelect
A93E	MenuKey
A93F	GetItmIcon
	GetItemIcon
A94Ø	SetItmIcon
	SetItemIcon
A941	GetItmStyle
	GetItemStyle
A942	SetItmStyle
	SetItemStyle
A943	GetItmMark
	GetItemMark
A944	SetItmMark
	SetItemMark
A945	CheckItem
A946	GetItem
A947	SetItem
A948	CalcMenuSize
A949	GetMHandle
A94A	SetMFlash
	SetMenuFlash
A94B	PlotIcon
A94C	FlashMenuBar
A94D	AddResMenu
A94E	PinRect
A94F	DeltaPoint
A95Ø	CountMItems
A951	InsertResMenu
A954	NewControl
A955	DisposControl
	DisposeControl
A956	KillControls
A957	ShowControl
A958	HideControl
A959	MoveControl
A95A	GetCRefCon
A95B	SetCRefCon
A95C	SizeControl
A95D	HiliteControl
A95E	GetCTitle
A95F	SetCTitle
A96Ø	GetCtlValue
A961	GetMinCtl
	GetCtlMin
A962	GetMaxCtl
	GetCtlMax
A963	SetCtlValue
A964	SetMinCtl
	SetCtlMin
A965	SetMaxCtl
	SetCtlMax
A966	TestControl
A967	DragControl
A968	TrackControl
A969	DrawControls
A96A	GetCtlAction
A96B	SetCtlAction
A96C	FindControl
A96E	Dequeue
A96F	Enqueue
A97Ø	GetNextEvent
A971	EventAvail
A972	GetMouse
A973	StillDown
A974	Button
A975	TickCount
A976	GetKeys
A977	WaitMouseUp
A979	CouldDialog
A97A	FreeDialog
A97B	InitDialogs
A97C	GetNewDialog
A97D	NewDialog

A97E	SelIText
A97F	IsDialogEvent
A98Ø	DialogSelect
A981	DrawDialog
A982	CloseDialog
A983	DisposDialog
A985	Alert
A986	StopAlert
A987	NoteAlert
A988	CautionAlert
A989	CouldAlert
A98A	FreeAlert
A98B	ParamText
A98C	ErrorSound
A98D	GetDItem
A98E	SetDItem
A98F	SetIText
A99Ø	GetIText
A991	ModalDialog
A992	DetachResource
A993	SetResPurge
A994	CurResFile
A995	InitResources
A996	RsrcZoneInit
A997	OpenResFile
A998	UseResFile
A999	UpdateResFile
A99A	CloseResFile
A99B	SetResLoad
A99C	CountResources
A99D	GetIndResource
A99E	CountTypes
A99F	GetIndType
A9AØ	GetResource
A9A1	GetNamedResource
A9A2	LoadResource
A9A3	ReleaseResource
A9A4	HomeResFile
A9A5	SizeRsrc
	SizeResource
A9A6	GetResAttrs
A9A7	SetResAttrs
A9A8	GetResInfo
A9A9	SetResInfo
A9AA	ChangedResource
A9AB	AddResource
A9AC	AddReference
A9AD	RmveResource
A9AE	RmveReference
A9AF	ResError
A9BØ	WriteResource
A9B1	CreateResFile
A9B2	SystemEvent
A9B3	SystemClick
A9B4	SystemTask
A9B5	SystemMenu
A9B6	OpenDeskAcc
A9B7	CloseDeskAcc
A9B8	GetPattern
A9B9	GetCursor
A9BA	GetString
A9BB	GetIcon
A9BC	GetPicture
A9BD	GetNewWindow
A9BE	GetNewControl
A9BF	GetRMenu
	GetMenu
A9CØ	GetNewMBar
A9C1	UniqueID
A9C2	SysEdit
	SystemEdit
A9C6	Secs2Date
A9C7	Date2Secs
A9C8	SysBeep
A9C9	SysError
A9CB	TEGetText
A9CC	TEInit
A9CD	TEDispose
A9CE	TextBox
A9CF	TESetText
A9DØ	TECalText
A9D1	TESetSelect
A9D2	TENew
A9D3	TEUpdate
A9D4	TEClick
A9D5	TECopy
A9D6	TECut
A9D7	TEDelete
A9D8	TEActivate
A9D9	TEDeactivate
A9DA	TEIdle
A9DB	TEPaste
A9DC	TEKey
A9DD	TEScroll
A9DE	TEInsert
A9DF	TESetJust
A9EØ	Munger
A9E1	HandToHand
A9E2	PtrToXHand
A9E3	PtrToHand
A9E4	HandAndHand
A9E5	InitPack
A9E6	InitAllPacks
A9E7	PackØ
A9E8	Pack1

A9E9	Pack2	
	DIBadMount	(Ø)
	DILoad	(2)
	DIUnload	(4)
	DIFormat	(6)
	DIVerify	(8)
	DIZero	(1Ø)
A9EA	Pack3	
	SFPutFile	(1)
	SFGetFile	(2)
	SFPPutFile	(3)
	SFPGetFile	(4)
A9EB	Pack4	
A9EC	Pack5	
A9ED	Pack6	
	IUDateString	(Ø)
	IUTimeString	(2)
	IUMetric	(4)
	IUDGetIntl	(6)
	IUSetIntl	(8)
	IUMagString	(1Ø)
	IUMagIDString	(12)
	IUDatePString	(14)
	IUTimePString	(16)
A9EE	Pack7	
	NumToString	(Ø)
	StringToNum	(1)
A9EF	PtrAndHand	
A9FØ	LoadSeg	
A9F1	UnloadSeg	
A9F2	Launch	
A9F3	Chain	
A9F4	ExitToShell	
A9F5	GetAppParms	
A9F6	GetResFileAttrs	
A9F7	SetResFileAttrs	
A9F9	InfoScrap	
A9FA	UnlodeScrap	
	UnloadScrap	
A9FB	LodeScrap	
	LoadScrap	
A9FC	ZeroScrap	
A9FD	GetScrap	
A9FE	PutScrap	

MACINTOSH USER EDUCATION

The Structure of a Macintosh Application

/STRUCTURE/STRUCT

See Also: Macintosh User Interface Guidelines
Inside Macintosh: A Road Map
The Segment Loader: A Programmer's Guide
Putting Together a Macintosh Application

Modification History: First Draft (ROM 7) Caroline Rose 2/8/84

ABSTRACT

This manual describes the overall structure of a Macintosh application program, including its interface with the Finder.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the overall structure of a Macintosh application program, including its interface with the Finder. *** Right now it describes only the Finder interface; the rest will be filled in later. Eventually it will become part of a comprehensive manual describing the entire Toolbox and Operating System. ***

(hand)

> This information in this manual applies to version 7 of the Macintosh ROM and version 1.Ø of the Finder.

You should already be familiar with the following:

- The details of the User Interface Toolbox, the Macintosh Operating System, and the other routines that your application program may call. For a list of all the technical documentation that provides these details, see Inside Macintosh: A Road Map.

- The Finder, which is described in the Macintosh owner's guide.

This manual doesn't cover the steps necessary to create an application's resources or to compile, link, and execute the application program. These are discussed in the manual Putting Together a Macintosh Application.

The manual begins with sections that describe the Finder interface: signatures and file types, used for identification purposes; application resources that provide icon and file information to the Finder; and the mechanism that allows documents to be opened or printed from the Finder.

*** more to come ***

Finally, there's a glossary of terms used in this manual.

SIGNATURES AND FILE TYPES

Every application must have a unique signature by which the Finder can identify it. The signature can be any four-character sequence not being used for another application on any currently mounted volume (except that it can't be one of the standard resource types). To ensure uniqueness on all volumes, your application's signature must be assigned by Macintosh Technical Support.

Signatures work together with file types to enable the user to open or print a document (any file created by an application) from the Finder. When the application creates a file, it sets the file's creator and file type. Normally it sets the creator to its signature and the file type to a four-character sequence that identifies files of that type. When the user asks the Finder to open or print the file, the Finder starts up the application whose signature is the file's creator and passes the file type to the application along with other identifying information, such as the file name. (More information about this process is given below under "Opening and Printing Documents from the Finder".)

An application may create its own special type or types of files. Like signatures, file types must be assigned by Macintosh Technical Support to ensure uniqueness. When the user chooses Open from an application's File menu, the application will display (via the Standard File Package) the names of all files of a given type or types, regardless of which application created the files. Having a unique file type for your application's special files ensures that only the names of those files will be displayed for opening.

(hand)

> Signatures and file types may be strange, unreadable combinations of characters; they're never seen by end users of Macintosh.

Applications may also create existing types of files. There might, for example, be one that merges two MacWrite documents into a single document. In such cases, the application should use the same file type as the original application uses for those files. It should also specify the original application's signature as the file's creator; that way, when the user asks the Finder to open or print the file, the Finder will call on the original application to perform the operation. To learn the signatures and file types used by existing applications, check with Macintosh Technical Support.

Files that consist only of text--a stream of characters, with Return characters at the ends of paragraphs or short lines--should be given the file type 'TEXT'. This is the type that MacWrite gives to text-only files it creates, for example. If your application uses this file type, its files will be accepted by MacWrite and it in turn will accept MacWrite text-only files (likewise for any other application that deals with 'TEXT' files). Your application can give its own signature as the file's creator if it wants to be called to open or print the file when the user requests this from the Finder.

For files that aren't to be opened or printed from the Finder, as may be the case for certain data files created by the application, the signature should be set to '????' (and the file type to whatever is appropriate).

FINDER-RELATED RESOURCES

To establish the proper interface with the Finder, every application's resource file must specify the signature of the application along with data that provides version information. In addition, there may be resources that provide information about icons and files related to the application. All of these Finder-related resources are described below, followed by a comprehensive example and (for interested programmers) the exact formats of the resources.

Version Data

Your application's resource file must contain a special resource that has the signature of the application as its resource type. This resource is called the version data of the application. The version data is typically a string that gives the name, version number, and date of the application, but it can in fact be any data at all. The resource ID of the version data is Ø by convention.

As described in detail in Putting Together a Macintosh Application, part of the process of installing an application on the Macintosh is to set the creator of the file that contains the application. You set the creator to the application's signature, and the Finder copies the corresponding version data into a resource file named Desktop. (The Finder doesn't display this file on the Macintosh desktop, to ensure that the user won't tamper with it.)

(hand)

> Additional, related resources may be copied into the Desktop file; see "Bundles" below for more information. The Desktop file also contains folder resources, one for each folder on the volume.

Icons and File References

For each application, the Finder needs to know:

- The icon to be displayed for the application on the desktop, if different from the Finder's default icon for applications (see Figure 1).

