INSIDE MACINTOSH

Operating System Utilities

Addison-Wesley Publishing Company

Reading, Massachusetts   Menlo Park, California   New York
Don Mills, Ontario   Wokingham, England   Amsterdam   Bonn
Sydney   Singapore   Tokyo   Madrid   San Juan
Paris   Seoul   Milan   Mexico City   Taipei

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

ISBN 0-201-62270-X
1 2 3 4 5 6 7 8 9-CRW-9897969594
First Printing, July 1994

Library of Congress Cataloging-in-Publication Data

Inside Macintosh. Operating System Utilities / [Apple Computer, Inc.]
p. cm.
Includes index.
ISBN 0-201-62270-X
1. Macintosh (Computer) 2. Operating systems (Computers)
3. Utilities (Computer programs) I. Apple Computer, Inc.
QA76.8.M3I5617 1994
005.4'469—dc20 94-18100
CIP

Contents

Chapter 2 System Error Handler 2-1

Chapter 3 Mathematical and Logical Utilities 3-1

Chapter 8 Trap Manager 8-1

Figures, Tables, and Listings

About This Book

This book, *Inside Macintosh: Operating System Utilities* describes the parts of the Macintosh Operating System that allow you to manage various low-level aspects of the system software. The chapters in this book and the information they contain are summarized here.

- "Gestalt Manager" describes how the Gestalt Manager works. This chapter also describes how you can make information about your own hardware or software available to other applications.
- "System Error Handler" explains what the Macintosh Operating System does when a system error is encountered. This chapter also describes how you can provide code that can help your application recover from a system error.
- "Mathematical and Logical Utilities" discusses how you can perform low-level logical manipulation of bits and bytes, save disk space by using simple compression and decompression routines, obtain a pseudorandom number, perform mathematical operations with two fixed-point data types supported directly by the Macintosh Operating System, and convert numeric variables of different types.
- "Date, Time, and Measurement Utilities" describes a set of utility routines that you can use to operate on dates and times. You can use these routines to get and change information about the current date, time, geographic location, time zone, and units of measurement.
- "Control Panel Extensions" describes how you can create a control panel extension to add a panel to an existing control panel.
- "Queue Utilities" describes how your application can directly add elements to and remove them from operating-system queues managed by the Macintosh Operating System. This chapter also describes how you can use the Queue Utilities to operate on queues that you create.
- "Parameter RAM" describes how your application can access and modify the information used by the system software at system startup time.
- "Trap Manager" describes how the Trap Manager works and then shows how you can use the Trap Manger to check for the availability of a system software routine. This chapter also describes how you can alter the behavior of a system software routine.
- "Start Manager" describes the system initialization and system startup process performed by the Macintosh computer. This chapter also describes how you can create a system extension.
- "Package Manager" lists all the standard Macintosh packages and it describes the routines that loads the packages into memory.

Additional information about the Macintosh Operating System can be found in other Inside Macintosh books. For information about processes and tasks, see *Inside Macintosh: Processes*. For information on how to allocate, release, or otherwise manipulate memory, see *Inside Macintosh: Memory*. For information about managing files and other objects in the file system, see *Inside Macintosh: Files*.

If you are new to programming the Macintosh computer, you should also read *Inside Macintosh: Overview* for an introduction to general concepts of Macintosh programming.

Format of a Typical Chapter

Almost all chapters in this book follow a standard structure. For example, the chapter "Queue Utilities" contains these sections:

- "About Queue Utilities." This section provides an overview of the features provided by the Queue Utilities.
- "Using Queue Utilities." This section describes the tasks you can accomplish using Queue Utilities. It describes how to use the most common routines, provides code samples, and supplies additional information.
- "Queue Utilities Reference." This section provides a complete reference for the Queue Utilities by describing the data structures, and routines it uses. Each routine description also follows a standard format, which presents the routine declaration followed by a description of every parameter of the routine. Some routine descriptions also give additional descriptive information, such as assembly-language information or result codes.
- "Summary of Queue Utilities." This section provides the Pascal and C interfaces for the constants, data structures, routines, and result codes associated with Queue Utilities. It also includes relevant assembly-language interface information.

Conventions Used in This Book

Inside Macintosh uses special conventions to present certain types of information.

Special Fonts

All code listings, reserved words, and names of actual data structures, fields, constants, parameters, and routines are shown in Courier (`this is Courier`).

Words that appear in **boldface** are key terms or concepts and are defined in the glossary.

Types of Notes

There are several types of notes used in this book.

Note

A note like this contains information that is interesting but not essential to an understanding of the main text. (An example appears on page 1-5.) u

IMPORTANT

A note like this contains information that is essential for an understanding of the main text. (An example appears on page 4-6.) s

s **WARNING**

Warnings like this indicate potential problems that you should be aware of as you design your application. Failure to heed these warnings could result in system crashes or loss of data. (An example appears on page 1-12.) s

Assembly-Language Information

Inside Macintosh provides information about the registers for specific routines in this format:

Registers on entry

A0	Contents of register A0 on entry

Registers on exit

D0	Contents of register D0 on exit

In the “Assembly-Language Summary” section at the end of each chapter, *Inside Macintosh* presents information about the fields of data structures in this format:

0	`what`	word	event code
2	`message`	long	event message
6	`when`	long	ticks since startup

The left column indicates the byte offset of the field from the beginning of the data structure. The second column shows the field name as defined in the MPW Pascal interface files; the third column indicates the size of that field. The fourth column provides a brief description of the use of the field. For a complete description of each field, see the discussion of the data structure in the reference section of the chapter.

In addition, *Inside Macintosh* presents information about the fields of a parameter block in this format:

Parameter block

↔	`inAndOut`	`Integer`	Input/output parameter.
←	`output1`	`Ptr`	Output parameter.
→	`input1`	`Ptr`	Input parameter.

The arrow in the far left column indicates whether the field is an input parameter, output parameter, or both. You must supply values for all input parameters and input/output parameters. The routine returns values in output parameters and input/output parameters.

The second column shows the field name as defined in the MPW Pascal interface files; the third column indicates the Pascal data type of that field. The fourth column provides a brief description of the use of the field. For a complete description of each field, see the discussion that follows the parameter block or the description of the parameter block in the reference section of the chapter.

The Development Environment

The system software routines described in this book are available using Pascal, C, or assembly-language interfaces. How you access these routines depends on the development environment you are using. When showing system software routines, this book uses the Pascal interface available with the Macintosh Programmer's Workshop (MPW).

All code listings in this book are shown in Pascal or assembly language. They show methods of using various routines and illustrate techniques for accomplishing particular tasks. All code listings have been compiled and in many cases tested. However, Apple Computer, Inc., does not intend for you to use these code samples in your application.

APDA is Apple's worldwide source for over three hundred development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the quarterly *APDA Tools Catalog* featuring all current versions of Apple and the most popular third-party development tools. Ordering is easy; there are no membership fees, and application forms are not required for most products. APDA offers convenient payment and shipping options including site licensing.

To order products or to request a complimentary copy of the *APDA Tools Catalog,* contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone:	800-282-2732 (United States) 800-637-0029 (Canada) 716-871-6555 (elsewhere in the world)
Fax:	716-871-6511
AppleLink:	APDA
America Online:	APDAorder
CompuServe:	76666,2405
Internet:	APDA@applelink.apple.com

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

For information on registering signatures, file types, and other technical information, contact

Macintosh Developer Technical Support
Apple Computer, Inc.
1 Infinite Loop, M / S 303-2T
Cupertino, CA 95014-6299

Gestalt Manager

Contents

This chapter describes how you can use the Gestalt Manager and other system software facilities to investigate the operating environment. You need to know about the operating environment if your application takes advantage of hardware (such as a floating-point unit) or software (such as Color QuickDraw) that is not available on all Macintosh computers. You can also use the Gestalt Manager to inform the Operating System that your software is present and to find out about other software registered with the Gestalt Manager.

The Gestalt Manager is available in system software versions 6.0.4 and later. The MPW software development system and some other development environments supply code that allows you to use the Gestalt Manager on earlier system software versions; check the documentation provided with your development system.

In system software versions earlier than 6.0.4, you can retrieve a limited description of the operating environment with the `SysEnvirons` function, also described in this chapter.

You need to read this chapter if you take advantage of specific hardware or software features that may not be present on all versions of the Macintosh, or if you wish to inform other software that your software is present in the operating environment.

This chapter describes how the Gestalt Manager works and then explains how you can

- determine whether the Gestalt Manager is available
- call the `Gestalt` function to investigate the operating environment
- make information about your own hardware or software available to other applications
- retrieve a limited description of the operating environment even if the Gestalt Manager is not available

About the Gestalt Manager

The Macintosh family of computers includes models that use a number of different processors, some accompanied by a floating-point unit (FPU) or memory management unit (MMU). Also, a single hardware configuration can have various versions of system software, drivers, and QuickDraw routines.

In general, applications should communicate with the system software and hardware through the available managers and device drivers. However, if your application takes advantage of hardware or software components that may not be present on all Macintosh computers, then you need some mechanism to determine whether those components are available.

The `Gestalt` function provides a simple, efficient way to determine the hardware and software configurations so your application can exploit as fully as possible whatever environment it is running in. When your application calls the `Gestalt` function, your application passes a **selector code** (or **selector**) as a parameter to specify the information it needs. Your application can call the `Gestalt` function to determine

- the version and features of QuickDraw
- the versions and features of various other managers and drivers
- the type of floating-point unit (FPU), if any
- the type of memory management unit (MMU), if any
- the amount of available RAM
- the amount of available virtual memory
- the version of the A / UX operating system, if it's running
- the type of keyboard
- the model of computer
- the version number of the System file
- the type of central processing unit (CPU)

Your application can use the information returned by `Gestalt` in various ways. It might branch to alternate code, for example, depending on the version of QuickDraw, or cancel an operation and present an alert box if a critical but optional hardware component is unavailable.

Associated with the `Gestalt` function are two other functions—one that allows an application to register new features with the Gestalt Manager and another that allows an application to change the function used by `Gestalt` to retrieve a particular piece of information. These two functions make it easy for your software to announce its presence to other applications. A debugger, for example, can register itself with the Gestalt Manager during system initialization; afterward, debugging code in an application under development can call `Gestalt` to verify that the special routines provided by the debugger are available on the local machine. In this way, the Gestalt Manager can act as a central clearinghouse for information on the available software and hardware features of the operating environment and enhance cooperation and awareness among third-party products.

Although the `Gestalt` function can provide much of the information your application needs, you might still need to call some special-purpose routines supplied by various parts of the system software. To determine the resolution of the main Macintosh screen, for example, you call the `ScreenRes` procedure, described in the book *Inside Macintosh: Imaging with QuickDraw*.

The `Gestalt` function has replaced both the `SysEnvirons` function and the `Environs` procedure. The `Gestalt` function is simpler to use and provides more information than either of those routines. Applications that use `SysEnvirons` execute correctly in system software versions 7.0 and later, in which `SysEnvirons` calls `Gestalt`.

The `SysEnvirons` function, introduced with the Macintosh SE and Macintosh II computers, fills in and returns a pointer to a **system environment record**, a data structure that describes some features of the operating environment. The `SysEnvirons` function cannot provide the detailed information supplied by `Gestalt`.

Like the `SysEnvirons` function, `Gestalt` can provide objective configuration information such as ROM version and size, but you should not infer the presence or absence of particular hardware or software features from that information. When you need to know whether a feature is present, you should request information about it directly by using the appropriate selector code. (“Getting Information About the Operating Environment” beginning on page 1-6, lists the Apple-defined selector codes for `Gestalt`.)

Using the Gestalt Manager

The Gestalt Manager includes three functions—`Gestalt`, `NewGestalt`, and `ReplaceGestalt`. You can use the `Gestalt` function to get information about hardware or software components available on the current machine. You can use `NewGestalt` to register new software modules (such as drivers and patches) with the Gestalt Manager. You can use `ReplaceGestalt` to replace the function associated with a particular selector code.

Note

Most applications do not need to use either `NewGestalt` or `ReplaceGestalt`. u

If the Gestalt Manager is not present, you can get a brief description of the operating environment by calling the `SysEnvirons` function.

Determining Whether the Gestalt Manager Is Available

Versions 3.2 and later of MPW provide glue routines that allow you to call the Gestalt Manager functions even if they’re not in ROM or in the System file (that is, if your application is running under a system software version earlier than 6.0.4). In assembly language, however, and possibly in other development environments, you must verify that the Gestalt Manager is available before you use it.

You can verify that the `Gestalt` function is available by calling the function `NGetTrapAddress`, specifying the trap number of `Gestalt`, and comparing the result with the address of the code that is executed when you invoke an unimplemented instruction. If `Gestalt` is available, you can safely assume that `NewGestalt` and `ReplaceGestalt` are also available. For efficiency, you might want to define a global Boolean variable that you can set at the beginning of your program. Listing 1-1 illustrates a test that sets the variable `gHasGestalt`.

Listing 1-1 Determining whether `Gestalt` is available

```
gHasGestalt := MySWRoutineAvailable(_Gestalt);
```

For a sample definition of the application-defined function `MySWRoutineAvailable`, see the chapter “Trap Manager” later in this book.

Getting Information About the Operating Environment

When your application needs information about a software or hardware feature, it calls the `Gestalt` function, which has this interface:

```
FUNCTION Gestalt (selector: OSType; VAR response: LongInt): OSErr;
```

The first parameter is a selector code, which specifies the kind of information your application needs. You can use any of the Apple-defined selector codes listed later in this section and described in more detail in the section “Constants” beginning on page 1-14. You can also define and register your own selector codes using the `NewGestalt` function (as described in “Adding a New Selector Code” beginning on page 1-10), and you can use selector codes defined and registered by other applications.

If `Gestalt` can determine the requested information, it returns that information in the `response` parameter and returns a result code of `noErr`. If Gestalt cannot obtain the information, it returns a result code indicating the cause of the error; in that case, the value of the `response` parameter is undefined. You should *always* check the result code returned by `Gestalt` to make sure that the `response` parameter contains meaningful information.

Listing 1-2 illustrates an application-defined function that retrieves the sound attributes of the current operating environment. The application-defined `MyGetSoundAttr` function checks the function result returned by `Gestalt` and passes any calls with a nonzero result code to an error-handling routine.

Listing 1-2 Calling `Gestalt` and checking its result code

```
FUNCTION MyGetSoundAttr: LongInt;
VAR
   myErr:   OSErr;
   myAttr:  LongInt;
BEGIN
   IF gHasGestalt THEN
   BEGIN
      myErr := Gestalt(gestaltSoundAttr, myAttr);
      IF myErr <> noErr THEN        {Gestalt failed}
         DoError(myErr)
   END
   ELSE
      myAttr := 0;                  {Gestalt not available}
   MyGetSoundAttr := myAttr;
END;
```

You get different kinds of information from `Gestalt` by passing selectors from two kinds of Apple-defined selector codes:

- **environmental selectors**, which return information your application can use to guide its actions
- **informational selectors**, which return information that cannot be used to determine whether a feature is available

It is particularly important that you understand the difference between environmental and informational selectors. The response returned by `Gestalt` when it is passed an informational selector is for your (or the user's) edification only; it should *never* be used by your application to determine whether a specific hardware or software feature is available. For example, you can use `Gestalt` to test for the version of the ROM installed on a particular machine. You can display this information to the user, but you should not infer from it anything about the actual software available. Routines you expect to be in ROM may actually be in RAM; hence, you cannot know that a routine usually found in ROM is not present simply because the ROM version predates the routine. Also, routines contained in ROM may have been patched by the system at startup time, in which case the system might not have the features you think it has on the basis of the reported ROM version. A Macintosh Plus with an old ROM, for example, could be running System 7. Similar remarks apply to other informational selectors, including ROM size, machine type, and System file version number.

To retrieve specific information about the hardware and software features available, you can use the following environmental selectors:

```
CONST
   gestaltAddressingModeAttr  = 'addr'; {addressing-mode attributes}
   gestaltAliasMgrAttr        = 'alis'; {Alias Manager attributes}
   gestaltAppleEventsAttr     = 'evnt'; {Apple events attributes}
   gestaltAppleTalkVersion    = 'atlk'; {old format AppleTalk version}
   gestaltATalkVersion        = 'atkv'; {new format AppleTalk version}
   gestaltAUXVersion          = 'a/ux'; {A/UX version, if present}
   gestaltCFMAttr             = 'cfrg'; {Code Fragment Manager attributes}
   gestaltCloseViewAttr       = 'BSDa'; {CloseView attributes}
   gestaltComponentMgr        = 'cpnt'; {Component Manager version}
   gestaltCompressionMgr      = 'icmp'; {Image Compression Manager version}
   gestaltConnMgrAttr         = 'conn'; {Connection Manager attributes}
   gestaltCRMAttr             = 'crm '; {Communication Resource Manager }
                                        { attributes}
   gestaltCTBVersion          = 'ctbv'; {Communication Toolbox version}
   gestaltDBAccessMgrAttr     = 'dbac'; {Data Access Manager attributes}
   gestaltDictionaryMgrAttr   = 'dict'; {Dictionary Manager attributes}
   gestaltDisplayMgrAttr      = 'dply'; {Display Manager atributes}
   gestaltDisplayMgrVers      = 'dplv'; {Display Manager version}
   gestaltDITLExtAttr         = 'ditl'; {Dialog Manager extensions}
   gestaltDragMgrAttr         = 'drag'; {Drag Manager attributes}
   gestaltEasyAccessAttr      = 'easy'; {Easy Access attributes}
   gestaltEditionMgrAttr      = 'edtn'; {Edition Manager attributes}
```

```
gestaltExtToolboxTable     = 'xttt'; {Toolbox trap dispatch table info}
gestaltFinderAttr          = 'fndr'; {Finder attributes}
gestaltFindFolderAttr      = 'fold'; {FindFolder attributes}
gestaltFirstSlotNumber     = 'slt1'; {first physical slot}
gestaltFontMgrAttr         = 'font'; {Font Manager attributes}
gestaltFPUType             = 'fpu '; {floating-point unit (FPU) type}
gestaltFSAttr              = 'fs  '; {file system attributes}
gestaltFXfrMgrAttr         = 'fxfr'; {File Transfer Manager attributes}
gestaltHelpMgrAttr         = 'help'; {Help Manager attributes}
gestaltIconUtilitiesAttr   = 'icon'; {Icon Utilities attributes}
gestaltKeyboardType        = 'kbd '; {keyboard type code}
gestaltLogicalPageSize     = 'pgsz'; {logical page size}
gestaltLogicalRAMSize      = 'lram'; {logical RAM size}
gestaltLowMemorySize       = 'lmem'; {size of low memory}
gestaltMiscAttr            = 'misc'; {miscellaneous attributes}
gestaltMixedModeVersion    = 'mixd'; {MixedMode version}
gestaltMMUType             = 'mmu '; {MMU type}
gestaltNativeCPUtype       = 'cput'; {native CPU type}
gestaltNotificationMgrAttr = 'nmgr'; {Notification Manager attributes}
gestaltNuBusConnectors     = 'sltc'; {NuBus connector bitmap}
gestaltNuBusSlotCount      = 'nubs'; {number of logical NuBus slots}
gestaltOSAttr              = 'os  '; {Operating System attributes}
gestaltOSTable             = 'ostt'; {base address of Operating System }
                                     { trap dispatch table}
gestaltParityAttr          = 'prty'; {parity attributes}
gestaltPCXAttr             = 'pcxg'; {PC exchange attributes}
gestaltPhysicalRAMSize     = 'ram '; {physical RAM size}
gestaltPopupAttr           = 'pop!'; {pop-up 'CDEF' attributes}
gestaltPowerMgrAttr        = 'powr'; {Power Manager attributes}
gestaltPPCToolboxAttr      = 'ppc '; {Program-to-Program Communications }
                                     { (PPC) Toolbox attributes}
gestaltProcessorType       = 'proc'; {microprocessor type code}
gestaltQuickdrawFeatures   = 'qdrw'; {QuickDraw features}
gestaltQuickdrawVersion    = 'qd  '; {QuickDraw version}
gestaltQuickTimeVersion    = 'qtim'; {QuickTime version}
gestaltRealTimeMgrAttr     = 'rtmr'; {Realtime Manager attributes}
gestaltResourceMgrAttr     = 'rsrc'; {Resource Manager attributes}
gestaltScrapMgrAttr        = 'scra'; {Scrap Manager attributes}
gestaltScriptCount         = 'scr#'; {number of active script systems}
gestaltScriptMgrVersion    = 'scri'; {Script Manager version}
gestaltSerialAttr          = 'ser '; {serial hardware attributes}
gestaltSlotAttr            = 'slot'; {slot attributes}
gestaltSoundAttr           = 'snd '; {sound attributes}
```

```
   gestaltSpeechAttr          = 'ttsc'; {Speech Manager attributes}
   gestaltStandardFileAttr    = 'stdf'; {Standard File attributes}}
   gestaltStdNBPAttr          = 'nlup'; {StandardNBP attributes}
   gestaltSysArchitecture     = 'sysa'; {Native System Architecture}
   gestaltTEAttr              = 'teat'; {TextEdit attributes}
   gestaltTermMgrAttr         = 'term'; {Terminal Manager attributes}
   gestaltTextEditVersion     = 'te  '; {TextEdit version code}
   gestaltThreadMgrAttr       = 'thds'; {Thread Manager attributes}
   gestaltTimeMgrVersion      = 'tmgr'; {Time Manager version code}
   gestaltToolboxTable        = 'tbtt'; {base address of Toolbox trap }
                                        { dispatch table}
   gestaltTranslationAttr     = 'xlat'; {Translation Manager attributes}
   gestaltTSMgrVersion        = 'tsmv'; {Text Services Manager version}
   gestaltVersion             = 'vers'; {Gestalt version}
   gestaltVMAttr              = 'vm  '; {virtual memory attributes}
```

The informational selectors are provided for your or the user's information only. You can display the information returned from these selectors, but you should never use this information as an indication of what hardware or software features may be available. You can use the following informational selectors:

```
CONST
   gestaltHardwareAttr        = 'hdwr'; {hardware attributes}
   gestaltMachineIcon         = 'micn'; {machine 'ICON'/'cicn' resource ID}
   gestaltMachineType         = 'mach'; {Macintosh model code}
   gestaltROMSize             = 'rom '; {ROM size}
   gestaltROMVersion          = 'romv'; {ROM version}
   gestaltSystemVersion       = 'sysv'; {System file version number}
```

For a description of the return values for these environmental and informational selectors, see the next section, "Interpreting Gestalt Responses," and the list of constants beginning on page 1-14.

Interpreting Gestalt Responses

The meaning of the value that `Gestalt` returns in the `response` parameter depends on the selector code with which it was called. For example, if you call `Gestalt` using the `gestaltTimeMgrVersion` selector, it returns a version code in the `response` parameter. In this case, a returned value of 3 indicates that the extended Time Manager is available.

In most cases, the last few characters in the selector's symbolic name form a suffix that indicates what type of value you can expect `Gestalt` to place in the `response` parameter. For example, if the suffix in a `Gestalt` selector is `Size`, then `Gestalt` returns a size in the `response` parameter. Table 1-1 lists the meaningful suffixes.

Table 1-1 `Gestalt` selector suffixes and their meanings

Suffix	Returned value
`Attr`	A range of 32 bits, the meanings of which are defined by a list of constants. Bit 0 is the least significant bit of the long word.
`Count`	A number indicating how many of the indicated type of item exist.
`Size`	A size, usually in bytes.
`Table`	The base address of a table.
`Type`	An index to a list of feature descriptions.
`Version`	A version number, which can be either a constant with a defined meaning or an actual version number, usually stored as four hexadecimal digits in the low-order word of the return value. Implied decimal points may separate digits. The value $0701, for example, returned in response to the `gestaltSystemVersion` selector, represents system software version 7.0.1.

Selectors that have the suffix `Attr` deserve special attention. They cause `Gestalt` to return a bit field that your application must interpret to determine whether a desired feature is present. For example, the application-defined sample function `MyGetSoundAttr`, defined in Listing 1-2 on page 1-6, returns a `LongInt` that contains the Sound Manager attributes field retrieved from `Gestalt`. To determine whether a particular feature is available, you need to look at the designated bit. The application-defined sample function `MyIsStereoMixing` in Listing 1-3, for example, determines whether stereo mixing is available.

Listing 1-3 Interpreting a `Gestalt` attributes response

```
FUNCTION MyIsStereoMixing: Boolean;
BEGIN
   MyIsStereoMixing := BTst(MyGetSoundAttr, gestaltStereoMixing);
END;
```

The `MyIsStereoMixing` function uses the MPW Pascal function `BTst` and the application-defined `MyGetSoundAttr` function to determine whether the stereo-mixing bit is set in the `response` value returned by `Gestalt` when it's called with the `gestaltSoundAttr` selector. The constant `gestaltStereoMixing` is defined in the header files.

Adding a New Selector Code

You can add your own selector code to those already understood by `Gestalt` by calling the `NewGestalt` function. Typically, a system extension registers itself with the Gestalt Manager so that applications that might use its services can find out whether it's there. A debugger, for example, could register its presence. Programmers working on an application could then embed instructions for the debugger in code under development and call Gestalt to make sure the debugger is available before invoking those instructions.

The NewGestalt function requires two parameters: the new selector to be registered and the address of the associated **selector function**. Gestalt executes the selector function to determine what value to pass back when it's called with the new selector code.

The selector code is a four-character sequence of type OSType. If you have registered a creator string with Apple Computer, Inc., you are strongly encouraged to use that sequence as your selector code. The Pipeline debugger, for example, with a creator string of 'PIPE', would use a Gestalt selector code of 'PIPE'.

Note

Apple reserves for its own use all four-character sequences consisting solely of lowercase letters and nonalphabetic ASCII characters. u

When you register your own selector code with the Gestalt Manager, you supply the address of the selector function to be executed when an application calls Gestalt with that code. Your selector function must reside in the system heap and must have the following interface:

```
FUNCTION MySelectorFunction (selector: OSType;
                             VAR response: LongInt): OSErr;
```

The Gestalt function passes its input parameters on to your selector function. Your function places the requested information in the LongInt pointed to by the response parameter and returns an error code, which Gestalt returns to its caller.

Your selector function should be as simple as possible. If your function needs to use global variables from the A5 world—that of your own software or that of some other software—it must explicitly set up A5 and then restore it upon exit. (See *Inside Macintosh: Memory* for an explanation of setting up and restoring the A5 world.)

Your selector function can, if necessary, call Gestalt and pass it other selector codes. Note that the response parameter is merely the address into which your function places the information requested. You cannot use that parameter to pass information to your selector function.

Listing 1-4 illustrates a minimal selector function that sets the response parameter and returns an error code of noErr. The application-defined sample function, MyGestaltPipe, is isolated in a UNIT element for separate compilation and placement in a resource.