- If the application creates any files, the icon to be displayed for each type of file it creates, if different from the Finder's default icon for documents.

- What files, if any, must accompany the application when it's transferred to another volume.

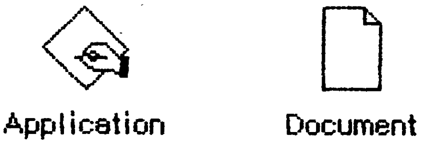


Figure 1. The Finder's Default Icons

The Finder learns this information from resources called file references in the application's resource file. Each file reference contains a file type and an ID number, called a local ID, that identifies the icon to be displayed for that type of file. (The local ID is mapped to an actual resource ID as described under "Bundles" below.) Any file reference may also include the name of a file that must accompany the application when it's transferred to another volume.

The file type for the application itself is 'APPL'. This is the file type in the file reference that designates the application's icon. You also specify it as the application's file type at the same time that you specify its creator--the first time you install the application on the Macintosh.

The ID number in a file reference corresponds not to a single icon but to an icon list in the application's resource file. The icon list consists of two icons: the actual icon to be displayed on the desktop, and a mask consisting of that icon's outline filled with black (see Figure 2). *** For existing types of files, there's currently no way to direct the Finder to use the original application's icon for that file type. ***

Figure 2. Icon and Mask

Bundles

A bundle in the application's resource file groups together all the Finder-related resources. It specifies the following:

- The application's signature and the resource ID of its version data
- A mapping between the local IDs for icon lists (as specified in file references) and the actual resource IDs of the icon lists in the resource file
- Local IDs for the file references themselves and a mapping to their actual resource IDs

The first time you install the application on the Macintosh, you set its "bundle bit", and the Finder copies the version data, bundle, icon lists, and file references from the application's resource file into the Desktop file. *** (The setting of the bundle bit will be covered in the next version of Putting Together a Macintosh Application.) *** If there are any resource ID conflicts between the icon lists and file references in the application's resource file and those in Desktop, the Finder will change those resource IDs in Desktop. The Finder does this same resource copying and ID conflict resolution when you transfer an application to another volume.

(hand)
The local IDs are needed only for use by the Finder.

An Example

Suppose you've written an application named SampWriter. The user can create a unique type of document from it, and you want a distinctive icon for both the application and its documents. The application's signature, as assigned by Macintosh Technical Support, is 'SAMP'; the file type assigned for its documents is 'SAMF'. Furthermore, a file named 'TgFil' should accompany the application when it's transferred to another volume. You would include the following resources in the application's resource file:

Resource	Resource ID	Contents
Version data with resource type 'SAMP'	Ø	The string 'SampWriter Version 1 -- 2/1/84'
Icon list	128	The icon for the application The icon's mask
Icon list	129	The icon for documents The icon's mask
File reference	128	File type 'APPL' Local ID Ø for the icon list
File reference	129	File type 'SAMF' Local ID 1 for the icon list File name 'TgFil'
Bundle	128	Signature 'SAMP' Resource ID Ø for the version data For icon lists, the mapping: local ID Ø --> resource ID 128 local ID 1 --> resource ID 129 For file references, the mapping: local ID Ø --> resource ID 128 local ID 1 --> resource ID 129

(hand)
See the manual Putting Together a Macintosh Application for information about how to include these resources in a resource file.

The file references in this example happen to have the same local IDs and resource IDs as the icon lists, but any of these numbers can be different. Different resource IDs can be given to the file references, and the local IDs specified in the mapping for file references can be whatever desired.

Formats of Finder-Related Resources

The resource type for an application's version data is the signature of the application, and the resource ID is Ø by convention. The resource data can be anything at all; typically it's a string giving the name, version number, and date of the application.

The resource type for an icon list is 'ICN#'. The resource data simply consists of the icons, 128 bytes each.

The resource type for a file reference is 'FREF'. The resource data has the format shown below.

Number of bytes	Contents
4 bytes	File type
2 bytes	Local ID for icon list
1 byte	Length of following file name in bytes; Ø if none
n bytes	Optional file name

The resource type for a bundle is 'BNDL'. The resource data has the format shown below. The format is more general than needed for Finder-related purposes because bundles will be used in other ways in the future.

Number of bytes	Contents
4 bytes	Signature of the application
2 bytes	Resource ID of version data
2 bytes	Number of resource types in bundle minus 1
For each resource type:	
4 bytes	Resource type
2 bytes	Number of resources of this type minus 1
For each resource:	
2 bytes	Local ID
2 bytes	Actual resource ID

A bundle used for establishing the Finder interface contains the two resource types 'ICN#' and 'FREF'.

OPENING AND PRINTING DOCUMENTS FROM THE FINDER

When the user selects a document and tries to open or print it from the Finder, the Finder starts up the application whose signature is the document file's creator. An application may be selected along with one or more documents for opening (but not printing); in this case, the Finder starts up that application. If the user selects more than one document for opening without selecting an application, the files must have the same creator. If more than one document is selected for printing, the Finder starts up the application whose signature is the first file's creator (that is, the first one selected if they were selected by Shift-clicking, or the top left one if they were selected by dragging a rectangle around them).

Any time the Finder starts up an application, it passes along information via the "Finder information handle" in the application parameter area (as described in the Segment Loader manual). Pascal programmers can call the Segment Loader procedure GetAppParms to get the Finder information handle. For example, if applParam is declared as type Handle, the call

```
GetAppParms(applName, applRefNum, applParam)
```

returns the Finder information handle in applParam. The Finder information has the following format:

Number of bytes	Contents
2 bytes	Ø if open, 1 if print
2 bytes	Number of files to open or print (Ø if none)
For each file:	
2 bytes	Volume reference number of volume containing the file
4 bytes	File type
1 byte	File's version number (typically Ø)
1 byte	Ignored
1 byte	Length of following file name in bytes
n bytes	Characters of file name (if n is even, add an extra byte)

The files are listed in order of the appearance of their icons on the desktop, from left to right and top to bottom. The file names don't include a volume prefix. An extra byte is added to any name of even length so that the entry for the next name will begin on a word boundary.

Every application that opens or prints documents should look at this information to determine what to do when the Finder starts it up. If the number of files is Ø, the application should start up with an untitled document on the desktop. If a file or files are specified for opening, it should start up with those documents on the desktop. If only one document can be open at a time but more than one file is specified, the application should open the first one and ignore the rest. If the application doesn't recognize a file's type (which can happen if the user selected the application along with another application's document), it may want to open the file anyway and check its internal structure to see if it's a compatible type. The response to an unacceptable type of file should be an alert box that shows the file name and says that the document can't be opened.

If a file or files are specified for printing, the application should print them in turn, preferably without doing its entire start-up sequence. For example, it may not be necessary to show the menu bar or a document window, and reading the desk scrap into memory is definitely not required. After successfully printing a document, the application should set the file type in the Finder information to Ø. Upon return from the application, the Finder will start up other applications as

necessary to print any remaining files whose type was not set to Ø.
*** The Finder doesn't currently do this, but it may in the future.
***

GLOSSARY

bundle: A resource that maps local IDs of resources to their actual resource IDs; used to provide mappings for file references and icon lists needed by the Finder.

Desktop file: A resource file in which the Finder stores folder resources and the version data, bundle, icons, and file references for each application on the volume.

file reference: A resource that provides the Finder with file and icon information about an application.

file type: A four-character sequence, specified when a file is created, the identifies the type of file.

icon list: A resource consisting of a list of icons.

local ID: A number that refers to an icon list or file reference in an application's resource file and is mapped to an actual resource ID by a bundle.

signature: A four-character sequence that uniquely identifies an application to the Finder.

version data: In an application's resource file, a resource that has the application's signature as its resource type; typically a string that gives the name, version number, and date of the application.

Apple

Apple Numerics Manual

Part I: The Standard Apple Numeric Environment

Part II: The 6502 Assembly-Language SANE Engine

Part III: The 68000 Assembly-Language SANE Engine

WORK-
BENCH

030-0247-A

The Apple Numerics Manual is included here for your convenience. It will not be a part of the final *Inside Macintosh* manual, but will be available separately.

Table of Contents

Chapter 1

Introduction

This manual describes the Standard Apple Numeric Environment (SANE). Apple supports SANE on several current products and plans to support SANE on future products. SANE gives you access to numeric facilities unavailable on almost any computer of the early 1980's—from microcomputers to extremely fast, extremely expensive supercomputers. The core features of SANE are not exclusive to Apple; rather they are taken from Draft 10.0 of Standard 754 for Binary Floating-Point Arithmetic [10] as proposed to the Institute of Electrical and Electronics Engineers (IEEE). Thus SANE is one of the first widely available products with the arithmetic capabilities destined to be found on the computers of the mid-1980's and beyond.

The IEEE Standard specifies standardized data types, arithmetic, and conversions, along with tools for handling limitations and exceptions, that are sufficient for numeric applications. SANE supports all requirements of the IEEE Standard. SANE goes beyond the specifications of the Standard by including a data type designed for accounting applications and by including several high-quality library functions for financial and scientific calculations.

IEEE arithmetic was specifically designed to provide advanced features for numerical analysts without imposing an extra burden on casual users. (This is an admirable but rarely attainable goal: text editors and word processors, for example, typically suffer increased complexity with added features, meaning more hurdles for the novice to clear before completing even the simplest tasks.) The independence of elementary and advanced features of the IEEE arithmetic was carried over to SANE.

Throughout this manual, references in brackets are to the annotated bibliography in Part I. Words printed in bold type are defined in the glossary in Part I.

Chapter 2

Data Types

SANE provides three **application** data types (single, double, and comp) and the **arithmetic** type (extended). Single, double, and extended store floating-point values and comp stores integral values.

The **extended** type is called the arithmetic type because, to make expression evaluation simpler and more accurate, SANE performs all arithmetic operations in extended precision and delivers arithmetic results to the extended type. **Single, double,** and **comp** can be thought of as space-saving storage types for the extended-precision arithmetic. (In this manual, we shall use the term *extended precision* to denote both the extended precision and the extended range of the extended type.)

All values representable in single, double, and comp (as well as 16-bit and 32-bit integers) can be represented exactly in extended. Thus values can be moved from any of these types to the extended type and back without any loss of information.

Choosing a Data Type

Typically, picking a data type requires that you determine the trade-offs between

- fixed- or floating-point form
- precision
- range
- memory usage
- speed.

The precision, range, and memory usage for each SANE data type are shown in Table 2-1. Effects of the data types on performance (speed) vary among the implementations of SANE. (See Chapter 4 for information on conversion problems relating to precision.)

Most accounting applications require a counting type that counts things (pennies, dollars, widgets) exactly. Accounting applications can be implemented by representing money values as integral numbers of cents or mils, which can be stored exactly in the storage format of the **comp** (for computational) type. The sum, difference, or product of any two comp values is exact if the magnitude of the result does not exceed $2^{63} - 1$ (that is, 9,223,372,036,854,775,807). This number is larger than the U.S. national debt expressed in Argentine pesos. In addition, comp values (such as the results of accounting computations) can be mixed with extended values in floating-point computations (such as compound interest).

Arithmetic with comp-type variables, like all SANE arithmetic, is done internally using extended-precision arithmetic. There is no loss of precision, as conversion from comp to extended is always exact. Space can be saved by storing numbers in the comp type, which is 20 percent shorter than extended. Non-accounting applications will normally be better served by the floating-point data formats.

Values Represented

The floating-point storage formats (single, double, and extended) provide binary encodings of a **sign** (+ or -), an **exponent**, and a **significand**. A represented number has the value

$\pm \text{significand} * 2^{\text{exponent}}$

where the significand has a single bit to the left of the binary point (that is, $0 \leq \text{significand} < 2$).

Range and Precision of SANE Types

This table describes the range and precision of the numeric data types supported by SANE. Decimal ranges are expressed as chopped two-digit decimal representations of the exact binary values.

Table 2-1. *SANE Types*

Type class	**Application**			**Arithmetic**
Type identifier	Single	Double	Comp	Extended
Size (bytes:bits)	4:32	8:64	8:64	10:80
Binary exponent range				
Minimum	-126	-1022	——	-16383
Maximum	127	1023	——	16383
Significand precision				
Bits	24	53	63	64
Decimal digits	7-8	15-16	18-19	19-20
Decimal range (approximate)				
Min negative	-3.4E+38	-1.7E+308	≅-9.2E18	-1.1E+4932
Max neg norm	-1.2E-38	-2.3E-308		-1.7E-4932
Max neg denorm†	-1.5E-45	-5.0E-324		-1.9E-4951
Min pos denorm†	1.5E-45	5.0E-324		1.9E-4951
Min pos norm	1.2E-38	2.3E-308		1.7E-4932
Max positive	3.4E+38	1.7E+308	≅9.2E18	1.1E+4932
Infinities†	Yes	Yes	No	Yes
NaNs†	Yes	Yes	Yes	Yes

†**Denorms (denormalized numbers)**, **NaNs (Not-a-Number)**, and **infinities** are defined in Chapter 7.

Usually numbers are stored in a **normalized** form, to afford maximum precision for a given significand width. Maximum precision is achieved if the high order bit in the significand is 1 (that is, 1 ≤ significand < 2).

Example

In Single, the largest representable number has

significand	=	$2 - 2^{-23}$
	=	$1.11111111111111111111111_2$
exponent	=	127
value	=	$(2 - 2^{-23}) * 2^{127}$
	≅	$3.403 * 10^{38}$

the smallest representable positive normalized number has

significand	=	1
	=	$1.00000000000000000000000_2$
exponent	=	-126
value	=	$1 * 2^{-126}$
	≅	$1.175 * 10^{-38}$

and the smallest representable positive denormalized number (see Chapter 7) has

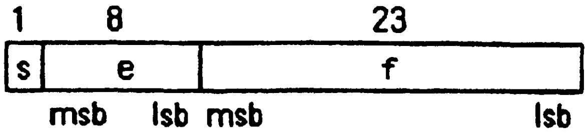
significand	=	2^{-23}
	=	$0.00000000000000000000001_2$
exponent	=	-126
value	=	$2^{-23} * 2^{-126}$
	≅	$1.401 * 10^{-45}$

Formats

This section shows the formats of the four SANE numeric data types. These are pictorial representations and may not reflect the actual byte order in any particular implementation.

Single

A 32-bit single format number is divided into three fields as shown below.



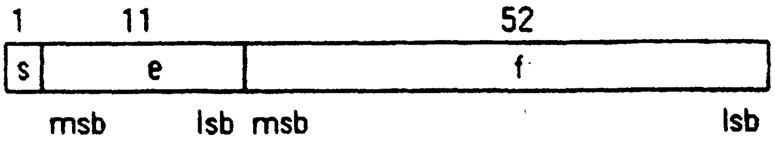
The value v of the number is determined by these fields as follows:

If $0 < e < 255$,	then $v = (-1)^s * 2^{(e-127)} * (1.f)$.
If $e = 0$ and $f \neq 0$,	then $v = (-1)^s * 2^{(-126)} * (0.f)$.
If $e = 0$ and $f = 0$,	then $v = (-1)^s * 0$.
If $e = 255$ and $f = 0$,	then $v = (-1)^s * \infty$.
If $e = 255$ and $f \neq 0$,	then v is a NaN.

See Chapter 7 for information on the contents of the f field for NaNs.

Double

A 64-bit double format number is divided into three fields as shown below.

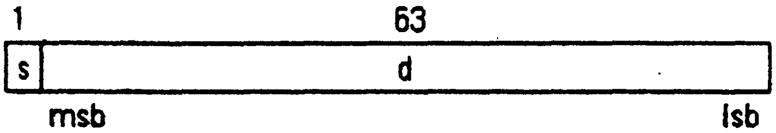


The value v of the number is determined by these fields as follows:

If $0 < e < 2047$,	then $v = (-1)^s * 2^{(e-1023)} * (1.f)$.
If $e = 0$ and $f \neq 0$,	then $v = (-1)^s * 2^{(-1022)} * (0.f)$.
If $e = 0$ and $f = 0$,	then $v = (-1)^s * 0$.
If $e = 2047$ and $f = 0$,	then $v = (-1)^s * \infty$.
If $e = 2047$ and $f \neq 0$,	then v is a NaN.

Comp

A 64-bit comp format number is divided into two fields as shown below.

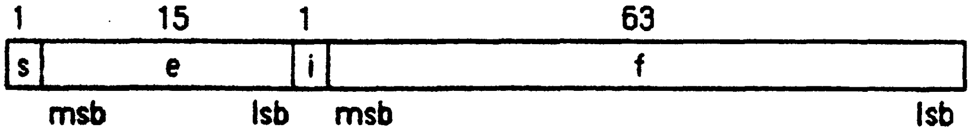


The value v of the number is determined by these fields as follows:

If s = 1 and d = 0,	then v is the unique comp NaN.
Otherwise,	v is the two's-complement value of the 64-bit representation.

Extended

An 80-bit extended format number is divided into four fields as shown below.



The value v of the number is determined by these fields as follows:

If 0 < = e < 32767,	then $v = (-1)^s * 2^{(e-16383)} * (i.f)$.
If e = 32767 and f = 0,	then $v = (-1)^s * \infty$, regardless of i.
If e = 32767 and f ≠ 0,	then v is a NaN, regardless of i.

Chapter 3

Arithmetic Operations

SANE provides these basic arithmetic operations for the SANE data types:

- add
- subtract
- multiply
- divide
- square root
- remainder
- round to integral value

(See Chapters 9 and 10 for auxiliary operations and higher-level functions supported by SANE.)

All the basic arithmetic operations produce the best possible result: the mathematically exact result coerced to the precision and range of the extended type. The coercions honor the user-selectable rounding direction and handle all exceptions according to the requirements of the IEEE Standard (see Chapter 8).

Remainder

Generally, remainder (and mod) functions are defined by the expression

$x \text{ rem } y = x - y * n$

where n is some integral approximation to the quotient x/y. This expression can be found even in the conventional integer-division algorithm:

```
                  n         (integral quotient approximation)
(divisor)     y ) x         (dividend)
                  y * n
              x - y * n     (remainder)
```

SANE supports the remainder function specified in the IEEE Standard:

> When $y \neq 0$, the remainder $r = x \text{ rem } y$ is defined regardless of the rounding direction by the mathematical relation $r = x - y * n$, where n is the integral value nearest the exact value x/y; whenever $|n - x/y| = 1/2$, n is even. The remainder is always exact. If $r = 0$, its sign is that of x.

Example 1

Find 5 rem 3 . Here $x = 5$ and $y = 3$. Since $1 < 5/3 < 2$ and since $5/3 = 1.66666...$ is closer to 2 than to 1, n is taken to be 2, so

$5 \text{ rem } 3 = r = 5 - 3 * 2 = -1$

Example 2

Find 7.0 rem 0.4 . Since $17 < 7.0/0.4 < 18$ and since $7.0/0.4 = 17.5$ is equally close to both 17 and 18, n is taken to be the even quotient, 18. Hence,

$7.0 \text{ rem } 0.4 = r = 7.0 - 0.4 * 18 = -0.2$

The IEEE remainder function differs from other commonly used remainder and mod functions. It returns a remainder of the smallest possible magnitude, and it always returns an exact remainder. All the other remainder functions can be constructed from the IEEE remainder.

Round to Integral Value

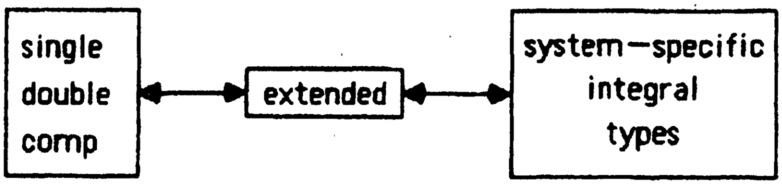
An input argument is rounded according to the current rounding direction to an integral value and delivered to the extended format. For example, 12345678.875 rounds to 12345678.0 or 12345679.0. (The rounding direction, which can be set by the user, is explained fully in Chapter 8.)

Note that, in each floating-point format, all values of sufficiently great magnitude are integral. For example, in single, numbers whose magnitudes are at least 2^{23} are integral.

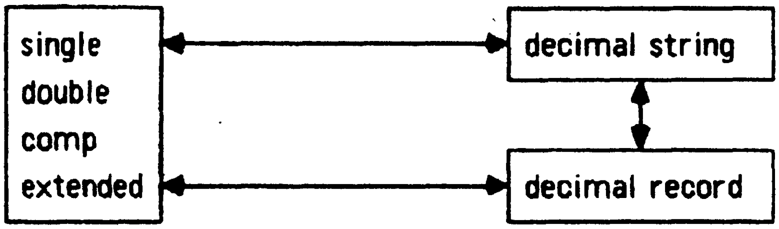
Chapter 4

Conversions

SANE provides conversions between the extended type and each of the other SANE types (single, double, and comp). A particular SANE implementation will provide conversions between extended and those numeric types supported in its particular larger environment. For example, a Pascal implementation will have conversions between extended and the Pascal integer type.



SANE implementations also provide either conversions between decimal strings and SANE types, or conversions between a decimal record type and SANE types, or both. Conversions between decimal records and decimal strings may be included too.