Listing 1-4 Defining a simple Gestalt selector function

```
UNIT GestaltFunc;
INTERFACE
   USES OSIntf;
   FUNCTION MyGestaltPipe (gestaltSelector: OSType;
```

```
                           VAR gestaltReply: LongInt): OSErr;
IMPLEMENTATION
   FUNCTION MyGestaltPipe;
   BEGIN
      gestaltReply := $ACE;      {reply defined by Pipeline}
      MyGestaltPipe := noErr;    {too simple for errors}
   END;
END.
```

This sample linking command places the compiled code in resource ID 128 of a type arbitrarily named `'GDEF'`.

```
Link GestaltFunc.p.o -rn -rt GDEF=128 -o Pipeline
```

To add a `Gestalt` selector code, you first move the selector function into the system heap and then call the `NewGestalt` function, which adds the selector code and its function to the `Gestalt` repertoire.

s **WARNING**

Take special care when accessing memory in the system heap; it persists even after your application terminates. s

Listing 1-5 illustrates the installation of a new `Gestalt` selector.

Listing 1-5 Installing a new `Gestalt` selector

```
PROCEDURE MyInstallGestFunc;
VAR
   gestFuncHandle:   Handle;
   gestFuncSize:     Size;
   gestSysPtr:       Ptr;
   myErr:            OSErr;
BEGIN
   gestFuncHandle := GetResource('GDEF', 128);
   IF ResError = noErr THEN
   BEGIN
      gestFuncSize := SizeResource(gestFuncHandle);
      gestSysPtr := NewPtrSys(gestFuncSize);
      IF MemError = noErr THEN
      BEGIN
         BlockMove(gestFuncHandle^, gestSysPtr, gestFuncSize);
         FlushInstructionCache;
         myErr := NewGestalt('PIPE',
                  SelectorFunctionUUP(gestSysPtr));
      END;
```

```
      ReleaseResource(gestFuncHandle);
   END;
END;
```

The application-defined sample procedure MyInstallGestFunc loads the resource and then gets its size so it can allocate a pointer in the system heap. It then copies the resource to the pointer and releases the resource.

s **WARNING**

Be sure to call the FlushInstructionCache procedure every time you modify code in RAM. See the chapter "Memory Management Utilities" in *Inside Macintosh: Memory* for details about FlushInstructionCache. s

Finally, MyInstallGestFunc calls NewGestalt to register the selector code 'PIPE' and its selector function with the Gestalt Manager.

Because the new selector function resides in the system heap, Gestalt recognizes and responds to the new selector until the machine restarts, even if your software terminates before that time. You might therefore want your selector function to determine whether your software is still running before filling in the response value. The simplest way to report that your application is not available is to return an error code.

If you attempt to add a selector code that Gestalt already recognizes, NewGestalt returns the error code gestaltDupSelectorErr.

Modifying a Selector Function

You can use the ReplaceGestalt function to modify the function that Gestalt executes when passed a particular selector code. Your replacement selector function must reside in the system heap and must conform to the interface defined in the previous section, "Adding a New Selector Code."

To allow the new function to call the function it's replacing, ReplaceGestalt returns the address of the previous function.

If you attempt to redefine a selector that is not yet defined, ReplaceGestalt returns an error code; in that case, the address of the previous function is undefined. Always test the result code of ReplaceGestalt before calling Gestalt with that selector or attempting to use the response parameter.

Note

If you modify the function associated with an existing Gestalt selector, do not use any bits in the response parameter that are not documented in this chapter. Apple reserves all undocumented bits in the response parameters returned by Apple-defined Gestalt selectors. u

Because ReplaceGestalt supplies the address of the function it's replacing, you can use it to retrieve the address of the selector function associated with a selector code.

Getting Environmental Information Without the Gestalt Manager

You can call the `SysEnvirons` function, which predates the Gestalt Manager, to get a brief description of the operating environment. The `SysEnvirons` function is available on all models of the Macintosh computer since the Macintosh SE and Macintosh II.

Note
The `SysEnvirons` function is not part of the Gestalt Manager, but is documented in this chapter for the sake of completeness. u

The `SysEnvirons` function fills in a record that contains the model of the machine, the System file version number, the microprocessor type, a keyboard type code, and Boolean indicators of whether the machine has a floating-point unit or Color QuickDraw. The system environment record includes one detail not available through `Gestalt`: the working directory reference number of the folder or volume that holds the System file (although that information is available through the `FindFolder` function). See “The System Environment Record” beginning on page 1-28 for a complete description of the system environment record.

Gestalt Manager Reference

This section lists the Gestalt selector codes and their defined return values and describes the system environment record, the three Gestalt Manager functions, and the `SysEnvirons` function.

Constants

This section lists the Apple-defined Gestalt Manager selector codes, describes the formats of their responses, and lists the constants defined for their return values.

You pass a selector code when you call `Gestalt` to specify the kind of information you need. Apple defines two distinct kinds of selector codes: environmental selectors, which supply information you can use to control the behavior of your application, and informational selectors, which supply information you can’t use to determine what hardware or software features are available.

The selector code constants use a set of suffixes that indicate what format the response value will take. Selectors with the suffix `Attr`, for example, return a 32-bit response value in which the individual bits represent specific attributes. The constants listed for these response values represent bit numbers. For a more general description of selectors and their response values, see “Interpreting Gestalt Responses” beginning on page 1-9.

The `Gestalt` function accepts the following environmental selectors.

Selector	Response bits and response values

`gestaltAddressingModeAttr`	Current addressing-mode attributes.
	`CONST`
	`  gestalt32BitAddressing = 0;`
	`  gestalt32BitSysZone    = 1;`
	`  gestalt32BitCapable    = 2;`
	The `gestalt32BitAddressing` attribute indicates that the machine started up with 32-bit addressing. The `gestalt32BitSysZone` attribute indicates that the system heap has 32-bit clean block headers (regardless of the type of addressing the machine started up in). See the book *Inside Macintosh: Memory* for more information about 32-bit addressing.

`gestaltAliasMgrAttr`	Alias Manager attributes.
	`CONST`
	`gestaltAliasMgrPresent   = 0;`
	`gestaltAliasMgrSupportsRemoteAppleTalk`
	`                         = 1;`

`gestaltAppleEventsAttr`	The Apple events attribute.
	`CONST`
	`  gestaltAppleEventsPresent = 0;`
	`  gestaltScriptingSupport   = 1;`
	`  gestaltOSLInSystem        = 2;`

`gestaltAppleTalkVersion`	The version number of the AppleTalk driver (in particular, the .MPP driver) currently installed. The version number is placed into the low-order byte of the result; ignore the three high-order bytes. If an AppleTalk driver is not currently open, the `response` parameter is 0.
`gestaltATalkVersion`	The version number of the AppleTalk driver, in the format introduced with AppleTalk version 56. (For a description of AppleTalk, see *Inside AppleTalk,* second edition.) The version is stored in the high 3 bytes of the return value. Byte 3 contains the major revision number, byte 2 contains the minor revision number, and byte 1 contains a constant that represents the release stage.

Selector	Response bits and response values
`gestaltATalkVersion` (continued)	`CONST development = $20; alpha = $40; beta = $60; final = $80; release = $80;` For example, if you call `Gestalt` with the `'atkv'` selector when AppleTalk version 57 is loaded, you receive the long integer response value `$39008000`. Byte 0 always contains 0.
`gestaltAUXVersion`	The version of A/UX if it is currently executing. The result is placed into the low-order word of the `response` parameter. If A/UX is not executing, `Gestalt` returns `gestaltUnknownErr`.
`gestaltCFMAttr`	Code Fragment Manager attributes. `CONST gestaltCFMPresent = 0;`
`gestaltCloseViewAttr`	The CloseView attributes `CONST gestaltCloseViewEnabled = 0; gestaltCloseViewDisplayMgrFriendly = 1;`
`getstaltComponentMgr`	The version of the Component Manager.
`getstaltCompressionMgr`	The version of the Image Compression Manager.
`gestaltConnMgrAttr`	Connection Manager attributes. `CONST gestaltConnMgrPresent = 0; gestaltConnMgrCMSearchFix = 1; gestaltConnMgrErrorString = 2; gestaltConnMgrMultiAsyncIO = 3;` The `gestaltConnMgrCMSearchFix` bit flag indicates that the fix is present that allows the `CMAddSearch` routine to work over the `mAttn` channel.
`gestaltCRMAttr`	Communications Resource Manager attributes. `CONST gestaltCRMPresent = 0; gestaltCRMPersistentFix = 1; gestaltCRMToolRsrcCalls = 2;`
`gestaltCTBVersion`	The version number of the Communications Toolbox (in the low-order word of the return value).
`gestaltDBAccessMgrAttr`	The Data Access Manager attribute. `CONST gestaltDBAccessMgrPresent = 0;`

Selector	Response bits and response values

`gestaltDictionaryMgrAttr`	The Dictionary Manager attributes.
	`CONST`
	`  gestaltDictionaryMgrPresent = 0;`

`gestaltDisplayMgrAttr`	The Display Manager attributes.
	`CONST`
	`  gestaltDisplayMgrPresent  = 0;`

`gestaltDITLExtAttr`	The Dialog Manager extensions attributes.
	`CONST`
	`  gestaltDITLExtPresent     = 0;`
	If this flag bit is TRUE, then the Dialog Manager extensions included in System 7 are available. See the book *Inside Macintosh: Macintosh Toolbox Essentials* for details about the Dialog Manager.

`gestaltDragMgrAttr`	Drag Manager attributes.
	`CONST`
	`  gestaltDragMgrPresent    = 0;`

`gestaltEasyAccessAttr`	Easy Access attributes.
	`CONST`
	`  gestaltEasyAccessOff      = 0;`
	`  gestaltEasyAccessOn       = 1;`
	`  gestaltEasyAccessSticky   = 2;`
	`  gestaltEasyAccessLocked   = 3;`

`gestaltEditionMgrAttr`	Edition Manager attributes.
	`CONST`
	`  gestaltEditionMgrPresent  = 0;`
	`  gestaltEditionMgrTranslationAware`
	`                            = 1;`

`gestaltExtToolboxTable`	The base address of the second half of the Toolbox trap table if the table is discontiguous. If the table is contiguous, this selector returns 0.

`gestaltFinderAttr`	Finder attributes.
	`CONST`
	`  gestaltFinderDropEvent    = 0;`
	`  gestaltFinderMagicPlacement`
	`                            = 1;`
	`  gestaltFinderCallsAEProcess`
	`                            = 2;`
	`  gestaltOSLCompliantFinder = 3;`
	`  gestaltFinderSupports4GBVolumes`
	`                            = 4;`
	`  gestaltFinderHandlesCFMFailures`
	`                            = 5;`
	`  gestaltFinderHasClippings = 6;`

Selector	Response bits and response values

`gestaltFindFolderAttr`	The `FindFolder` function attribute.
	`CONST`
	`  gestaltFindFolderPresent = 0;`

`gestaltFirstSlotNumber`	The first physical slot.

`gestaltFontMgrAttr`	The Font Manager attribute.
	`CONST`
	`  gestaltOutlineFonts = 0;`

`gestaltFPUType`	A constant that represents the type of floating-point unit currently installed, if any.
	`CONST`
	`  gestaltNoFPU     = 0;`
	`  gestalt68881     = 1;`
	`  gestalt68882     = 2;`
	`  gestalt68040FPU = 3;`

`gestaltFSAttr`	File system attributes.
	`CONST`
	`  gestaltFullExtFSDispatching = 0;`
	`  gestaltHasFSSpecCalls       = 1;`
	`  gestaltHasFileSystemManager = 2;`
	`  gestaltFSMDoesDynamicLoad   = 3;`
	`  gestaltFSSupports4GBVols    = 4;`
	`  gestaltHasExtendedDiskInit  = 6;`

`gestaltFXfrMgrAttr`	The File Transfer Manager attributes.
	`CONST`
	`  gestaltFXfrMgrPresent       = 0;`
	`  gestaltFXfrMgrMultiFile     = 1;`
	`  gestaltFXfrMgrErrorString   = 2;`

`gestaltHelpMgrAttr`	The Help Manager attribute.
	`CONST`
	`  gestaltHelpMgrPresent       = 0;`

`gestaltIconUtilitiesAttr`	The Icon Utilities attribute.
	`CONST`
	`  gestaltIconUtilitiesPresent = 0;`

`gestaltKeyboardType`	A constant that represents the type of keyboard.
	`CONST`
	`  gestaltMacKbd               = 1;`
	`  gestaltMacAndPad            = 2;`
	`  gestaltMacPlusKbd           = 3;`
	`  gestaltExtADBKbd            = 4;`
	`  gestaltStdADBKbd            = 5;`
	`  gestaltPrtblADBKbd          = 6;`
	`  gestaltPrtblISOKbd          = 7;`

Selector	Response bits and response values
gestaltKeyboardType (continued)	`gestaltStdISOADBKbd = 8;` `gestaltExtISOADBKbd = 9;` `gestaltADBKbdII = 10;` `gestaltADBISOKbdII = 11;` `gestaltPwrBookADBKbd = 12;` `gestaltPwrBookISOADBKbd = 13;` `gestaltAppleAdjustKeypad = 14;` `gestaltAppleAdjustADBKbd = 15;` `gestaltAppleAdjustISOKbd = 16;` If the Apple Desktop Bus (ADB) is in use, there may be multiple keyboards or other ADB devices attached to the machine. The gestaltKeyboardType selector identifies only the type of the keyboard on which the last keystroke occurred. You cannot use this selector to find out what ADB devices are connected. For that, you can use the Apple Desktop Bus Manager, described in *Inside Macintosh: Devices.* Note that the ADB keyboard types described by Gestalt do not necessarily map directly to ADB device handler IDs. Future support for the gestaltKeyboardType selector is not guaranteed. To determine the type of the keyboard last touched without using Gestalt, check the system global variable KbdType, documented in *Inside Macintosh: Devices.* If the Gestalt Manager does not recognize the keyboard type, it returns an error.
gestaltLogicalPageSize	The logical page size. This value is defined only on machines with the MC68010, MC68020, MC68030, or MC68040 microprocessors. On a machine with the MC68000, Gestalt returns an error when called with this selector.
gestaltLogicalRAMSize	The amount of logical memory available. This value is the same as that returned by gestaltPhysicalRAMSize when virtual memory is not installed. On some machines, however, this value might be less than the value returned by gestaltPhysicalRAMSize because some RAM may be used by the video display and the Operating System.
gestaltLowMemorySize	The size (in bytes) of the low-memory area. The low-memory area is used for vectors, global variables, and dispatch tables.