Conversions Between Extended and Single or Double

A conversion to extended is always exact. A conversion from extended to single or double moves a value to a storage type with less range and precision, and sets the overflow, underflow, and inexact exception flags as appropriate. (See Chapter 8 for a discussion of exception flags.)

Conversions to Comp and Other Integral Formats

Conversions to integral formats are done by first rounding to an integral value (honoring the current rounding direction) and then, if possible, delivering this value to the destination format. If the source operand of a conversion from extended to comp is a NaN, an infinity, or out-of-range for the comp format, then the result is the comp NaN and for infinities and values out-of-range, the invalid exception is signaled. If the source operand of a conversion to a system-specific integer type is a NaN, infinity, or out-of-range for that format, then invalid is signaled (unless the type has an appropriate representation for the exceptional result). NaNs, infinities, and out-of-range values are stored in a two's-complement integer format as the extreme negative value (for example, in the 16-bit integer format, as -32768).

Note that IEEE rounding into integral formats differs from most common rounding functions on halfway cases. With the default rounding direction (to nearest), conversions to comp or to a system-specific integer type will round 0.5 to 0, 1.5 to 2, 2.5 to 2, and 3.5 to 4, rounding to even on halfway cases. (Rounding is discussed in detail in Chapter 8.)

Formats

This section shows the formats of the four SANE numeric data types. These are pictorial representations and may not reflect the actual byte order in any particular implementation.

Conversions Between Binary and Decimal

The IEEE Standard for binary floating-point arithmetic specifies the set of numerical values representable within each floating-point format. It is important to recognize that binary storage formats can exactly represent the fractional part of decimal numbers in only a few cases; in all other cases, the representation will be approximate. For example, 0.5_{10}, or $1/2_{10}$, can be represented exactly as 0.1_2. On the other hand, 0.1_{10}, or $1/10_{10}$, is a repeating fraction in binary: $0.00011001100...._2$. Its closest representation in single is $0.000110011001100110011001101_2$, which is closer to 0.10000000149_{10} than to 0.10000000000_{10}.

As binary storage formats generally provide only close approximations to decimal values, it is important that conversions between the two types be as accurate as possible. Given a rounding direction, for every decimal value there is a best (correctly rounded) binary value for each binary format. Conversely, for any rounding direction, each binary value has a corresponding best decimal representation for a given decimal format. Ideally, binary-decimal conversions should obtain this best value to reduce accumulated errors. Conversion routines in SANE implementations meet or exceed the stringent error bounds specified by the IEEE Standard. This means that although in extreme cases the conversions do not deliver the correctly rounded result, the result delivered is very nearly as good as the correctly rounded result. (See the IEEE Standard [10] for a more detailed description of error bounds.)

Conversions From Decimal Strings to SANE Types

Routines may be provided to convert numeric decimal strings to the SANE data types. These routines are provided for the convenience of those who do not wish to write their own parsers and scanners. Examples of acceptable input are

```
123    123.4E-12    -123.    .456    3e9    -0

-INF    Inf    NAN(12)    -NaN()    nan
```

The 12 in NAN(12) is a NaN code (see Chapter 8).

The accepted syntax is formally defined, using Backus-Naur form, in Table 4-1.

Table 4-1. *Syntax for String Conversions*

<table>
<tr><td><decimal number></td><td>::=</td><td>[{space | tab}] <left decimal></td></tr>
<tr><td><left decimal></td><td>::=</td><td>[+|-] <unsigned decimal></td></tr>
<tr><td><unsigned decimal></td><td>::=</td><td><finite number> | <infinity> |
<NAN></td></tr>
<tr><td><finite number></td><td>::=</td><td><significand> [<exponent>]</td></tr>
<tr><td><significand></td><td>::=</td><td><integer> | <mixed></td></tr>
<tr><td><integer></td><td>::=</td><td><digits> [.]</td></tr>
<tr><td><digits></td><td>::=</td><td>{0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}</td></tr>
<tr><td><mixed></td><td>::=</td><td>[<digits>] . <digits></td></tr>
<tr><td><exponent></td><td>::=</td><td>E [+|-] <digits></td></tr>
<tr><td><infinity></td><td>::=</td><td>INF</td></tr>
<tr><td><NAN></td><td>::=</td><td>NAN[([<digits>])]</td></tr>
</table>

(Note: In the table, square brackets enclose optional items, braces (curly brackets) enclose elements to be repeated at least once, and vertical bars separate alternative elements; letters that appear literally, like the 'E' marking the exponent field, may be either upper or lower case.)

Decform Records and Conversions From SANE Types to Decimal Strings

Each conversion to a decimal string is controlled by a **decform** record, which contains two fields:

style	-- 16– bit word (Pascal 0 . . 1)
digits	-- 16– bit integer

Style equals 0 for floating and 1 for fixed. Following the Lisa Pascal convention, the value of style is stored in the high– order byte on 68000 systems. Following the Apple II Pascal convention, the value of style is stored as a 16– bit integer on 6502 systems. Digits gives the number of significant digits for the floating style and the number of digits to the right of the decimal point for the fixed style (digits may be negative if the style is fixed). Decimal strings resulting from these conversions are always acceptable input for conversions from decimal strings to SANE types. Further formatting details are implementation-dependent.

The Decimal Record Type

The decimal record type provides an intermediate unpacked form for programmers who wish to do their own parsing of numeric input or formatting of numeric output. The decimal record format has three fields:

sgn	—	16-bit word (Pascal 0 . . 1)
exp	—	16-bit integer
sig	—	string (maximum length is implementation-dependent)

The value represented is

$(-1)^{sgn} * sig * 10^{exp}$

when the length of sig is 18 or less. (Some implementations allow additional information in characters past the eighteenth.) Following the Lisa Pascal convention, the value of sgn is stored in the high– order byte on 68000 systems. Following the Apple II Pascal convention, the value of sgn is stored as a 16– bit integer on 6502 systems. Sig contains the internal decimal significand: the initial byte of sig (sig[0]) is the length byte, which gives the length of the ASCII string that is left-justified in the remaining bytes. Sgn is 0 for + and 1 for – . For example, if sgn = 1, exp = – 3, and sig = '85' (sig[0] = 2, not shown), then the number represented is – 0.085.

Conversions From Decimal Records to SANE Types

Conversions from the decimal record type handle any sig digit-string of length 18 or less (with an implicit decimal point at the right end). The following special cases apply:

- If sig[1] = '0' (zero), the decimal record is converted to zero. For example, a decimal record with sig = '0913' is converted to zero.
- If sig[1] = 'N', the decimal record is converted to a NaN. Except when the destination is of type comp (which has a unique NaN), the succeeding characters of sig are interpreted as a hex representation of the result significand: if fewer than 4 characters follow 'N' then they are right justified in the high-order 15 bits of the field f illustrated in the section "Formats" in Chapter 2; if 4 or more characters follow 'N' then they are left justified in the result's significand; if no characters, or only '0's, follow N, then the result NaN code is set to nanzero = 15 (hex).
- If sig[1] = 'I' and the destination is not of comp type, the decimal record is converted to an infinity. If the destination is of comp type, the decimal record is converted to a NaN and invalid is signaled.
- Other special cases produce undefined results.

Conversions From SANE Types to Decimal Records

Each conversion to a decimal record is controlled by a decform record (see above). All implementations allow at least 18 digits to be returned in sig. The implied decimal point is at the right end of sig, with exp set accordingly.

Zeroes, infinities, and NaNs are converted to decimal records with sig parts '0' (zero), 'I', and strings beginning with 'N', while exp is undefined. For NaNs, 'N' may be followed by a hex representation of the input significand. The third and forth hex digits following 'N' give the NaN code. For example, 'N0021000000000000' has NaN code 21 (hex).

When the number of digits specified in a decform record exceeds an implementation maximum (which is at least 18), the result is undefined.

A number may be too large to represent in a chosen fixed style. For instance, if the implementation's maximum length for sig is 18, then 10^{15} (which requires 16 digits to the left of the point in fixed-style representations) is too large for a fixed-style representation specifying more than 2 digits to the right of the point. If a number is too large for a chosen fixed style, then (depending on the SANE implementation) one of two results is returned: an implementation may return the most significant digits of the number in sig and set exp so that the decimal record contains a valid floating-style representation of the number; alternatively, an implementation may simply set the string sig to '?'. Note that in any implementation, the test

(-exp < > decform digits) or (sig[1] = '?')

determines whether a nonzero finite number is too large for the chosen fixed style.

Conversions Between Decimal Formats

SANE implementations may provide conversions between decimal strings and decimal records.

Conversion From Decimal Strings to Decimal Records

This conversion routine is intended as an aid to programmers doing their own scanning. The routine is designed for use either with fixed strings or with strings being received (interactively) character by character. An integer argument on input gives the starting index into the string, and on output is one greater than the index of the last character in the numeric substring just parsed. The longest possible numeric substring is parsed; if no numeric substring is recognized, then the index remains unchanged. Also, a Boolean argument is returned indicating that the input string, beginning at the input index, is a valid numeric string or a valid prefix of a numeric string. The accepted input for this conversion is the same as for conversions from decimal strings to SANE types (see above). Output is the same as for conversions from SANE types to decimal records (also above).

Examples

Input String	Index		Output Value	Valid-Prefix
	In	Out		
12	1	3	12	TRUE
12E	1	3	12	TRUE
12E-	1	3	12	TRUE
12E-3	1	6	12E-3	TRUE
12E-x	1	3	12	FALSE
12E-3x	1	6	12E-3	FALSE
x12E-3	2	7	12E-3	TRUE
IN	1	1	UNDEFINED	TRUE
INF	1	4	INF	TRUE

Conversion From Decimal Records to Decimal Strings

This conversion is controlled by the style field of a decform record (the digits field is ignored). Input is the same as for conversions from decimal records to SANE types, and output formatting is the same as for conversions from SANE types to decimal strings. This conversion, actually a formatting operation, is exact and signals no exception.

Chapter 5

Expression Evaluation

SANE arithmetic is extended-based. Arithmetic operations produce results with extended precision and extended range. For minimal loss of accuracy in more complicated computations, you should use extended temporary variables to store intermediate results.

Using Extended Temporaries

A programmer may use extended temporaries deliberately to reduce the effects of round-off error, overflow, and underflow on the final result.

Example 1

To compute the single-precision sum

$S = X[1] * Y[1] + X[2] * Y[2] + \ldots + X[N] * Y[N]$

where X and Y are arrays of type single, declare an extended variable XS and compute

```
XS := 0;
FOR I := 1 TO N DO
  XS := XS + X[I] * Y[I];       {extended-precision arithmetic }
S := XS;                        {deliver final result to single.}
```

Even when input and output values have only single precision, it may be very difficult to prove that single-precision arithmetic is sufficient for a given calculation. Using extended-precision arithmetic for intermediate values will often improve the accuracy of single-precision results more than virtuoso algorithms would. Likewise, using the extra range of the extended type for intermediate results may yield correct final results in the single type in cases when using the single type for intermediate results would cause an overflow or a catastrophic underflow. Extended-precision arithmetic is also useful for calculations involving double or comp variables: see Example 2.

Extended-Precision Expression Evaluation

High-level languages that support SANE evaluate all non-integer numeric expressions to extended precision, regardless of the types of the operands.

Example 2

If C is of type comp and MAXCOMP is the largest comp value, then the right-hand side of

C := (MAXCOMP + MAXCOMP) / 2

would be evaluated in extended to the exact result C = MAXCOMP, even though the intermediate result MAXCOMP + MAXCOMP exceeds the largest possible comp value.

Extended-Precision Expression Evaluation and the IEEE Standard

The IEEE Standard encourages extended-precision expression evaluation. Extended evaluation will on rare occasions produce results slightly different from those produced by other IEEE implementations that lack extended evaluation. Thus in a single-only IEEE implementation,

```
z := x + y
```

with x, y, and z all single, is evaluated in one single-precision operation, with at most one rounding error. Under extended evaluation, however, the addition x + y is performed in extended, then the result is coerced to the single precision of z, with at most two rounding errors. Both implementations conform to the standard.

The effect of a single- or double-only IEEE implementation can be obtained under SANE with rounding precision control, as described in Chapter 8.

Chapter 6

Comparisons

SANE supports the usual numeric comparisons: less, less-or-equal, greater, greater-or-equal, equal, and not-equal. For real numbers, these comparisons behave according to the familiar ordering of real numbers.

SANE comparisons handle NaNs and infinities as well as real numbers. The usual trichotomy for real numbers is extended so that, for any SANE values a and b, exactly one of the following is true:

a < b
a > b
a = b
a and b are unordered

Determination is made by the following rule: If x or y is a NaN, then x and y are unordered; otherwise, x and y are less, equal, or greater according to the ordering of the real numbers, with the understanding that +0 = -0 = real 0, and
-∞ < each real number < +∞.

(Note that a NaN always compares unordered—even with itself.)

The meaning of high-level language relational operators is a natural extension of their old meaning based on trichotomy. For example, the Pascal or BASIC expression x < = y is true if x is less than y or if x equals y, and is false if x is greater than y or if x and y are unordered. Note that the SANE not-equal relation means less, greater, or unordered—even if not-equal is written < >, as in Pascal and BASIC. High-level languages supporting SANE supplement the usual comparison operators with a function that takes two numeric arguments and returns the appropriate relation (less, equal, greater, or unordered). This function can be used to determine whether two numeric representations satisfy any combination of less, equal, greater, and unordered.

A high-level language comparison that involves a relational operator containing less or greater, but not unordered, signals invalid if the operands are unordered (that is, if either operand is a NaN). For example, in Pascal or BASIC if x or y is a quiet NaN then x < y, x < = y, x > = y, and x > y signal invalid, but x = y and x < > y (recall that < > contains unordered) do not. If a comparison operand is a signaling NaN, then invalid is always signaled, just as in arithmetic operations.

Chapter 7

Infinities, NaNs, and Denormalized Numbers

In addition to the normalized numbers supported by most floating-point packages, IEEE floating-point arithmetic also supports infinities, NaNs, and denormalized numbers.

Infinities

An **infinity** is a special bit pattern that can arise in one of two ways:

1. When a SANE operation should produce an exact mathematical infinity (such as 1/0), the result is an infinity bit pattern.
2. When a SANE operation attempts to produce a number with magnitude too great for the number's intended floating-point storage format, the result may (depending on the current rounding direction) be an infinity bit pattern.

These bit patterns (as well as NaNs, introduced next) are recognized in subsequent operations and produce predictable results. The infinities, one positive (+ INF) and one negative (-INF), generally behave as suggested by the theory of limits. For example, 1 added to + INF yields + INF; -1 divided by + 0 yields -INF; and 1 divided by -INF yields -0.

Each of the storage types single, double, and extended provides unique representations for + INF and -INF. The comp type has no representations for infinities. (An infinity moved to the comp type becomes the comp NaN.)

NaNs

When a SANE operation cannot produce a meaningful result, the operation delivers a special bit pattern called a **NaN** (Not-a-Number). For example, 0 divided by 0, +INF added to -INF, and sqrt(-1) yield NaNs. A NaN can occur in any of the SANE storage types (single, double, extended, and comp); but, generally, system-specific integer types have no representation for NaNs. NaNs propagate through arithmetic operations. Thus, the result of 3.0 added to a NaN is the same NaN (that is, has the same NaN code). If two operands of an operation are NaNs, the result is one of the NaNs. NaNs are of two kinds: **quiet NaNs**, the usual kind produced by floating-point operations; and **signaling NaNs.**

When a signaling NaN is encountered as an operand of an arithmetic operation, the invalid-operation exception is signaled and, if no halt occurs, a quiet NaN is the delivered result. Signaling NaNs could be used for uninitialized variables. They are not created by any SANE operations. The most significant bit of the field f illustrated in the section "Formats" in Chapter 2 is clear for quiet NaNs and set for signaling NaNs. The unique comp NaN generally behaves like a quiet NaN.

A NaN in a floating-point format has an associated NaN code that indicates the NaN's origin. (These codes are listed in Table 7-1). The NaN code is the 8th through 15th most significant bits of the field f illustrated in Chapter 2. The comp NaN is unique and has no NaN code.

Table 7-1. *SANE NaN Codes*

Name	Dec	Hex	Meaning
NANSQRT	1	$01	Invalid square root, such as sqrt(-1)
NANADD	2	$02	Invalid addition, such as (+INF) - (+INF)
NANDIV	4	$04	Invalid division, such as 0/0
NANMUL	8	$08	Invalid multiplication, such as 0 * INF
NANREM	9	$09	Invalid remainder or mod such as x rem 0
NANASCBIN	17	$11	Attempt to convert invalid ASCII string
NANCOMP	20	$14	Result of converting comp NaN to floating
NANZERO	21	$15	Attempt to create a NaN with a zero code
NANTRIG	33	$21	Invalid argument to trig routine
NANINVTRIG	34	$22	Invalid argument to inverse trig routine
NANLOG	36	$24	Invalid argument to log routine
NANPOWER	37	$25	Invalid argument to x^i or x^y routine
NANFINAN	38	$26	Invalid argument to financial function
NANINIT	255	$FF	Uninitialized storage (signaling NaN)

Denormalized Numbers

Whenever possible, floating-point numbers are **normalized** to keep the leading significand bit 1: this maximizes the resolution of the storage type. When a number is too small for a normalized representation, leading zeros are placed in the significand to produce a **denormalized** representation. A denormalized number is a nonzero number that is not normalized and whose exponent is the minimum exponent for the storage type.

Example

The sequence below shows how a single-precision value becomes progressively denormalized as it is repeatedly divided by 2, with rounding to nearest. This process is called **gradual underflow.**

A_0 $= 1.100\ 1100\ 1100\ 1100\ 1100\ 1101 * 2^{-126} \cong 0.1_{10} * 2^{-122}$

$A_1 = A_0/2$ $= 0.110\ 0110\ 0110\ 0110\ 0110\ 0110 * 2^{-126}$ (underflow)

$A_2 = A_1/2$ $= 0.011\ 0011\ 0011\ 0011\ 0011\ 0011 * 2^{-126}$

$A_3 = A_2/2$ $= 0.001\ 1001\ 1001\ 1001\ 1001\ 1010 * 2^{-126}$ (underflow)

. . . .

$A_{22} = A_{21}/2$ $= 0.000\ 0000\ 0000\ 0000\ 0000\ 0011 * 2^{-126}$

$A_{23} = A_{22}/2$ $= 0.000\ 0000\ 0000\ 0000\ 0000\ 0010 * 2^{-126}$ (underflow)

$A_{24} = A_{23}/2$ $= 0.000\ 0000\ 0000\ 0000\ 0000\ 0001 * 2^{-126}$

$A_{25} = A_{24}/2$ $= 0.0$ (underflow)

$A_1 \ldots A_{24}$ are denormalized; A_{24} is the smallest positive denormalized number in single type.

Why Denormalized Numbers?

The use of denormalized numbers makes statements like the following true for all real numbers:

x - y = 0 if and only if x = y

This statement is not true for most older systems of computer arithmetic, because they exclude denormalized numbers. For these systems, the smallest nonzero number is a normalized number with the minimum exponent; when the result of an operation is smaller than this smallest normalized number, the system delivers zero as the result. For such **flush-to-zero** systems, if x ≠ y but x - y is smaller than the smallest normalized number, then x - y = 0. IEEE systems do not have this defect, as x - y, although denormalized, is not zero.

(A few old programs that rely on premature flushing to zero may require modification to work properly under IEEE arithmetic. For example, some programs may test x - y = 0 to determine whether x is very near y.)

Inquiries: Class and Sign

Each valid representation in a SANE data type (single, double, comp, or extended) belongs to exactly one of these classes:

- signaling NaN
- quiet NaN
- infinite
- zero
- normalized
- denormalized

SANE implementations provide the user with the facility to determine easily the class and sign of any valid representation.

Chapter 8

Environmental Control

Environmental controls include the rounding direction, rounding precision, exception flags, and halt settings.

Rounding Direction

The available rounding directions are

- to-nearest
- upward
- downward
- toward-zero

The rounding direction affects all conversions and arithmetic operations except comparison and remainder. Except for conversions between binary and decimal (described in Chapter 4), all operations are computed as if with infinite precision and range and then rounded to the destination format according to the current rounding direction. The rounding direction may be interrogated and set by the user.

The default rounding direction is to-nearest. In this direction the representable value nearest to the infinitely precise result is delivered; if the two nearest representable values are equally near, the one with least significant bit zero is delivered. Hence, halfway cases round to even when the destination is the comp or a system-specific integer type, and when the round-to-integer operation is used. If the magnitude of the infinitely precise result exceeds the format's largest value (by at least one half unit in the last place), then the corresponding signed infinity is delivered.

The other rounding directions are upward, downward, and toward-zero. When rounding upward, the result is the format's value (possibly INF) closest to and no less than the infinitely precise result. When rounding downward, the result is the format's value (possibly -INF) closest to and no greater than the infinitely precise result. When rounding toward zero, the result is the format's value closest to and no greater in magnitude than the infinitely precise result. To truncate a number to an integral value, use toward-zero rounding either with conversion into an integer format or with the round-to-integer operation.