Selector	Response bits and response values

`gestaltMiscAttr`	Information about miscellaneous pieces of the Operating System or hardware configuration.
	`CONST`
	`  gestaltScrollingThrottle = 0;`
	`  gestaltSquareMenuBar     = 2;`

`gestaltMixedModeVersion`	The version of Mixed Mode Manager.

`gestaltMMUType`	A constant that represents the type of MMU currently installed.
	`CONST`
	`  gestaltNoMMU     = 0;`
	`  gestaltAMU       = 1;`
	`  gestalt68851     = 2;`
	`  gestalt68030MMU  = 3;`
	`  gestalt68040MMU  = 4;`
	`  gestaltEMMU1     = 5;`

`gestaltNativeCPUtype`	Native CPU type.
	`CONST`
	`  gestaltCPU68000  = $000;`
	`  gestaltCPU68010  = $001;`
	`  gestaltCPU68020  = $002;`
	`  gestaltCPU68030  = $003;`
	`  gestaltCPU68040  = $004;`
	
	`  gestaltCPU601    = $101;`
	Note, to check whether the native system architecture is a MC680x0 or a PowerPC microprocessor, use the `gestaltSysArchitecture` selector.

`gestaltNotificationMgrAttr`	The Notification Manager attribute.
	`CONST`
	`  gestaltNotificationPresent = 0;`

`gestaltNuBusConnectors`	A bitmap that describes the NuBus™ slot connector locations. On a Macintosh II, for example, the return value would have bits 9 through 14 set, indicating that 6 NuBus slots are present, at locations 9 through 14.
`gestaltOSAttr`	General Operating System attributes, such as whether temporary memory handles are real handles. The low-order bits of the `response` parameter are interpreted as bit flags. A flag is set to 1 to indicate that the corresponding feature is available. Currently, the following bits are significant:

Selector	Response bits and response values

`gestaltOSAttr` (continued)	`CONST`
	`  gestaltSysZoneGrowable      = 0;`
	`  gestaltLaunchCanReturn      = 1;`
	`  gestaltLaunchFullFileSpec   = 2;`
	`  gestaltLaunchControl        = 3;`
	`  gestaltTempMemSupport       = 4;`
	`  gestaltRealTempMemory       = 5;`
	`  gestaltTempMemTracked       = 6;`
	See the book *Inside Macintosh: Memory* for a full explanation of the temporary memory features, and see the book *Inside Macintosh: Processes* for a full explanation of the launch control features.

`gestaltOSTable`	The base address of the Operating System trap dispatch table.

`gestaltParityAttr`	Information about the machine's parity-checking features.
	`CONST`
	`  gestaltHasParityCapability = 0;`
	`  gestaltParityEnabled       = 1;`
	Note that parity is not considered to be enabled unless *all* installed memory is parity RAM.

`gestaltPCXAttr`	PC Exchange attributes.
	`CONST`
	`  gestaltPCXHas8and16BitFAT  = 0;`
	`  gestaltPCXHasProDOS        = 1;`

`gestaltPhysicalRAMSize`	The number of bytes of physical RAM currently installed.
`gestaltPopupAttr`	The attribute of the pop-up control definition. `CONST gestaltPopupPresent = 0;`

`gestaltPowerMgrAttr`	Power Manager attributes.
	`CONST`
	`  gestaltPMgrExists          = 0;`
	`  gestaltPMgrCPUIdle         = 1;`
	`  gestaltPMgrSCC             = 2;`
	`  gestaltPMgrSound           = 3;`
	`  gestaltPMgrDispatchExists  = 4;`

`gestaltPPCToolboxAttr`	Program-to-Program Communication (PPC) Toolbox attributes. Note that these constants are defined as masks, not bit numbers.
	`CONST`
	`  gestaltPPCToolboxPresent   = $0000;`
	`  gestaltPPCSupportsRealTime = $1000;`
	`  gestaltPPCSupportsIncoming = $0001;`
	`  gestaltPPCSupportsOutgoing = $0002;`

Selector	Response bits and response values

`gestaltProcessorType`	A constant that represents the type of microprocessor currently running.
	`CONST`
	`  gestalt68000 = 1;`
	`  gestalt68010 = 2;`
	`  gestalt68020 = 3;`
	`  gestalt68030 = 4;`
	`  gestalt68040 = 5;`

`gestaltQuickdrawFeatures`	QuickDraw features.
	`CONST`
	`  gestaltHasColor          = 0;`
	`  gestaltHasDeepGWorlds    = 1;`
	`  gestaltHasDirectPixMaps  = 2;`
	`  gestaltHasGrayishTextOr  = 3;`
	`  gestaltSupportsMirroring = 4;`

`gestaltQuickdrawVersion`	The version of QuickDraw, encoded as a revision number in the low-order word of the return value. The high-order byte represents the major revision number, and the low-order byte represents the minor revision number. For example, version 1.3 of 32-Bit QuickDraw represents QuickDraw revision 2.3; its `response` value is `$0230`.
	`CONST`
	`  gestaltOriginalQD  = $000;`
	`  gestalt8BitQD      = $100;`
	`  gestalt32BitQD     = $200;`
	`  gestalt32BitQD11   = $210;`
	`  gestalt32BitQD12   = $220;`
	`  gestalt32BitQD13   = $230;`
	Values having a major revision number of 1 or 2 indicate that Color QuickDraw is available, in either the 8-bit or 32-bit version. These results do not, however, indicate whether a color monitor is attached to the system. You must use high-level QuickDraw routines to obtain that information.

`gestaltQuickTimeVersion`	The QuickTime version.

`gestaltRealtimeMgrAttr`	Realtime Manager attributes.
	`CONST`
	`  gestaltRealtimeMgrPresent = 0;`

`gestaltResourceMgrAttr`	The Resource Manager attribute.
	`CONST`
	`  gestaltPartialRsrcs = 0;`

Selector	Response bits and response values

`gestaltScrapMgrAttr`	Scrap Manager attributes.
	`CONST`
	`  gestaltScrapMgrTranslationAware = 0;`
	`  gestaltTranslationMgrHintOrder = 1;`

`gestaltScriptCount`	The number of script systems currently active.
`gestaltScriptMgrVersion`	The version number of the Script Manager (in the low-order word of the return value).

`gestaltSerialAttr`	Serial hardware attributes of the machine, such as whether or not the GPIa line is connected and can be used for external clocking.
	`CONST`
	`  gestaltHasGPIaToDCDa  = 0;`
	`  gestaltHasGPIaToRTxCa = 1;`
	`  gestaltHasGPIaToDCDb  = 2;`

`gestaltSlotAttr`	Slot Manager attributes.
	`CONST`
	`  gestaltSlotMgrExists       = 0;`
	`  gestaltNuBusPresent        = 1;`
	`  gestaltSESlotPresent       = 2;`
	`  gestaltSE30SlotPresent     = 3;`
	`  gestaltPortableSlotPresent = 4;`

`gestaltSoundAttr`	Sound attributes.
	`CONST`
	`  gestaltStereoCapability     = 0;`
	`  gestaltStereoMixing         = 1;`
	`  gestaltSoundIOMgrPresent    = 3;`
	`  gestaltBuiltInSoundInput    = 4;`
	`  gestaltHasSoundInputDevice  = 5;`
	`  gestaltPlayAndRecord        = 6;`
	`  gestalt16BitSoundIO         = 7;`
	`  gestaltStereoInput          = 8;`
	`  gestaltLineLevelInput       = 9;`
	`  gestaltSndPlayDoubleBuffer  = 10;`
	`  gestaltMultiChannels        = 11`
	`  gestalt16BitAudioSupport    = 12;`

Selector	Response bits and response values
`gestaltSoundAttr` (continued)	If the bit `gestaltStereoCapability` is `TRUE`, the available hardware can play stereo sounds. The bit `gestaltStereoMixing` indicates that the sound hardware of the machine mixes both left and right channels of stereo sound into a single audio signal for the internal speaker. The `gestaltSoundIOMgrPresent` bit indicates that the new sound input routines are available, and the `gestaltBuiltInSoundInput` bit indicates that a built-in sound input device is available. The `gestaltHasSoundInputDevice` bit indicates that some sound input device is available. Note, bits 7 through 12 are not defined for versions of the Sound Manager prior to version 3.0.

`gestaltSpeechAttr`	Speech Manager attributes.
	
	`CONST`
	`  getaltSpeechMgrPresent     = 0;`
	`  getaltSpeechHasPPCGlue     = 1;`

`gestaltStandardFileAttr`	Standard File Package attributes.
	
	`CONST`
	`  gestaltStandardFile58      = 0;`
	`  gestaltStandardFileTranslationAware`
	`                             = 1;`
	`  gestaltStandardFileHasColorIcons`
	`                             = 2;`
	
	If the `gestaltStandardFile58` flag bit is set, you can call the four new procedures—`StandardPutFile`, `StandardGetFile`, `CustomPutFile`, and `CustomGetFile`—introduced with System 7. (The name of the constant reflects the enabling of selectors 5 through 8 on the trap macro that handles the Standard File Package.)

`gestaltStdNBPAttr`	Information about the `StandardNBP` (Name-Binding Protocol) function.
	
	`CONST`
	`  gestaltStdNBPPresent = 0;`

`gestaltSysArchitecture`	The native system architecture.
	
	`CONST`
	`  gestalt68k        = 1;`
	`  gestaltPowerPC    = 2;`
	
	If the `gestalt68k` flag bit is set, the native microprocessor is a MC680x0 chip. If the `gestaltPowerPC` flag bit is set, the native microprocessor is a PowerPC chip.

Selector	Response bits and response values

`gestaltTEAttr`	TextEdit attributes.
	`CONST`
	`  gestaltTEHasGetHiliteRgn = 0;`

`gestaltTermMgrAttr`	Terminal Manager attributes.
	`CONST`
	`  gestaltTermMgrPresent     = 0;`
	`  gestaltTermMgrErrorString = 2;`

`gestaltTextEditVersion`	A constant that indicates which version of TextEdit is present.
	`CONST`
	`  gestaltTE1 = 1;`
	`  gestaltTE2 = 2;`
	`  gestaltTE3 = 3;`
	`  gestaltTE4 = 4;`
	`  gestaltTE5 = 5;`

`gestaltThreadMgrAtt`	Thread Manager attributes.
	`CONST`
	`  gestaltThreadMgrPresent     = 0;`
	`  gestaltSpecificMatchSupport = 1;`

`gestaltTimeMgrVersion`	A constant that indicates which version of the Time Manager is present.
	`CONST`
	`  gestaltStandardTimeMgr = 1;`
	`  gestaltRevisedTimeMgr  = 2;`
	`  gestaltExtendedTimeMgr = 3;`

`gestaltToolboxTable`	The base address of the Toolbox trap dispatch table.

`gestaltTranslationAttr`	The Translation Manager attributes.
	`CONST`
	`  gestaltTranslationMgrExists = 0;`

`gestaltTSMgrVersion`	The version of the Text Services.
`gestaltVersion`	The version of the Gestalt Manager (in the low-order word of the return value). The current version is 1, corresponding to a returned value of $0001.

`gestaltVMAttr`	The virtual memory attributes.
	`CONST`
	`  gestaltVMPresent      = 0;`

The `Gestalt` function also accepts the following informational selectors.

s **WARNING**
Never infer the existence of certain hardware or software features from the responses that `Gestalt` returns when you pass it an informational selector. s

Selector	Meaning

`gestaltHardwareAttr`	Low-level hardware configuration attributes.
	`CONST`
	`  gestaltHasVIA1         = 0;`
	`  gestaltHasVIA2         = 1;`
	`  gestaltHasASC          = 3;`
	`  gestaltHasSCC          = 4;`
	`  gestaltHasSCSI         = 7;`
	`  gestaltHasSoftPowerOff = 19;`
	`  gestaltHasSCSI961      = 21;`
	`  gestaltHasSCSI962      = 22;`
	`  gestaltHasUniversalROM = 24;`
	The `gestaltHasSCSI` bit means the machine is equipped with a SCSI implementation based on the 53C80 chip, which was introduced in the Macintosh Plus. This bit is 0 on computers with a different SCSI implementation. Those computers set the `gestaltHasSCSI961` or `gestaltHasSCSI962` bit to report a SCSI implementation based on the 53C96 chip installed on an internal or external bus, respectively.
	The `gestaltHasSCC` bit is normally returned as 0 on the Macintosh IIfx and Macintosh Quadra 900 computers, which have intelligent I/O processors that isolate the hardware and make direct access to the SCC impossible. However, if the user has used the Compatibility Switch control panel to enable compatibility mode, `gestaltHasSCC` is set.

`gestaltMachineIcon`	The icon family resource ID for the current type of Macintosh.

`gestaltMachineType`	A constant that indicates the model of computer.
	`CONST`
	`  gestaltClassic          = 1;`
	`  gestaltMacXL            = 2;`
	`  gestaltMac512KE         = 3;`
	`  gestaltMacPlus          = 4;`
	`  gestaltMacSE            = 5;`
	`  gestaltMacII            = 6;`
	`  gestaltMacIIx           = 7;`
	`  gestaltMacIIcx          = 8;`
	`  gestaltMacSE030         = 9;`
	`  gestaltPortable         = 10;`

Selector	Meaning

`gestaltMachineType` (continued)	`gestaltMacIIci = 11;` `gestaltMacIIfx = 13;` `gestaltMacClassic = 17;` `gestaltMacIIsi = 18;` `gestaltMacLC = 19;` `gestaltQuadra900 = 20;` `gestaltPowerBook170 = 21;` `gestaltQuadra700 = 22;` `gestaltClassicII = 23;` `gestaltPowerBook100 = 24;` `gestaltPowerBook140 = 25;` `gestaltQuadra950 = 26;` `gestaltMacLCIII = 27;` `gestaltPowerBookDuo210 = 29;` `gestaltMacCentris650 = 30;` `gestaltPowerBookDuo230 = 32;` `gestaltPowerBook180 = 33;` `gestaltPowerBook160 = 34;` `gestaltMacQuadra800 = 35;` `gestaltMacLCII = 37;` `gestaltPowerBookDuo250 = 38;` `gestaltMacIIvi = 44;` `gestaltPerforma600 = 45;` `gestaltMacIIvx = 48;` `gestaltMacColorClassic = 49;` `gestaltPowerBook165c = 50;` `gestaltMacCentris610 = 52;` `gestaltMacQuadra610 = 53;` `gestaltPowerBook145 = 54;` `gestaltMacLC520 = 56;` `gestaltMacCentris660AV = 60;` `gestaltPowerBook180c = 71;` `gestaltPowerBookDuo270c = 77;` `gestaltMacQuadra840AV = 78;` `gestaltPowerBook165 = 84;` `gestaltMacTV = 88;` `gestaltMacLC475 = 89;` `gestaltMacLC575 = 92;` `gestaltMacQuadra605 = 94;` `gestaltPowerMac8100_80 = 65;` `gestaltPowerMac6100_60 = 75;` `gestaltPowerMac7100_66 = 112;`
	To obtain a string containing the machine's name, you can pass the returned value to the `GetIndString` procedure as an index into the resource of type `'STR#'` in the System file having the resource ID defined by the constant `kMachineNameStrID`.
	`CONST`
	`kMachineNameStrID = -16395;`

Selector	Meaning
gestaltROMSize	The size of the installed ROM, in bytes. The value is returned in only one word.
gestaltROMVersion	The version number of the installed ROM (in the low-order word of the return value).
gestaltSystemVersion	The version number of the currently active System file, represented as four hexadecimal digits in the low-order word of the return value. For example, if your application is running in version 7.0.1, then Gestalt returns the value $0701. Ignore the high-order word of the returned value.

Data Structures

This section describes the record filled in by the SysEnvirons function.

The System Environment Record

The SysEnvirons function fills in a system environment record, which describes some aspects of the software and hardware environment.

```
TYPE SysEnvRec =
   RECORD
      environsVersion:  Integer;
      machineType:      Integer;
      systemVersion:    Integer;
      processor:        Integer;
      hasFPU:           Boolean;
      hasColorQD:       Boolean;
      keyBoardType:     Integer;
      atDrvrVersNum:    Integer;
      sysVRefNum:       Integer;
   END;
```

FIELD DESCRIPTIONS

environsVersion

The version number of the SysEnvirons function that was used to fill in the record.

When you call SysEnvirons, you specify a version number to ensure that you receive a system environment record that matches your expectations, as explained in the description of SysEnvirons beginning on page 1-32. If you request a more recent version of SysEnvirons than is available, SysEnvirons places its own version number in the environsVersion field and returns a function result envVersTooBig.

`machineType` A code for the Macintosh model, which can be one of these values:

```
CONST
   envXL              = -2; {Macintosh XL}
   envMac             = –1; {Macintosh with 64K }
                            { ROM}
   envMachUnknown     = 0;  {unknown model, }
                            { after Macintosh }
                            { IIfx}
   env512KE           = 1;  {Macintosh 512K }
                            { enhanced}
   envMacPlus         = 2;  {Macintosh Plus}
   envSE              = 3;  {Macintosh SE}
   envMacII           = 4;  {Macintosh II}
   envMacIIx          = 5;  {Macintosh IIx}
   envMacIIcx         = 6;  {Macintosh IIcx}
   envSE30            = 7;  {Macintosh SE30}
   envPortable        = 8;  {Macintosh Portable}
   envMacIIci         = 9;  {Macintosh IIci}
   envMacIIfx         = 11; {Macintosh IIfx}
```

Note

Use Gestalt to obtain information about machine types not listed above. u

`systemVersion` The version number of the current System file, represented as two byte-long numbers with one or more implied decimal points. The value $0410, for example, represents system software version 4.1.

If you call `SysEnvirons` when a system earlier than 4.1 is running, the MPW glue places $0 in this field and returns a result code of `envNotPresent`.

`processor` A code for the microprocessor, which can be one of these values:

```
CONST
   envCPUUnknown      = 0;  {unknown }
                            { microprocessor}
   env68000           = 1;  {MC68000}
   env68010           = 2;  {MC68010}
   env68020           = 3;  {MC68020}
   env68030           = 4;  {MC68030}
   env68040           = 5;  {MC68040}
```

`hasFPU` A Boolean value that indicates whether hardware floating-point processing is available.

`hasColorQD` A Boolean value that indicates whether Color QuickDraw is present. This field says nothing about the presence of a color monitor.

keyboardType A code for the keyboard type, which can be one of these values:

```
CONST
   envUnknownKbd     = 0;  {Macintosh Plus with }
                           { keypad}
   envMacKbd         = 1;  {Macintosh}
   envMacAndPad      = 2;  {Macintosh with keypad}
   envMacPlusKbd     = 3;  {Macintosh Plus}
   envAExtendKbd     = 4;  {Apple extended}
   envStandADBKbd    = 5;  {standard ADB}
   envPrtblADBKbd    = 6;  {Macintosh Portable ADB}
   envPrtblISOKbd    = 7;  {Macintosh Portable ISO}
   envStdISOADBKbd   = 8;  {standard ISO ADB}
   envExtISOADBKbd   = 9;  {extended ISO ADB}
```

Note

Use Gestalt to obtain information about keyboard types not listed above. u

If the Apple Desktop Bus is in use, this field returns the keyboard type of the keyboard on which the last keystroke was made.

atDrvrVersNum The version number of the AppleTalk driver (specifically, the .MPP driver) currently installed. If AppleTalk is not loaded, this field is 0.

sysVRefNum The working-directory reference number of the folder or volume that holds the open System file.

Gestalt Manager Routines

This section describes the three Gestalt Manager functions, `Gestalt`, `NewGestalt`, and `ReplaceGestalt`. It also describes the `SysEnvirons` function, which can give you a brief description of the operating environment when `Gestalt` is not available. The Gestalt Manager functions allow you to

- find out what hardware and software features are present
- add new selectors to those understood by the `Gestalt` function
- replace the functions associated with known selectors

Getting Information About the Operating Environment

This section describes both the `Gestalt` function, which you use to find out about the operating environment, and the `SysEnvirons` function, which you use only when `Gestalt` is not available.

Gestalt

You can use the Gestalt function to obtain information about the operating environment. You specify what information you need by passing one of the selector codes recognized by Gestalt.

```
FUNCTION Gestalt (selector: OSType; VAR response: LongInt): OSErr;
```

`selector` The selector code for the information you need.

`response` On exit, the requested information whose format depends on the selector code specified in the selector parameter.

DESCRIPTION

The Gestalt function places the information requested by the selector parameter in the variable parameter response. Note that Gestalt returns the response from all selectors in a long word, which occupies 4 bytes. When not all 4 bytes are needed, the significant information appears in the low-order byte or bytes. Although the response parameter is declared as a variable parameter, you cannot use it to pass information to Gestalt or to a Gestalt selector function. Gestalt interprets the response parameter as an address at which it is to place the result returned by the selector function specified by the selector parameter. Gestalt ignores any information already at that address.

The Apple-defined selector codes fall into two categories: environmental selectors, which supply specific environmental information you can use to control the behavior of your application, and informational selectors, which supply information you can't use to determine what hardware or software features are available. You can use one of the selector codes defined by Apple (listed in the "Constants" section beginning on page 1-14) or a selector code defined by a third-party product.

Selectors with the suffix Attr return a 32-bit response value in which the individual bits represent specific attributes. The constants listed for these response values represent bit numbers.

SPECIAL CONSIDERATIONS

When passed one of the Apple-defined selector codes, the Gestalt function does not move or purge memory and therefore may be called at any time, even at interrupt time. However, selector functions associated with non-Apple selector codes might move or purge memory, and third-party software can alter the Apple-defined selector functions. Therefore, it is safest always to assume that Gestalt could move or purge memory.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for the `Gestalt` function are

Registers on entry

D0	Selector code

Registers on exit

A0	Response
D0	Result code

RESULT CODES

`noErr`	0	No error
`gestaltUnknownErr`	–5550	Could not obtain the response
`gestaltUndefSelectorErr`	–5551	Undefined selector

SEE ALSO

See the documentation of the features you're interested in for more information on the various response values and their meanings.

See "Interpreting Gestalt Responses" beginning on page 1-9 for a discussion of the different response value formats and a sample function that checks an attributes value for a specific feature.

See "Getting Information About the Operating Environment" beginning on page 1-6 for a sample function that calls the `Gestalt` function and checks the validity of the return value. See the "Constants" section beginning on page 1-14 for a list of selector codes defined by Apple and the formats of their responses.

SysEnvirons

You can use the `SysEnvirons` function when you need information about the operating environment and the `Gestalt` function is not available.

```
FUNCTION SysEnvirons (versionRequested: Integer;
                      VAR theWorld: SysEnvRec): OSErr;
```

`versionRequested`
The version number of `SysEnvirons` you expect.

`theWorld` A system environment record.

DESCRIPTION

The `SysEnvirons` function fills in a system environment record identified by the variable parameter `theWorld`. It returns a result code.

You use the `versionRequested` parameter to tell `SysEnvirons` which version of the system environment record you're prepared to receive. This chapter documents version 2, which contains the same fields as version 1 but recognizes a more complete set of descriptive constants. Apple will raise the `SysEnvirons` version number in the future only if the record structure changes. You can trust any future revision to return the version 2 record if you request it, although the record might contain whatever constants are then current. To request the most recent version, you can use the constant `curSysEnvVers`:

```
CONST
   curSysEnvVers = 2;
```

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for the `SysEnvirons` function are

Registers on entry

A0	Address of a system environment record
D0	Version requested

Registers on exit

A0	Address of a system environment record
D0	Result code

RESULT CODES

`noErr`	0	No error
`envNotPresent`	–5500	`SysEnvirons` trap not present
`envBadVers`	–5501	Nonpositive version number passed
`envVersTooBig`	–5502	Requested version of `SysEnvirons` not available

SEE ALSO

See "The System Environment Record" beginning on page 1-28 for a detailed description of the system environment record.

Adding a Selector Code

You can add your own selector code using the `NewGestalt` function.

NewGestalt

You can use the `NewGestalt` function to add a selector code to those already recognized by `Gestalt`.

```
FUNCTION NewGestalt (selector: OSType;
                     gestaltFunction: SelectorFunctionUUP)
                     : OSErr;
```

`selector` The selector code you're adding, which is a four-character sequence of type `OSType`.

`gestaltFunction`
A pointer to the selector function that `Gestalt` executes when it receives the new selector code.

DESCRIPTION

The `NewGestalt` function registers a specified selector code with the Gestalt Manager so that when `Gestalt` is called with that selector code, the specified selector function is executed. The function result of `NewGestalt` is a result code.

Before calling `NewGestalt`, you must define a selector function and install it in the system heap. The selector function must conform to the interface defined in "Adding a New Selector Code" beginning on page 1-10.

Registering with the Gestalt Manager is a way for software such as system extensions to make their presence known to potential users of their services.

SPECIAL CONSIDERATIONS

The `NewGestalt` function might move memory and should not be called at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for the `NewGestalt` function are

Registers on entry

A0	Address of new selector function
D0	Selector code

Registers on exit

D0	Result code

RESULT CODES

noErr	0	No error
memFullErr	–108	Ran out of memory
gestaltDupSelectorErr	–5552	Selector already exists
gestaltLocationErr	–5553	Function not in system heap

SEE ALSO

See “Adding a New Selector Code” beginning on page 1-10 for a sample selector function and a sample procedure that installs it. For information about the `Gestalt` function, see page 1-31.

Modifying a Selector Function

You can install your own selector function for an established selector code using the `ReplaceGestalt` function.

ReplaceGestalt

You can use the `ReplaceGestalt` function to replace the function that is currently associated with a selector.

```
FUNCTION ReplaceGestalt (selector: OSType;
                         gestaltFunction: SelectorFunctionUUP;
                         VAR oldGestaltFunction:
                         SelectorFunctionUUP): OSErr;
```

`selector` The selector code for the function being replaced.

`gestaltFunction`
A pointer to the new selector function.

`oldGestaltFunction`
On exit, a pointer to the function previously associated with the specified selector.

DESCRIPTION

The `ReplaceGestalt` function replaces the selector function associated with an existing selector code.

So that your function can call the function previously associated with the selector, `ReplaceGestalt` places the address of the old selector function in the `oldGestaltFunction` parameter. If `ReplaceGestalt` returns an error of any type, then the value of `oldGestaltFunction` is undefined.

SPECIAL CONSIDERATIONS

The `ReplaceGestalt` function might move memory and should not be called at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for the `ReplaceGestalt` function are

Registers on entry

A0	Address of new selector function
D0	Selector code

Registers on exit

A0	Address of old selector function
D0	Result code

RESULT CODES

`noErr`	0	No error
`gestaltUndefSelectorErr`	–5551	Undefined selector
`gestaltLocationErr`	–5553	Function not in system heap

SEE ALSO

See “Modifying a Selector Function” on page 1-13 for a discussion of replacing selector functions. See “Adding a New Selector Code” beginning on page 1-10 for a sample selector function.