Rounding Precision

Normally, SANE arithmetic computations produce results to extended precision and range. To facilitate simulations of arithmetic systems that are not extended-based, the IEEE Standard requires that the user be able to set the rounding precision to single or double. If the SANE user sets rounding precision to single (or double) then all arithmetic operations produce results that are correctly rounded and that overflow or underflow as if the destination were single (or double), even though results are typically delivered to extended formats. Conversions to double and extended formats are affected if rounding precision is set to single, and conversions to extended formats are affected if rounding precision is set to double; conversions to decimal, comp, and system-specific integer types are not affected by the rounding precision. Rounding precision can be interrogated as well as set.

Setting rounding precision to single or double does not significantly enhance performance, and in some SANE implementations may hinder performance.

Exception Flags and Halts

SANE supports five exception flags with corresponding halt settings:

- invalid-operation (or invalid, for short)
- underflow
- overflow

- divide-by-zero
- inexact

These exceptions are signaled when detected; and, if the corresponding halt is enabled, the SANE engine will jump to a user-specified location. (A high-level language need not pass on to its user the facility to set this location, but may halt the user's program). The user's program can examine or set individual exception flags and halts, and can save and get the entire environment (rounding direction, rounding precision, exception flags, and halt settings). Further details of the halt (trap) mechanism are SANE implementation-specific.

Exceptions

The *invalid-operation* exception is signaled if an operand is invalid for the operation to be performed. The result is a quiet NaN, provided the destination format is single, double, extended, or comp. The invalid conditions are these:

- (addition or subtraction) magnitude subtraction of infinities, for example, (+INF) + (-INF);
- (multiplication) 0 * INF;
- (division) 0/0 or INF/INF;
- (remainder) x rem y, where y is zero or x is infinite;
- (square root) if the operand is less than zero;
- (conversion) to the comp format or to a system-specific integer format when excessive magnitude, infinity, or NaN precludes a faithful representation in that format (see Chapter 4 for details);
- (comparison) via predicates involving "<" or ">", but not "unordered", when at least one operand is a NaN;
- any operation on a signaling NaN except sign manipulations (negate, absolute-value, and copy-sign) and class and sign inquiries.

The *underflow* exception is signaled when a floating-point result is both tiny and inexact (and therefore is perhaps significantly less accurate than it would be if the exponent range were unbounded). A result is considered tiny if, before rounding, its magnitude is smaller than its format's smallest positive normalized number.

The *divide-by-zero* exception is signaled when a finite nonzero number is divided by zero. It is also signaled, in the more general case, when an operation on finite operands produces an exact infinite result: for example, logb (0) returns -INF and signals divide-by-zero. (Overflow, rather than divide-by-zero, flags the production of an inexact infinite result.)

The *overflow* exception is signaled when a floating-point destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result were the exponent range unbounded. (Invalid, rather than overflow, flags the production of an out-of-range value for an integral destination format.)

The *inexact* exception is signaled if the rounded result of an operation is not identical to the mathematical (exact) result. Thus, inexact is always signaled in conjunction with overflow or underflow. Valid operations on infinities are always exact and therefore signal no exceptions. Invalid operations on infinities are described above.

Managing Environmental Settings

The environmental settings in SANE are global and can be explicitly changed by the user. Thus all routines inherit these settings and are capable of changing them. Often special precautions must be taken because a routine requires certain environmental settings, or because a routine's settings are not intended to propagate outside the routine. (Examples in this section use Pascal syntax. SANE implementations in other languages have operations with equivalent functionality.)

Example 1

The subroutine below uses to-nearest rounding while not affecting its caller's rounding direction.

```
                - - -
var r: RoundDir;        { local storage for rounding direction }
                - - -
begin
    r := GetRound;          { save caller's rounding direction }
    SetRound (TONEAREST);   {          set to-nearest rounding }
        - - -
    SetRound (r)        {  restore caller's rounding direction }
end;
```

Note that, if the subroutine is to be reentrant, then storage for the caller's environment must be local.

SANE implementations may provide two efficient functions for managing the environment as a whole: procedure-entry and procedure-exit.

The procedure-entry function returns the current environment (for saving in local storage) and sets the default environment: rounding direction to-nearest, rounding precision extended, and exception flags and halts clear.

Example 2

The following subroutine runs under the default environment while not affecting its caller's environment.

```
        - - -
var e: Environment;            { local storage for environment }
        - - -
begin
    e := ProcEntry;            { save caller's envirnoment and }
                               { set default environment       }
        - - -
    SetEnvironment (e)         { restore caller's environment  }
end;
```

The procedure-exit function facilitates writing subroutines that appear to their callers to be atomic operations (such as addition, sqrt, and others). Atomic operations pass extra information back to their callers by signaling exceptions; however, they hide internal exceptions, which may be irrelevant or misleading. Procedure-exit, which takes a saved environment as arguments, does the following:

1. It temporarily saves the exception flags (raised by the subroutine).
2. It restores the environment received as argument.
3. It signals the temporarily saved exceptions. (Note that if enabled, halts could occur at this step.)

Thus exceptions signaled between procedure-entry and procedure-exit are hidden from the calling program unless the exceptions remain raised when the procedure-exit function is called.

Example 3

The following function signals underflow if its result is denormal, and overflow if its result is infinite, but hides spurious exceptions occurring from internal computations.

```
function compres: double;
           - - -
var e: Environ;                { local storage for environment }
    c: NumClass;               { for class inquiry             }
           - - -
begin {compres}
    e := ProcEntry;            { save caller's environment and }
                               { set default environment -     }
                               { now halts disabled            }
           - - -
    compres := result;                 { result to be returned }
    c := ClassD (result);              { class inquiry         }
    ClearXcps;          { clear possibly spurious exceptions   }

{ now raise specified exception flags:                         }
     if c = INFINITE then SetXcp (OVERFLOW, TRUE)
     else if c = DENORMALNUM then SetXcp (UNDERFLOW, TRUE);
     ProcExit (e)            { restore caller's environment,    }
                             { including any halt enables,  and }
                             { then signal exceptions from      }
                             { subroutine                       }
end {compres} ;
```

Chapter 9

Auxiliary Procedures

SANE includes a set of special routines that are recommended in an appendix to the IEEE Standard as aids to programming:

- negate
- absolute value
- copy-sign
- next-after
- scalb
- logb

Sign Manipulation

The sign manipulation operations change only the sign of their argument. Negate reverses the sign of its argument. Absolute-value makes the sign of its argument positive. Copy-sign takes two arguments and copies the sign of one of its arguments onto the sign of its other argument.

These operations are treated as nonarithmetic in the sense that they raise no exceptions: even signaling NaNs do not signal the invalid-operation exception.

Next-After Functions

The floating-point values representable in single, double, and extended formats constitute a finite set of real numbers. The next-after functions (one for each of these formats) generate the next representable neighbor in the proper format, given an initial value x and another value y indicating a direction from the initial value.

Each of the next-after functions takes two arguments, x and y:

nextsingle(x,y)	(x and y are single)
nextdouble(x,y)	(x and y are double)
nextextended(x,y)	(x and y are extended)

As elsewhere, the names of the functions may vary with the implementation.

Special Cases for Next-After Functions

If the initial value and the direction value are equal, then the result is the initial value.

If the initial value is finite but the next representable number is infinite, then overflow and inexact are signaled.

If the next representable number lies strictly between -M and +M, where M is the smallest positive normalized number for that format, and if the arguments are not equal, then underflow and inexact are signaled.

Binary Scale and Log Functions

The scalb and logb functions are provided for manipulating binary exponents.

Scalb efficiently scales a given number (x) by a given integer power (n) of 2, returning $x * 2^n$.

Logb returns the binary exponent of its input argument as a signed integral value. When the input argument is denormalized, the exponent is determined as if the input argument had first been normalized.

Special Cases for Logb

If x is infinite, logb(x) returns +INF.

If x = 0, logb(x) returns -INF and signals divide-by-zero.

Chapter 10

Elementary Functions

SANE provides a number of basic mathematical functions, including logarithms, exponentials, two important financial functions, trigonometric functions, and a random number generator. These functions are computed using the basic SANE arithmetic heretofore described.

All of the elementary functions, except the random number generator, handle NaNs, overflow, and underflow appropriately. All signal inexact appropriately, except that the general exponential and the financial functions may conservatively signal inexact when determining exactness would be too costly.

Logarithm Functions

SANE provides three logarithm functions:

base-2 logarithm	:	$\log_2(x)$
base-e or natural logarithm	:	ln(x)
base-e logarithm of 1 plus argument	:	ln1(x)

Ln1(x) accurately computes ln(1 + x). If the input argument x is small, such as an interest rate, the computation of ln1(x) is more accurate than the straightforward computation of ln(1 + x) by adding x to 1 and taking the natural logarithm of the result.

Special Cases for Logarithm Functions

If x = +INF, then $\log_2(x)$, ln(x), and ln1(x) return +INF. No exception is signaled.

If x = 0, then $\log_2(x)$ and ln(x) return -INF and signal divide-by-zero. Similarly, if x = -1, then ln1(x) returns -INF and signals divide-by-zero.

If x < 0, then $\log_2(x)$ and ln(x) return a NaN and signal invalid. Similarly, if x < -1, then ln1(x) returns a NaN and signals invalid.

Exponential Functions

SANE provides five exponential functions:

base-2 exponential	:	2^x
base-e or natural exponential	:	e^x
base-e exponential minus 1	:	exp1(x)
integer exponential	:	x^i (i of integer type)
general exponential	:	x^y

Exp1(x) accurately computes $e^x - 1$. If the input argument x is small, such as an interest rate, then the computation of exp1(x) is more accurate than the straightforward computation of $e^x - 1$ by exponentiation and subtraction.

Special Cases for 2^x, e^x, exp1(x)

If x = +INF, then 2^x, e^x, and exp1(x) return +INF. No exception is signaled.

If x = -INF, then 2^x and e^x return 0; and exp1(x) returns -1. No exception is signaled.

Special Cases for x^i

If the integer exponent i equals 0 and x is not a NaN, then x^i returns 1. Note that with the integer exponential, $x^0 = 1$ even if x is zero or infinite.

If x is +0 and i is negative, then x^i returns +INF and signals divide-by-zero.

If x is -0 and i is negative, then x^i returns +INF if i is even, or -INF if i is odd: both cases signal divide-by-zero.

Special Cases for x^y

If x is +0 and y is negative, then the general exponential x^y returns +INF and signals divide-by-zero.

If x is -0 and y is integral and negative, then x^y returns +INF if y is even, or -INF if y is odd: both cases signal divide-by-zero.

The general exponential x^y returns a NaN and signals invalid if

- both x and y equal 0;
- x is infinite and y equals 0;
- x = 1 and y is infinite; or
- x is -0 or less than 0 and y is nonintegral.

Financial Functions

SANE provides two functions, compound and annuity, that can be used to solve various financial, or time-value-of-money, problems.

Compound

The compound function computes

$\text{compound}(r,n) = (1 + r)^n$

where r is the interest rate and n is the number (perhaps nonintegral) of periods. When the rate r is small, compound gives a more accurate computation than does the straightforward computation of $(1 + r)^n$ by addition and exponentiation.

Compound is directly applicable to computation of present and future values:

$$PV = FV * (1 + r)^{(-n)} = \frac{PV}{\text{compound}(r,n)}$$

$$FV = PV * (1 + r)^n = PV * \text{compound}(r,n)$$

Annuity

The annuity function computes

$$\text{annuity}(r,n) = \frac{1 - (1 + r)^{(-n)}}{r}$$

where r is the interest rate and n is the number of periods. Annuity is more accurate than the straightforward computation of the expression above using basic arithmetic operations and exponentiation. The annuity function is directly applicable to the computation of present and future values of ordinary annuities:

$$PV = PMT * \frac{1 - (1 + r)^{(-n)}}{r}$$

$$= PMT * \text{annuity}(r,n)$$

$$FV = PMT * \frac{(1 + r)^{n} - 1}{r}$$

$$= PMT * (1 + r)^{n} * \frac{1 - (1 + r)^{(-n)}}{r}$$

$$= PMT * \text{compound}(r,n) * \text{annuity}(r,n)$$

where PMT is the amount of one periodic payment.

Special Cases for compound(r,n)

If r = 0 and n is infinite, or if r = -1, then compound(r,n) returns a NaN and signals invalid.

If r = -1 and n < 0, then compound(r,n) returns +INF and signals divide-by-zero.

Special Cases for annuity(r,n)

If r = 0, then annuity(r,n) computes the sum of 1 + 1 + ... + 1 over n periods, and therefore returns the value n and signals no exceptions (the value n corresponds to the limit as r approaches 0).

If r < -1, then annuity(r,n) returns a NaN and signals invalid.

If r = -1 and n > 0, then annuity(r,n) returns -INF and signals divide-by-zero.

Trigonometric Functions

SANE provides the basic trigonometric functions

cosine	:	cos(x)
sine	:	sin(x)
tangent	:	tan(x)
arctangent	:	arctan(x)

The arguments for cosine, sine, and tangent and the results of arctangent are expressed in radians. The cosine, sine, and tangent functions use an argument reduction based on the remainder function (see Chapter 3) and the nearest extended-precision approximation of pi/2. Thus the cosine, sine, and tangent functions have periods slightly different from their mathematical counterparts and diverge from their counterparts when their arguments become large. Number results from arctangent lie between -pi/2 and pi/2.

The remaining trigonometric functions can be easily and efficiently computed from the elementary functions provided (see Appendix A).

Special Cases for sin(x), cos(x)

If x is infinite, then cos(x) and sin(x) return a NaN and signal invalid.

Special Cases for tan(x)

If x is the nearest extended approximation to ±pi/2, then tan(x) returns ±INF.

If x is infinite, then tan(x) returns a NaN and signals invalid.

Special Case for arctan(x)

If x = ±INF, then arctan(x) returns the nearest extended approximation to ±pi/2.

Random Number Generator

SANE provides a pseudorandom number generator, random. Random has one argument, passed by address. A sequence of (pseudo) random integral values r in the range

$1 \le r \le 2^{31} - 2$

can be generated by initializing an extended variable r to an integral value (the seed) in the above range and making repeated calls random(r); each call delivers in r the next random number in the sequence.

Random uses the iteration formula

$r \leftarrow (7^5 * r) \bmod (2^{31} - 1)$.

If seed values of r are nonintegral or outside the range

$1 \le r \le 2^{31} - 2$

then results are unspecified.

A pseudorandom rectangular distribution on the interval (0,1) can be obtained by dividing the results from random by

$2^{31} - 1$ = scalb (31,1) - 1 .

Appendix A

Other Elementary Functions

The Standard Apple Numeric Environment (SANE) provides the several transcendental functions; from these, you can construct other high-quality functions, as shown by the pseudocode examples below. These robust, accurate functions are based on algorithms developed by Professor William Kahan of the University of California at Berkeley.

All variables in the pseudocode below are extended. The constant C is 2^{-33} = scalb (-33,1). C is chosen to be nearly the largest value for which 1 - C^2 rounds to 1.

Exception Handling

Unlike the SANE elementary functions, these functions do not provide complete handling of special cases and exceptions. The most troublesome exceptions can be correctly handled if you

- begin each function with a call to procedure-entry;
- clear the spurious exceptions indicated in the comment;
- end each function with a call to procedure-exit (see Chapter 8).

Functions

Secant

```
sec(x) ← 1 / cos(x)
```

CoSecant

```
csc(x) ← 1 / sin(x)
```

CoTangent

```
cot(x) ← 1 / tan(x)
```

ArcSine

```
y ← |x|
If y ≥ 0.3 then begin
                  y ← Atan (x/sqrt ((1 - x) * (1 + x)))
                  {spurious divide-by-zero may arise}
                end
else if y ≥ C then y ← Atan (x / (sqrt (1 - x^2))
              else y ← x
arcsin(x) ← y
```

ArcCosine

```
arccos(x) ← 2 * Atan (sqrt ((1 - x)/(1 + x)) )
{spurious divide-by-zero may arise}
```

Sinh

```
y ← |x|
If y ≥ C then begin
                y ← exp1(y)
                y ← 0.5 * (y + y/(1 + y))
              end
copy the sign of x onto y
sinh(x) ← y
```

Cosh

```
y ← exp(|x|)
cosh(x) ← 0.5 * y + 0.25 / (0.5 * y)
```

Tanh

```
y ← |x|
If y ≥ C then begin
               y ← exp1(-2 * y)
               y ← -y/(2 + y)
             end
copy the sign of x onto y
tanh(x) ← y
```

ArcSinh

```
y ← |x|
If y ≥ C then begin
               y ← ln1 (y + y / (1/y + sqrt(1 + (1/y)^2) ))
               {spurious underflow may arise}
             end
copy the sign of x onto y
asinh(x) ← y
```

ArcCosh

```
y ← |x|
acosh(x) ← ln1 ( (sqrt (y-1)) * (sqrt (y-1) + sqrt (y+1)) )
```

ArcTanh

```
y ← |x|
If y ≥ C then y ← ln1 (2 * y/(1 - y)) / 2
copy the sign of x onto y
atanh(x) ← y
```

Glossary

Application type: A data type used to store data for applications.

Arithmetic type: A data type used to hold results of calculations inside the computer. The SANE arithmetic type, extended, has greater range and precision than the application types, in order to improve the mathematical properties of the application types.

Binary floating-point number: A string of bits representing a sign, an exponent, and a significand. Its numerical value, if any, is the signed product of the significand and two raised to the power of its exponent.

Comp type: A 64-bit application data type for storing integral values of up to 18- or 19-decimal-digit precision. It is used for accounting applications, among others.

Decform record: A data type for specifying the formatting for decimal results (of conversions). It specifies fixed- or floating-point form and the number of digits.

Denormalized number, or **denorm:** A nonzero binary floating-point number that is not normalized (that is, whose significand has a leading bit of zero) and whose exponent is the minimum exponent for the number's storage type.

Double type: A 64-bit application data type for storing floating-point values of up to 15- or 16-decimal-digit precision. It is used for statistical and financial applications, among others.

Environmental settings: The rounding direction and rounding precision, plus the exception flags and their respective halts.

Exceptions: Special cases, specified by the IEEE Standard, in arithmetic operations. The exceptions are invalid, underflow, overflow, divide-by-zero, and inexact.

Exception flag: Each exception has a flag that can be set, cleared and tested. It is set when its respective exception occurs and stays set until explicitly cleared.

Exponent: The part of a binary floating-point number that indicates the power to which two is raised in determining the value of the number. The wider the exponent field in a numeric type, the greater range it will handle.

Extended type: An 80-bit arithmetic data type for storing floating-point values of up to 19- or 20-decimal-digit precision. SANE uses it to hold the results of arithmetic operations.

Flush-to-zero: A system that excludes denormalized numbers. Results smaller than the smallest normalized number are rounded to zero.

Gradual underflow: A system that includes denormalized numbers.

Halt: Each exception has a halt-enable that can be set or cleared. When an exception is signaled and the corresponding halt is enabled, the SANE engine will transfer control to the address in a halt vector. A high-level language need not pass an to its user the facility to set the halt vector, but may halt the user's program. Halts remain set until explicitly cleared.

Infinity: A special bit pattern produced when a floating-point operation attempts to produce a number greater in magnitude than the largest representable number in a given format. Infinities are signed.

Integer types: System types for integral values. Integer types typically use 16- or 32-bit two's complement integers. Integer types are not SANE types but are available to SANE users.

Integral value: A value in a SANE type that is exactly equal to a mathematical integer: ..., -2, -1, 0, 1, 2, ...

NaN (Not a Number): A special bit pattern produced when a floating-point operation cannot produce a meaningful result (for example, 0/0 produces a NaN). NaNs can also be used for uninitialized storage. NaNs propagate through arithmetic operations.

Normalized number: A binary floating-point number in which all significand bits are significant: that is, the leading bit of the significand is 1.

Quiet NaN: A NaN that propagates through arithmetic operations without signaling an exception (and hence without halting a program).

Rounding direction: When the result of an arithmetic operation cannot be represented exactly in a SANE type, the computer must decide how to round the result. Under SANE, the computer resolves rounding decisions in one of four directions, chosen by the user: tonearest (the default), upward, downward, and towardzero.

Sign bit: The bit of a single, double, comp, or extended number that indicates the number's sign: 0 indicates a positive number; 1, a negative number.

Signaling NaN: A NaN that signals an invalid exception when the NaN is an operand of an arithmetic operation. If no halt occurs, a quiet NaN is produced for the result. No SANE operation creates signaling NaNs.

Significand: The part of a binary floating-point number that indicates where the number falls between two successive powers of two. The wider the significand field in a numeric type, the more resolution it will have.

Single type: A 32-bit application data type for storing floating-point values of up to 7- or 8-decimal-digit precision. It is used for engineering applications, among others.

Annotated Bibliography

[1] *Apple III Pascal Programmer's Manual*, Volume 2. "Appendix A: The TRANSCEND and REALMODES Units" and "Appendix E: Floating-Point Arithmetic." Cupertino, Calif.: Apple Computer, Inc., 1981.