Application-Defined Routines

This section describes the `Gestalt` selector function, which is the function `Gestalt` executes to retrieve the information specified by a selector.

The Selector Function

If you add your own selector code or modify an existing selector code, you supply a selector function that returns the information associated with the selector.

MySelectorFunction

The selector function is responsible for placing the requested information in the `response` parameter and returning an appropriate error code.

```
FUNCTION MySelectorFunction (selector: OSType;
                             VAR response: LongInt): OSErr;
```

`selector` The selector code that triggers the function.

`response` On exit, the information.

DESCRIPTION

The selector function places the requested information in the `response` parameter and returns a result code. If the information is not available, the selector function returns the appropriate error code, which `Gestalt` returns as its function result.

A selector function can call `Gestalt` or even other selector functions. It must reside in the system heap.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for the selector function are

Registers on entry

D0	Selector code

Registers on exit

A0	Response
D0	Result code

RESULT CODES

`noErr`	0	No error
`gestaltUnknownErr`	–5550	Could not obtain the response

SEE ALSO

See “Adding a New Selector Code” beginning on page 1-10 for a sample selector function and a procedure that installs it in the system heap. For information about the `NewGestalt` function, see page 1-34. For information about the `ReplaceGestalt` function, see page 1-35.

Summary of the Gestalt Manager

Pascal Summary

Constants

Environmental Selector Codes

```
CONST
   gestaltAddressingModeAttr     = 'addr';   {addressing-mode attributes}
   gestaltAliasMgrAttr           = 'alis';   {Alias Manager attributes}
   gestaltAppleEventsAttr        = 'evnt';   {Apple events attributes}
   gestaltAppleTalkVersion       = 'atlk';   {old format AppleTalk version}
   gestaltATalkVersion           = 'atkv'    {new format AppleTalk version}
   gestaltAUXVersion             = 'a/ux';   {A/UX version, if present}
   gestaltCFMAttr                = 'cfrg';   {Code Fragment Manager attr}
   gestaltCloseViewAttr          = 'BSDa';   {CloseView attributes}
   gestaltComponentMgr           = 'cpnt';   {Component Manager version}
   gestaltCompressionMgr         = 'icmp'    {Image Compression Manager }
                                             { version}
   gestaltConnMgrAttr            = 'conn';   {Connection Manager attributes}
   gestaltCRMAttr                = 'crm ';   {Communication Resource }
                                             { Manager attr}
   gestaltCTBVersion             = 'ctbv';   {Comm Toolbox version}
   gestaltDBAccessMgrAttr        = 'dbac';   {Data Access Manager attributes}
   gestaltDictionaryMgrAttr      = 'dict';   {Dictionary Manager attributes}
   gestaltDisplayMgrAttr         = 'dply';   {Display Manager attributes}
   gestaltDisplayMgrVers         = 'dplv';   {Display Manager version}
   gestaltDITLExtAttr            = 'ditl';   {Dialog Manager extensions}
   gestaltDragMgrAttr            = 'drag';   {Drag Manager attributes}
   gestaltEasyAccessAttr         = 'easy';   {Easy Access attributes}
   gestaltEditionMgrAttr         = 'edtn';   {Edition Manager attributes}
   gestaltExtToolboxTable        = 'xttt';   {Toolbox trap dispatch table}
   gestaltFinderAttr             = 'fndr'    {Finder attributes}
   gestaltFindFolderAttr         = 'fold';   {FindFolder attributes}
   gestaltFirstSlotNumber        = 'slt1';   {first physical slot}
   gestaltFontMgrAttr            = 'font';   {Font Manager attributes}
   gestaltFPUType                = 'fpu ';   {floating-point unit type}
```

```
gestaltFSAttr                 = 'fs  ';   {file system attributes}
gestaltFXfrMgrAttr            = 'fxfr';   {File Transfer Manager attr}
gestaltHardwareAttr           = 'hdwr';   {hardware attributes}
gestaltHelpMgrAttr            = 'help';   {Help Manager attributes}
gestaltIconUtilitiesAttr      = 'icon';   {Icon Utilities attributes}
gestaltKeyboardType           = 'kbd ';   {keyboard type code}
gestaltLogicalPageSize        = 'pgsz';   {logical page size}
gestaltLogicalRAMSize         = 'lram';   {logical RAM size}
gestaltLowMemorySize          = 'lmem';   {size of low memory}
gestaltMiscAttr               = 'misc';   {miscellaneous attributes}
gestaltMixedModeVersion       = 'mixd';   {MixedMode version}
gestaltMMUType                = 'mmu ';   {MMU type}
gestaltNativeCPUtype          = 'cput';   {Native CPU type}
gestaltNotificationMgrAttr    = 'nmgr';   {Notification Manager attr}
gestaltNuBusConnectors        = 'sltc';   {NuBus connector bitmap}
getstaltNuBusSlotCount        = 'nubs';   {count of logical NuBus slots}
gestaltOSAttr                 = 'os  ';   {Operating System attributes}
gestaltOSTable                = 'ostt';   {base address of Operating }
                                          { System trap dispatch table}
gestaltParityAttr             = 'prty';   {parity attributes}
gestaltPCXAttr                = 'pcxg';   {PC exchange attributes}
gestaltPhysicalRAMSize        = 'ram ';   {physical RAM size}
gestaltPopupAttr              = 'pop!';   {pop-up 'CDEF' attributes}
gestaltPowerMgrAttr           = 'powr';   {Power Manager attributes}
gestaltPPCToolboxAttr         = 'ppc ';   {PPC Toolbox attributes}
gestaltProcessorType          = 'proc';   {microprocessor type code}
gestaltQuickdrawFeatures      = 'qdrw';   {QuickDraw features}
gestaltQuickdrawVersion       = 'qd  ';   {QuickDraw version}
gestaltQuickTime              = 'qtim';   {QuickTime version}
gestaltRealtimeAttr           = 'rtmr';   {Realtime Manager attributes}
gestaltResourceMgrAttr        = 'rsrc';   {Resource Manager attributes}
gestaltScrapMgrAttr           = 'scra';   {Scrap Manager attributes}
gestaltScriptCount            = 'scr#';   {number of active script }
                                          { systems}
gestaltScriptMgrVersion       = 'scri';   {Script Manager version}
gestaltSerialAttr             = 'ser ';   {serial hardware attributes}
gestaltSoundAttr              = 'snd ';   {sound attributes}
gestaltSpeechAttr             = 'ttsc';   {Speech Manager attributes}
gestaltStandardFileAttr       = 'stdf';   {Standard File attributes}
gestaltStdNBPAttr             = 'nlup';   {StandardNBP attributes}
gestaltSysArchitecture        = 'sysa';   {native system architecture}
gestaltTEAttr                 = 'teat';   {TextEdit attributes}
gestaltTermMgrAttr            = 'term';   {Terminal Manager attributes}
```

```
   gestaltTextEditVersion        = 'te  ';   {TextEdit version code}
   gestaltThreadMgrAtt           = 'thds';   {Thread Manager attributes}
   gestaltTimeMgrVersion         = 'tmgr';   {Time Manager version code}
   gestaltTranslationAttr        = 'xlat';   {Translation Manager attributes}
   gestaltTSMgrVersion           = 'tsmv';   {Text Services Manager version}
   gestaltToolboxTable           = 'tbtt';   {base address of Toolbox trap }
                                             { dispatch table}
   gestaltVersion                = 'vers';   {Gestalt version}
   gestaltVMAttr                 = 'vm  ';   {virtual memory attributes}
```

Informational Selector Codes

```
CONST
   gestaltHardwareAttr           = 'hdwr';   {hardware attributes}
   gestaltMachineIcon            = 'micn';   {machine 'ICON'/'cicn' res ID}
   gestaltMachineType            = 'mach';   {Macintosh model code}
   gestaltROMSize                = 'rom ';   {ROM size}
   gestaltROMVersion             = 'romv';   {ROM version}
   gestaltSystemVersion          = 'sysv';   {System file version number}
```

Environmental Selector Response Values

```
CONST
   {gestaltAddressingModeAttr response bits}
   gestalt32BitAddressing        = 0;        {booted in 32-bit mode}
   gestalt32BitSysZone           = 1;        {32-bit compatible system zone}
   gestalt32BitCapable           = 2;        {machine is 32-bit capable}

   {gestaltAliasMgrAttr response bits}
   gestaltAliasMgrPresent        = 0;        {Alias Manager is present}
   gestaltAliasMgrSupportsRemoteAppletalk    {Alias Manager knows about }
                                 = 1;        { remote AppleTalk}

   {gestaltAppleEventsAttr response bits}
   gestaltAppleEventsPresent     = 0;        {Apple events available}
   gestaltScriptingSupport       = 1;
   gestaltOSLInSystem            = 2;        {OSL in system}

   {gestaltATalkVersion release stage constants}
   development                   = $20;      {development}
   alpha                         = $40;      {alpha}
   beta                          = $60;      {beta}
   final                         = $80;      {final}
   release                       = $80;      {release}
```

```
{gestaltCFMAttr response bits}
gestaltsCFMPresent            = 0;     {Code Fragment Manager present}

{gestaltCloseViewAttr response bits}
gestaltCloseViewEnabled       = 0;     {CloseView enabled}
gestaltCloseViewDisplayMgrFriendly
                              = 1;     {CloseView compatible with }
                                       { Display Manger}

{gestaltConnMgrAttr response bits}
gestaltConnMgrPresent         = 0;     {Connection Manager present}
gestaltConnMgrCMSearchFix     = 1;     {CMAddSearch fix present}
gestaltConnMgrErrorString     = 2;     {has CMGetErrorString}
gestaltConnMgrMultiAsyncIO    = 3;     {has CMNewIOPB, CMDisposeIOPB, }
                                       { CMPBRead, CMPBWrite,and }
                                       { CMPBIOKill}

{gestaltCRMAttr response bits}
gestaltCRMPresent             = 0;     {Communication Resource Manager }
                                       { present}
gestaltCRMPersistentFix       = 1;     {fix for persistent tools}
gestaltCRMToolRsrcCalls       = 2;     {tool resource calls available}

{gestaltDBAccessMgrAttr response bits}
gestaltDBAccessMgrPresent     = 0;        {Data Access Manager present}

{gestaltDisplayMgrAttr response bits}
gestaltDisplayMgrPresent      = 0;        {Display Manager Present}

{gestaltDictionaryMgrAttr response bits}
gestaltDictionaryMgrPresent   = 0;        {Dictionary Manager present}

{gestaltDITLExtAttr response bits}
gestaltDITLExtPresent         = 0;        {Dialog Manager extensions }
                                          { present}

{gestaltDragMgrATtr response bits}
gestaltDragMgrPresent         = 0;        {Drag Manager present}

{gestaltEasyAccessAttr response bits}
gestaltEasyAccessOff          = 0;        {Easy Access present but off}
gestaltEasyAccessOn           = 1;        {Easy Access on}
gestaltEasyAccessSticky       = 2;        {Easy Access sticky}
gestaltEasyAccessLocked       = 3;        {Easy Access locked}
```

```
{gestaltEditionMgrAttr response bits}
gestaltEditionMgrPresent      = 0;        {Edition Manager present}
gestaltEditionMgrTranslationAware         {Edition Manager aware of }
                              = 1;        { Translation Manager}

{gestaltFinderAttr response bits}
gestaltFinderDropEvent        = 0;        {Finder recognizes drop event}
gestaltFinderMagicPlacement   = 1;        {Finder supports magic icon }
                                          { placement}
gestaltFinderCallsAEProcess   = 2;        {Finder calls }
                                          { AEProcessAppleEvent}
gestaltFinderOSLCompliantFinder
                              = 3;        {Finder is scriptable and }
                                          { recordable}
getstaltFinderSupports4GBVolumes
                              = 4;        {Finder handles 4GB volumes}
getstaltFinderHandlesCFMFailures
                              = 5;        {Finder handles Code Fragment }
                                          { Manager errors}
getstaltFinderHasClippings    = 6;        {Finder supports Drag Manager }
                                          { cliping files}

{gestaltFindFolderAttr response bits}
gestaltFindFolderPresent      = 0;        {FindFolder available}

{gestaltFontMgrAttr response values}
gestaltOutlineFonts           = 0;        {outline fonts supported}

{gestaltFPUType response values}
gestaltNoFPU                  = 0;        {no FPU}
gestalt68881                  = 1;        {Motorola 68881 FPU}
gestalt68882                  = 2;        {Motorola 68882 FPU}
gestalt68040FPU               = 3;        {built-in 68040 }
                                          { floating-point processing}

{gestaltFSAttr response bits}
gestaltFullExtFSDispatching   = 0;        {new HFSDispatch available}
gestaltHasFSSpecCalls         = 1;        {has FSSpec calls}
gestaltHasFileSystemManager   = 2;        {has File System Manager}
gestaltHasFileSystemManager   = 3;        {supports dynamic loading}
gestaltFSSupports4GBVols      = 4;        {supports 4 gigabyte volume}
gestaltHasExtendedDiskInit    = 6;        {has extended disk }
                                          { initialization calls}
```

```
{gestaltFXfrMgrAttr response bits}
gestaltFXfrMgrPresent          = 0;       {File Transfer Manager present}
gestaltFXfrMgrMultiFile        = 1;       {supports FTSend and FTReceive}
gestaltFXfrMgrErrorString      = 2;       {supports FTGetErrorString}

{gestaltHelpMgrAttr response bits}
gestaltHelpMgrPresent          = 0;       {Help Manager present}

{gestaltIconUtilitiesAttr response value}
gestaltIconUtilitiePresents    = 0;       {Icon Utilities are present}

{gestaltKeyboardType response values}
gestaltMacKbd                  = 1;       {Macintosh}
gestaltMacAndPad               = 2;       {Macintosh with keypad}
gestaltMacPlusKbd              = 3;       {Macintosh Plus}
gestaltExtADBKbd               = 4;       {extended ADB}
gestaltStdADBKbd               = 5;       {standard ADB}
gestaltPrtblADBKbd             = 6;       {Portable ADB}
gestaltPrtblISOKbd             = 7;       {Portable ISO ADB}
gestaltStdISOADBKbd            = 8;       {ISO standard ADB}
gestaltExtISOADBKbd            = 9;       {ISO extended ADB}
gestaltADBKbdII                = 10;      {ADB II}
gestaltADBISOKbdII             = 11;      {ISO ADB II}
gestaltPwrBookADBKbd           = 12;      {PowerBook ADB}
gestaltPwrBookISOADBKbd        = 13;      {PowerBook ISO ADB}
gestaltAppleAdjustKeypad       = 14;      {Adjustable Keypad}
gestaltAppleAdjustADBKbd       = 15;      {Adjustable ADB}
gestaltAppleAdjustISOKbd       = 16;      {Adjustable ISO}

{gestaltMiscAttr response bits}
gestaltScrollingThrottle       = 0;       {scrolling throttle is on}
gestaltSquareMenuBar           = 2;       {menu bar is square}

{gestaltMMUType response values}
gestaltNoMMU                   = 0;       {no MMU}
gestaltAMU                     = 1;       {Mac II address management unit}
gestalt68851                   = 2;       {Motorola 68851 PMMU}
gestalt68030MMU                = 3;       {built-in 68030 MMU}
gestalt68040MMU                = 4;       {built-in 68040 MMU}
gestaltEMMU1                   = 5;       {emulated MMU type 1}

{gestaltNativeCPUtype response values}
gestaltCPU68000                = $000;    {Macintosh 68000 CPU}
gestaltCPU68010                = $001;    {Macintosh 68010 CPU}
gestaltCPU68020                = $002;    {Macintosh 68020 CPU}
```

```
gestaltCPU68030              = $003;     {Macintosh 68030 CPU}
gestaltCPU68040              = $004;     {Macintosh 68040 CPU}
gestaltCPU601                = $101;     {PowerPC 601 CPU}

{gestaltNotificationMgrAttr response bits}
gestaltNotificationPresent   = 0;        {Notification Manager present}

{gestaltOSAttr response bits}
gestaltSysZoneGrowable       = 0;        {system heap can grow}
gestaltLaunchCanReturn       = 1;        {can return from launch}
gestaltLaunchFullFileSpec    = 2;        {LaunchApplication available}
gestaltLaunchControl         = 3;        {Process Manager available}
gestaltTempMemSupport        = 4;        {temporary memory support }
                                         { available}
gestaltRealTempMemory        = 5;        {temporary memory handles are }
                                         { real}
gestaltTempMemTracked        = 6;        {temporary memory handles are}
                                         { tracked}

{gestaltParityAttr response bits}
gestaltHasParityCapability   = 0;        {machine can check parity}
gestaltParityEnabled         = 1;        {parity RAM is installed}

{gestaltPCXAttr response bits}
gestaltPCXHas8and16BitFat    = 0;        {PC exchange supports both }
                                         { 8 and 16 bit FATs}
gestaltPCXHasProDOS          = 1;        {PC exchange supports ProDos}

{gestaltPopupAttr response bits}
gestaltPopupPresent          = 0;        {pop-up 'CDEF' is present}

{gestaltPowerMgrAttr response bits}
gestaltPMgrExists            = 0;        {Power Manager is present}
gestaltPMgrCPUIdle           = 1;        {CPU can idle}
gestaltPMgrSCC               = 2;        {Power Manager can stop SCC }
                                         { clock}
gestaltPMgrSound             = 3;        {Power Manager can turn off }
                                         { sound power}
gestaltPMgrDispatchExists    = 4;        {Power Manager dispatch exists}

{gestaltPPCToolboxAttr response masks}
gestaltPPCToolboxPresent     = $0000;    {PPC Toolbox is present;
                                         { PPCInit has been called}
gestaltPPCSupportsRealTime   = $1000;    {supports real-time delivery}
gestaltPPCSupportsIncoming   = $0001;    {accepts sessions from remote }
```

```
                                        { computers}
gestaltPPCSupportsOutGoing    = $0002;  {can initiate sessions with }
                                        { remote computers}

{gestaltProcessorType response values}
gestalt68000                  = 1;      {68000 microprocessor}
gestalt68010                  = 2;      {68010 microprocessor}
gestalt68020                  = 3;      {68020 microprocessor}
gestalt68030                  = 4;      {68030 microprocessor}
gestalt68040                  = 5;      {68040 microprocessor}

{gestaltQuickdrawFeatures response bits}
gestaltHasColor               = 0;      {Color QuickDraw present}
gestaltHasDeepGWorlds         = 1;      {graphics worlds can be deeper }
                                        { than 1 bit}
gestaltHasDirectPixMaps       = 2;      {PixMaps can be direct }
                                        { (16- or 32-bit)}
gestaltHasGrayishTextOr       = 3;      {supports text mode }
                                        { grayishTextOr}
gestaltSupportsMirroring      = 4;      {supports video mirroring }
                                        { using the Display Manager}

{gestaltQuickdrawVersion response values}
gestaltOriginalQD             = $000;   {original 1-bit QuickDraw}
gestalt8BitQD                 = $100;   {8-bit QuickDraw}
gestalt32BitQD                = $200;   {32-Bit QuickDraw vers. 1.0}
gestalt32BitQD11              = $210;   {32-Bit QuickDraw vers. 1.1}
gestalt32BitQD12              = $220;   {32-Bit QuickDraw vers. 1.2}
gestalt32BitQD13              = $230;   {32-Bit QuickDraw vers. 1.3}

{gestaltRealtimeAttr response bits}
gestaltRealtimeMgrPresent     = 0;      {Realtime Manager present}

{gestaltResourceMgrAttr response bits}
gestaltPartialRsrcs           = 0;      {partial resources supported}

{gestaltScrapMgrAttr response bits}
gestaltScrapMgrTranslationAware
                              = 0;      {aware of Translation Manager}
gestaltTranslationMgrHintOrder
                              = 1;      {hint order reversal present}
```

```
{gestaltSerialAttr response bits}
gestaltHasGPIaToDCDa          = 0;       {GPI connected to DCD on port A}
gestaltHasGPIaToRTxCa         = 1;       {GPI connected to RTxC on }
                                         { port A}
gestaltHasGPIaToDCDb          = 2;       {GPI connected to DCD on port B}

{gestaltSoundAttr response bits}
gestaltStereoCapability       = 0;       {stereo capability present}
gestaltStereoMixing           = 1;       {stereo mixing on internal }
                                         { speaker}
gestaltSoundIOMgrPresent      = 3;       {sound input routines present}
gestaltBuiltInSoundInput      = 4;       {built-in input device present}
gestaltHasSoundInputDevice    = 5;       {sound input device present}
gestaltPlayAndRecord          = 6;       {built-in hardware can play }
                                         { and record simultaneously}
getstalt16BitSoundIO          = 7;       {sound hardware can play and }
                                         { record 16-bit samples}
getstaltStereoInput           = 8;       {sound hardware can }
                                         { record steore}
getstaltSndPlayDoubleBuffer   = 10;      {SndPlayDouble buffer present}
getstaltMultiChannels         = 11;      {multiple channel support}
getstalt16BitAudioSuuport     = 12;      {16-bit audio data supported}

{gestaltSpeechAttr response bits}
gestaltSpeechMgrPresent       = 0;       {Speech Manager present}
gestaltSpeechHasPPCGlue       = 1;       {Speech Manager has native PPC }
                                         { glue for API}

{gestaltStandardFileAttr response bits}
getaltStandardFile58          = 0;       {has functions new with 7.0}
gestaltStandardFileTranslationAware
                              = 1;       {aware of Translation Manager}
gestaltStandardFileHasColorIcons
                              = 2;       {dialog boxes use small color }
                                         { icons}

{gestaltStdNBPAttr response bits}
gestaltStdNBPPresent          = 0;       {StandardNBP is present}

{gestaltSysArchitecture response bits}
gestalt68k                    = 1;       {MC680x0 architecture}
gestaltPowerPC                = 2;       {PowerPC architecture}

{gestaltTEAttr response bits}
gestaltTEHasGetHiliteRgn      = 0;       {TextEdit has TEGetHiliteRgn}
```

```
   {gestaltTermMgrAttr response bits}
   gestaltTermMgrPresent         = 0;        {Terminal Manager present}
   gestaltTermMgrErrorString     = 2;        {supports error string }
                                             { function}

   {gestaltTextEditVersion response values}
   gestaltTE1                    = 1;        {in MacIIci ROM}
   gestaltTE2                    = 2;        {with 6.0.4 scripts on Mac IIci}
   gestaltTE3                    = 3;        {with 6.0.4 scripts on other }
                                             { machines}
   gestaltTE4                    = 4;        {in 6.0.5 and 7.0}
   gestaltTE5                    = 5;        {TextWidthHook available}

   {gestaltThreadMgrAttr response bits}
   gestaltThreadMgrPresent       = 0;        {Thread Manger present}
   gestaltSpecificMatchSupport   = 1;        {Thread Manager supports }
                                             { exact match creation option}

   {gestaltTimeMgrVersion response values}
   gestaltStandardTimeMgr        = 1;        {standard Time Manager}
   gestaltRevisedTimeMgr         = 2;        {revised Time Manager}
   gestaltExtendedTimeMgr        = 3;        {extended Time Manager}

   {getstaltTranslationAttr response codes}
   gestaltTranslationMgrExists   = 0;        {Translation Manager present}

   {gestaltVMAttr response bits}
   gestaltVMPresent              = 0;        {virtual memory present}
```

Informational Selector Response Values

```
CONST
   {gestaltHardwareAttr response bits}
   gestaltHasVIA1                = 0;        {has VIA1 chip}
   gestaltHasVIA2                = 1;        {has VIA2 chip}
   gestaltHasASC                 = 3;        {has Apple sound chip}
   gestaltHasSCC                 = 4;        {has SCC}
   gestaltHasSCSI                = 7;        {has SCSI}
   gestaltHasSoftPowerOff        = 19;       {capable of software power off}
   gestaltHasSCSI961             = 21;       {has 53C96 SCSI on internal bus}
   gestaltHasSCSI962             = 22;       {has 53C96 SCSI on external bus}
   gestaltHasUniversalROM        = 24;       {has universal ROM}
```

```
{gestaltMachineType response values}
gestaltClassic              = 1;     {Macintosh 128K}
gestaltMacXL                = 2;     {Macintosh XL}
gestaltMac512KE             = 3;     {Macintosh 512K enhanced}
gestaltMacPlus              = 4;     {Macintosh Plus}
gestaltMacSE                = 5;     {Macintosh SE}
gestaltMacII                = 6;     {Macintosh II}
gestaltMacIIx               = 7;     {Macintosh IIx}
gestaltMacIIcx              = 8;     {Macintosh IIcx}
gestaltMacSE030             = 9;     {Macintosh SE/30}
gestaltPortable             = 10;    {Macintosh Portable}
gestaltMacIIci              = 11;    {Macintosh IIci}
gestaltMacIIfx              = 13;    {Macintosh IIfx}
gestaltMacClassic           = 17;    {Macintosh Classic}
gestaltMacIIsi              = 18;    {Macintosh IIsi}
gestaltMacLC                = 19;    {Macintosh LC}
gestaltQuadra900            = 20;    {Macintosh Quadra 900}
gestaltPowerBook170         = 21;    {Macintosh PowerBook 170}
gestaltQuadra700            = 22;    {Macintosh Quadra 700}
gestaltClassicII            = 23;    {Macintosh Classic II}
gestaltPowerBook100         = 24;    {Macintosh PowerBook 100}
gestaltPowerBook140         = 25;    {Macintosh PowerBook 140}
gestaltQuadra950            = 26;    {Macintosh Quadra 950}
gestaltMacLCIII             = 27;    {Macintosh LC III}
gestaltPowerBookDuo210      = 29;    {Macintosh PowerBook Duo 210}
gestaltMacCentris650        = 30;    {Macintosh Centris 650}
gestaltPowerBookDuo230      = 32;    {Macintosh PowerBook Duo 230}
gestaltPowerBook180         = 33;    {Macintosh PowerBook 180}
gestaltPowerBook160         = 34;    {Macintosh PowerBook 160}
gestaltMacQuadra800         = 35;    {Macintosh Quadra 800}
gestaltMacLCII              = 37;    {Macintosh LC II}
gestaltPowerBookDuo250      = 38;    {Macintosh PowerBook Duo 230}
gestaltMacIIvi              = 44;    {Macintosh IIvi}
gestaltPerforma600          = 45;    {Macintosh Performa 600}
gestaltMacIIvx              = 48;    {Macintosh IIvx}
gestaltMacColorClassic      = 49;    {Macintosh Color Classic}
gestaltPowerBook165c        = 50;    {Macintosh PowerBook 165c}
gestaltMacCentris610        = 52;    {Macintosh Centris 610}
gestaltMacQuadra610         = 53;    {Macintosh Quadra 610}
gestaltPowerBook145         = 54;    {Macintosh PowerBook 145}
getstaltMacLC520            = 56;    {Macintosh LC 520}
getstaltMacCentris660AV     = 60;    {Macintosh Centris 660 AV}
getstaltPowerBook180c       = 71;    {Macintosh PowerBook 180c}
```

```
getstaltPowerBookDuo270c      = 77;       {Maciintosh PowerBook Duo 270c}
getstaltMacQuadra840AV        = 78;       {Macintosh Quadra 840 AV}
getstaltPowerBook165          = 84;       {Macintosh PowerBook 165}
getstaltMacTV                 = 88;       {Macintosh TV}
getstaltMacLC475              = 89;       {Macintosh LC 475}
getstaltMacLC575              = 92;       {Macintosh LC 575}
getstaltMacQuadra605          = 94;       {Macintosh Quadra 605}

getstaltPowerMac8100_80       = 65;       {Power Macintosh 8100/80}
getstaltPowerMac6100_60       = 75;       {Power Macintosh 6100/60}
getstaltPowerMac7100_66       = 112;      {Power Macintosh 7100/66}

kMachineNameStrID             = -16395;   {'STR#' resource that }
                                          { contains machine names}
```