These appendixes describe the implementation of single-precision arithmetic in Apple III Pascal, which was based upon Draft 8.0 of the proposed Standard.

[2] *Apple III Pascal Numerics Manual: A Guide to Using the Apple III Pascal SANE and Elems Units*. Cupertino, Calif.: Apple Computer, Inc., 1983.

This manual describes the Apple III Pascal implementation of the Standard Apple Numeric Environment (SANE) through procedure calls to the SANE and Elems units. This was Apple's first full implementation of IEEE arithmetic.

[3] *Apple Pascal Numerics Manual: A Guide to Using the Apple Pascal SANE and Elems Units*. Cupertino, Calif.: Apple Computer, Inc., 1983.

This manual, generalized from [2], describes the Apple II and Apple III Pascal implementation of the Standard Apple Numeric Environment (SANE) through procedure calls to the SANE and Elems units.

[4] Cody, W. J. "Analysis of Proposals for the Floating-Point Standard." *IEEE Computer* Vol. 14, No. 3 (March 1981).

This paper compares the several contending proposals presented to the Working Group.

[5] Coonen, Jerome T. "Accurate, Yet Economical Binary-Decimal Conversions." To appear in *ACM Transactions on Mathematical Software*.

[6] Coonen, Jerome T. "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic." *IEEE Computer* Vol. 13, No. 1 (January 1980).

This paper is a forerunner to the work on the draft Standard.

[7] Coonen, Jerome T. "Underflow and the Denormalized Numbers." *IEEE Computer* Vol. 14, No. 3 (March 1981).

[8] Demmel, James. "The Effects of Underflow on Numerical Computation." To appear in *SIAM Journal on Scientific and Statistical Computing*.

These papers examine one of the major features of the proposed Standard, gradual underflow, and show how problems of bounded exponent range can be handled through the use of denormalized values.

[9] Fateman, Richard J. "High-Level Language Implications of the Proposed IEEE Floating-Point Standard." *ACM Transactions on Programming Languages and Systems* Vol. 4, No. 2 (April 1982).

This paper describes the significance to high-level languages, especially FORTRAN, of various features of the IEEE proposed Standard.

[10] Floating-Point Working Group 754 of the Microprocessor Standards Committee, IEEE Computer Society. "A Standard for Binary Floating-Point Arithmetic." Proposed to IEEE, 345 East 47th Street, New York, NY 10017.

The implementation of SANE is based upon the final draft of this Standard, submitted December 1982.

[11] Floating-Point Working Group 754 of the Microprocessor Standards Committee, IEEE Computer Society. "A Proposed Standard for Binary Floating-Point Arithmetic." *IEEE Computer* Vol. 14, No. 3 (March 1981).

This is Draft 8.0 of the proposed Standard, which was offered for public comment. The current Draft 10.0 is substantially simpler than this draft; for instance, warning mode and projective mode have been eliminated, and the definition of underflow has changed. However, the intent of the Standard is basically the same, and this paper includes some excellent introductory comments by David Stevenson, Chairman of the Floating-Point Working Group.

[12] Hough, D. "Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic." *IEEE Computer* Vol. 14, No. 3 (March 1981).

This paper is an excellent introduction to the floating-point environment provided by the proposed Standard, showing how it facilitates the implementation of robust numerical computations.

[13] Kahan, W. "Interval Arithmetic Options in the Proposed IEEE Floating-Point Arithmetic Standard." In *Interval Mathematics 1980*, edited by K. E. L. Nickel. New York: Academic Press, 1980.

This paper shows how the proposed Standard facilitates interval arithmetic.

[14] Kahan, W., and Jerome T. Coonen. "The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments." In *The Relationship between Numerical Computation and Programming Languages*, edited by J. K. Reid. New York: North Holland, 1982.

This paper describes high-level language issues relating to the proposed IEEE Standard, including expression evaluation and environment handling.

Part II of the Apple Numerics Manual, which deals with implementation of SANE on the 6502 microprocessor, has been omitted from this edition.

rounding precision, bits 0060			RR
0000	—	extended	
0020	—	double	
0040	—	single	
0060	—	UNDEFINED	

halts enabled, bits 001F			
0001	—	invalid	I
0002	—	underflow	U
0004	—	overflow	O
0008	—	division-by-zero	D
0010	—	inexact	X

Bits 8000 and 0080 are undefined.

Note that the default environment is represented by the integer value zero.

Index

MACINTOSH USER EDUCATION

Index to Technical Documentation /TOOLBOX/INDEX

See Also: Inside Macintosh: A Road Map
Macintosh User Interface Guidelines
Macintosh Memory Management: An Introduction
Programming Macintosh Applications in Assembly Language
The Resource Manager: A Programmer's Guide
QuickDraw: A Programmer's Guide
The Font Manager: A Programmer's Guide
The Toolbox Event Manager: A Programmer's Guide
The Window Manager: A Programmer's Guide
The Control Manager: A Programmer's Guide
The Menu Manager: A Programmer's Guide
TextEdit: 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
Toolbox Utilities: A Programmer's Guide
Macintosh Packages: A Programmer's Guide
The Memory Manager: A Programer's Guide
The Segment Loader: A Programmer's Guide
The Operating System Event Manager: A Programmer's Guide
The File Manager: A Programmer's Guide
Printing from Macintosh Applications
The Device Manager: A Programmer's Guide
The Disk Driver: A Programmer's Guide
The Sound Driver: A Programmer's Guide
The Serial Drivers: A Programmer's Guide
The AppleTalk Manager: A Programmer's Guide
The Vertical Retrace Manager: A Programmer's Guide
The System Error Handler: A Programmer's Guide
The Operating System Utilities: A Programmer's Guide
The Structure of a Macintosh Application

Modification History:	First Draft	Caroline Rose	8/5/83
	Updated	Caroline Rose	10/5/83, 1/9/84, 6/5/84, 8/13/84, 10/5/84, 12/6/84, 2/1/85

ABSTRACT

This is an index to all the documentation listed under "See Also:" above, as of 2/1/85.

INDEX

The page numbers are preceded by a two-letter designation of which manual the information is in:

AL	Programming Macintosh Applications in Assembly Language	1/22/85
AM	The AppleTalk Manager: A Programmer's Guide	1/31/85
CM	The Control Manager: A Programmer's Guide	5/30/84
DD	The Disk Driver: A Programmer's Guide	9/18/84
DL	The Dialog Manager: A Programmer's Guide	7/6/84
DS	The Desk Manager: A Programmer's Guide	8/22/84
DV	The Device Manager: A Programmer's Guide	6/15/84
EH	The System Error Handler: A Programmer's Guide	9/26/84
EM	The Toolbox Event Manager: A Programmer's Guide	11/19/84
FL	The File Manager: A Programmer's Guide	5/21/84
FM	The Font Manager: A Programmer's Guide	6/11/84
MI	Macintosh Memory Management: An Introduction	8/20/84
MM	The Memory Manager: A Programmer's Guide	10/9/84
MN	The Menu Manager: A Programmer's Guide	9/24/84
OE	The Operating System Event Manager: A Programmer's Guide	11/19/84
OU	The Operating System Utilities: A Programmer's Guide	1/11/85
PK	Macintosh Packages: A Programmer's Guide	5/7/84
PR	Printing from Macintosh Applications	6/15/84
PT	Putting Together a Macintosh Application	7/10/84
QD	QuickDraw: A Programmer's Guide	3/2/83
RD	Inside Macintosh: A Road Map	9/10/84
RM	The Resource Manager: A Programmer's Guide	11/28/84
SL	The Segment Loader: A Programmer's Guide	8/24/84
SM	The Scrap Manager: A Programmer's Guide	1/31/85
SN	The Sound Driver: A Programmer's Guide	11/15/84
SR	The Serial Drivers: A Programmer's Guide	9/28/84
ST	The Structure of a Macintosh Application	2/8/84
TE	TextEdit: A Programmer's Guide	1/14/85
TU	The Toolbox Utilities: A Programmer's Guide	11/13/84
UI	Macintosh User Interface Guidelines	11/30/84
VR	The Vertical Retrace Manager: A Programmer's Guide	6/15/84
WM	The Window Manager: A Programmer's Guide	5/30/84

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 Caroline Rose
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!

The Apple Certified Developer Program

If your primary business is developing software products for commercial markets, we strongly suggest that you investigate the Apple Certified Developer program, an agressive program designed to help independent software developers successfully get products to market in a timely manner. Apple Certified Developers receive regular mailings and can participate in a number of attractive programs.

For information about the Apple Certified Developer program contact:

Developer Relations
Apple Computer, Inc.
20525 Mariani Avenue, Mail Stop 27S
Cupertino, CA 95014
(408) 973-4897

Recommended System Configurations

We recommend the following configurations for Lisa Pascal/Macintosh cross-development and native Macintosh development.

Lisa Pascal/Macintosh Cross-Development

Macintosh 128K or 512K
Imagewriter plus Macintosh Accessory Kit
Macintosh External Disk Drive
3 1/2 inch blank disks
Macintosh XL
Macintosh XL 1/2 Mbyte RAM Card
Lisa Pascal Workshop
Software Supplement
Macintosh XL Imagewriter Accessory Kit

Native Macintosh Development

Macintosh 512K & Macintosh 128K
Imagewriter plus Macintosh Accessory Kit
Macintosh External Disk Drive
Software Supplement
3 1/2 inch blank disks

Macintosh Development Software List (March 1985)

Company	Phone	Product
Absoft	(313) 549-7111	MacFortran
Apple Computer, Inc.	(408) 996-1010	Macintosh 68000 Development System
		Macintosh Pascal
		Lisa Pascal Workshop
Consulair Corporation	(415) 851-3849	Mac C Compiler
		Mac C ToolKit
Creative Solutions	(301) 984-0262	MacForth
Expertelligence	(805) 969-7874	ExperLogo
FairCom	(314) 445-6833	C-Tree ISAM Package
Hippopotamus	(408) 730-2601	Hippo C
IQ Software	(817) 589-2000	CP/M for Macintosh
Kriya Systems, Inc.	(312) 822-0624	Neon (Forth/Smalltalk)
Mainstay	(818) 991-6540	Mac-Asm (Assembler)
Manx Software Systems	(201) 780-4004	Aztec C 68K
MegaMax, Inc.	(214) 987-4931	MegaMax C
Micro Focus, Inc.	(415) 856-4161	MacCobol
Microsoft	(206) 828-8080	Basic 2.0
Modula Corporation	(800) 545-4842	Modula-2
Softech Microsystems	(619) 451-1230	P-System
		Pascal/Fortran
Softworks, Ltd.	(312) 327-7666	Softworks C
Volition Systems	(619) 270-6800	Modula-2