SysEnvirons Constants

```
CONST
   curSysEnvVers              = 2;        {current SysEnvirons version}

   {machine types}
   envXL                      = -2;       {Macintosh XL}
   envMac                     = -1;       {Macintosh with 64K ROM}
   envMachUnknown             = 0;        {unknown model, after }
                                          { Macintosh IIfx}
   env512KE                   = 1;        {Macintosh 512K enhanced}
   envMacPlus                 = 2;        {Macintosh Plus}
   envSE                      = 3;        {Macintosh SE}
   envMacII                   = 4;        {Macintosh II}
   envMacIIx                  = 5;        {Macintosh IIx}
   envMacIIcx                 = 6;        {Macintosh IIcx}
   envSE30                    = 7;        {Macintosh SE30}
   envPortable                = 8;        {Macintosh Portable}
   envMacIIci                 = 9;        {Macintosh IIci}
   envMacIIfx                 = 11;       {Macintosh IIfx}

   {system environment record microprocessor codes}
   envCPUUnknown              = 0;        {unknown microprocessor}
   env68000                   = 1;        {68000 microprocessor}
   env68010                   = 2;        {68010 microprocessor}
   env68020                   = 3;        {68020 microprocessor}
   env68030                   = 4;        {68030 microprocessor}
   env68040                   = 5;        {68040 microprocessor}
```

```
{system environment record keyBoardType codes}
envUnknownKbd                = 0;      {Macintosh Plus with keypad}
envMacKbd                    = 1;      {Macintosh}
envMacAndPad                 = 2;      {Macintosh with keypad}
envMacPlusKbd                = 3;      {Macintosh Plus}
envAExtendKbd                = 4;      {Apple extended}
envStandADBKbd               = 5;      {standard ADB}
envPrtblADBKbd               = 6;      {Macintosh Portable ADB}
envPrtblISOKbd               = 7;      {Macintosh Portable ISO}
envStdISOADBKbd              = 8;      {standard ISO ADB}
envExtISOADBKbd              = 9;      {extended ISO ADB}
```

Data Types

```
TYPE SysEnvRec =                 {system environment record}
RECORD
    environsVersion: Integer;    {SysEnvirons version number}
    machineType:     Integer;    {Macintosh model code}
    systemVersion:   Integer;    {System file version number}
    processor:       Integer;    {microprocessor type code}
    hasFPU:          Boolean;    {floating-point unit flag}
    hasColorQD:      Boolean;    {Color QuickDraw flag}
    keyBoardType:    Integer;    {keyboard type code}
    atDrvrVersNum:   Integer;    {AppleTalk driver version number}
    sysVRefNum:      Integer;    {working directory reference number of }
                                 { folder or volume containing open }
                                 { System file}
END;
```

Gestalt Manager Routines

Getting Information About the Operating Environment

```
FUNCTION Gestalt           (selector: OSType;
                            VAR response: LongInt): OSErr;
FUNCTION SysEnvirons       (versionRequested: Integer;
                            VAR theWorld: SysEnvRec): OSErr;
```

Adding a Selector Code

```
FUNCTION NewGestalt          (selector: OSType;
                              gestaltFunction: SelectorFunctionUUP): OSErr;
```

Modifying a Selector Function

```
FUNCTION ReplaceGestalt      (selector: OSType;
                              gestaltFunction: SelectorFunctionUUP;
                              VAR oldGestaltFunction: SelectorFunctionUUP)
                              : OSErr;
```

Application-Defined Routines

```
FUNCTION MySelectorFunction
                             (selector: OSType; VAR response: LongInt)
                              : OSErr;
```

C Summary

Constants

Environmental Selector Codes

```
#define gestaltAddressingModeAttr   'addr'   /*addressing-mode attributes*/
#define gestaltAliasMgrAttr         'alis'   /*Alias Manager attributes*/
#define gestaltAppleEventsAttr      'evnt'   /*Apple events attributes*/
#define gestaltAppleTalkVersion     'atlk'   /*old format AppleTalk version*/
#define gestaltATalkVersion         'atkv'   /*new format AppleTalk version*/
#define gestaltAUXVersion           'a/ux'   /*A/UX version, if present*/
#define gestaltCFMAttr              'cfrg'   /*Code Fragment Manager attr*/
#definegestaltCloseViewAttr         'BSDa'   /*CloseView attributes*/
#define gestaltComponentMgr         'cpnt'   /*Component Manager version*/
#define gestaltCompressionMgr       'icmp'   /*Image Compression Manager */
                                             /* version*/
#define gestaltConnMgrAttr          'conn'   /*Connection Manager attr*/
#define gestaltCRMAttr              'crm '   /*Comm Resource Manager attr*/
#define gestaltCTBVersion           'ctbv'   /*Comm Toolbox version*/
#define gestaltDBAccessMgrAttr      'dbac'   /*Data Access Manager attr*/
#define gestaltDictionaryMgrAttr    'dict'   /*Dictionary Manager attr*/
#define gestaltDisplayMgrAttr       'dply'   /*Display Manager attributes*/
#define gestaltDisplayMgrVers       'dplv'   /*Display Manager version*/
```

```
#define gestaltDITLExtAttr          'ditl'   /*Dialog Manager extensions*/
#define gestaltDragMgrAttr          'drag'   /*Drag Manager attributes*/
#define gestaltEasyAccessAttr       'easy'   /*Easy Access attributes*/
#define gestaltEditionMgrAttr       'edtn'   /*Edition Manager attributes*/
#define gestaltExtToolboxTable      'xttt'   /*Toolbox trap dispatch table*/
#define gestaltFinderAttr           'fndr'   /*Finder attributes*/
#define gestaltFindFolderAttr       'fold'   /*FindFolder attributes*/
#define gestaltFirstSlotNumber      'slt1'   /*first physical slot*/
#define gestaltFontMgrAttr          'font'   /*Font Manager attributes*/
#define gestaltFPUType              'fpu '   /*floating-point unit type*/
#define gestaltFSAttr               'fs  '   /*file system attributes*/
#define gestaltFXfrMgrAttr          'fxfr'   /*File Transfer Manager attr*/
#define gestaltHelpMgrAttr          'help'   /*Help Manager attributes*/
#define gestaltKeyboardType         'kbd '   /*keyboard type code*/
#define gestaltLogicalPageSize      'pgsz'   /*logical page size*/
#define gestaltLogicalRAMSize       'lram'   /*logical RAM size*/
#define gestaltLowMemorySize        'lmem'   /*size of low memory*/
#define gestaltMiscAttr             'misc'   /*miscellaneous attributes*/
#define gestaltMixedModeVersion     'mixd'   /*MixedMode version*/
#define gestaltMMUType              'mmu '   /*MMU type*/
#define gestaltNativeCPUtype        'cput'   /*Native CPU type*/
#define gestaltNotificationMgrAttr  'nmgr'   /*Notification Manager attr*/
#define gestaltNuBusConnectors      'sltc'   /*NuBus connector bitmap*/
#define getstaltNuBusSlotCount      'nubs'   /*count of logical NuBus slots*/
#define gestaltOSAttr               'os  '   /*Operating System attributes*/
#define gestaltOSTable              'ostt'   /*base address of Operating */
                                             /* System trap dispatch table*/
#define gestaltParityAttr           'prty'   /*parity attributes*/
#define gestaltPCXAttr              'pcxg'   /*PC exchange attributes*/
#define gestaltPhysicalRAMSize      'ram '   /*physical RAM size*/
#define gestaltPopupAttr            'pop!'   /*pop-up 'CDEF' attributes*/
#define gestaltPowerMgrAttr         'powr'   /*Power Manager attributes*/
#define gestaltPPCToolboxAttr       'ppc '   /*PPC Toolbox attributes*/
#define gestaltProcessorType        'proc'   /*microprocessor type code*/
#define gestaltQuickdrawFeatures    'qdrw'   /*QuickDraw features*/
#define gestaltQuickdrawVersion     'qd  '   /*QuickDraw version*/
#define gestaltQuickTime            'qtim'   /*QuickTime version*/
#define gestaltRealtimeAttr         'rtmr'   /*Realtime Manager attributes*/
#define gestaltResourceMgrAttr      'rsrc'   /*Resource Manager attributes*/
#define gestaltScrapMgrAttr         'scra'   /*Scrap Manager attributes*/
#define gestaltScriptCount          'scr#'   /*number of active script */
                                             /* systems*/
#define gestaltScriptMgrVersion     'scri'   /*Script Manager version*/
```

```
#define gestaltSerialAttr           'ser '   /*serial hardware attributes*/
#define gestaltSoundAttr            'snd '   /*sound attributes*/
#define gestaltSpeechAttr           'ttsc'   /*Speech Manager attributes*/
#define gestaltStandardFileAttr     'stdf'   /*Standard File attributes*/
#define gestaltStdNBPAttr           'nlup'   /*StandardNBP attributes*/
#define gestaltSysArchitecture      'sysa'   /*native system architecture*/
#define gestaltTEAttr               'teat'   /*TextEdit attributes*/
#define gestaltTermMgrAttr          'term'   /*Terminal Manager attributes*/
#define gestaltTextEditVersion      'te  '   /*TextEdit version code*/
#define gestaltThreadMgrAttr        'thds'   /*Thread Manager attributes*/
#define gestaltTimeMgrVersion       'tmgr'   /*Time Manager version code*/
#define gestaltToolboxTable         'tbtt'   /*base address of Toolbox */
                                             /* trap dispatch table*/
#define gestaltTranslationAttr      'xlat'   /*Translation Manager */
                                             /* attributes*/
#define gestaltTSMgrVersion         'tsmv'   /*Text Services Manager */
                                             /* version*/
#define getstaltIconUtilities       'icon'   /*Icon Utilities attributes*/
#define gestaltVersion              'vers'   /*Gestalt version*/
#define gestaltVMAttr               'vm  '   /*virtual memory attributes*/
```

Informational Selector Codes

```
#define gestaltHardwareAttr         'hdwr'   /*hardware attributes*/
#define gestaltMachineIcon          'micn'   /*machine 'ICON'/'cicn' res ID*/
#define gestaltMachineType          'mach'   /*Macintosh model code*/
#define gestaltROMSize              'rom '   /*ROM size*/
#define gestaltROMVersion           'romv'   /*ROM version*/
#define gestaltSystemVersion        'sysv'   /*System file version number*/
```

Environmental Selector Response Values

```
enum {
   /*gestaltAddressingModeAttr response bits*/
   gestalt32BitAddressing         = 0,     /*booted in 32-bit mode*/
   gestalt32BitSysZone            = 1,     /*32-bit compatible system */
                                           /* zone*/
   gestalt32BitCapable            = 2      /*machine is 32-bit capable*/
};

enum {
   /*gestaltAliasMgrAttr response bits*/
   gestaltAliasMgrPresent         = 0,     /*Alias Manager present*/
```

```
   gestaltAliasMgrSupportsRemoteAppletalk    /*Alias Manager knows about */
                                    = 1      /* remote Appletalk*/
};

enum {
   /*gestaltAppleEventsAttr response bits*/
   gestaltAppleEventsPresent        = 0,     /*Apple Events available*/
   gestaltScriptingSupport          = 1,
   gestaltOSLInSystem               = 2      /*OSL in system*/
};

enum {
   /*gestaltATalkVersion release stage constants*/
   development                      = $20,      /*development*/
   alpha                            = $40,      /*alpha*/
   beta                             = $60,      /*beta*/
   final                            = $80,      /*final*/
   release                          = $80       /*release*/
};

enum {
   /*gestaltCFMAttr response bits*/
   gestaltCFMPresent                = 0      /*Code Fragment Manager */
                                             /* present*/
};

enum {
   /*gestaltCloseViewAttr response bits*/
   gestaltCloseViewEnabled          = 0,     /*CloseView enabled*/
   gestaltCloseViewDisplayMgrFriendly
                                    = 1      /*CloseView compatible with */
                                             /* Display Manger*/
};

enum {
   /*gestaltConnMgrAttr response bits*/
   gestaltConnMgrPresent            = 0,     /*Connection Manager present*/
   gestaltConnMgrCMSearchFix        = 1,     /*CMAddSearch fix present*/
   gestaltConnMgrErrorString        = 2,     /*has CMGetErrorString*/
   gestaltConnMgrMultiAsyncIO       = 3      /*has CMNewIOPB, */
                                             /* CMDisposeIOPB, CMPBRead, */
                                             /* CMPBWrite, CMPBIOKill*/
};
```

```
enum {
   /*gestaltCRMAttr response bits*/
   gestaltCRMPresent               = 0,     /*Comm Resource Manager */
                                            /* present*/
   gestaltCRMPersistentFix         = 1,     /*fix for persistent tools*/
   gestaltCRMToolRsrcCalls         = 2      /*tool resource calls */
                                            /* available*/
};

enum {
   /*gestaltDBAccessMgrAttr response bits*/
   gestaltDBAccessMgrPresent       = 0      /*Data Access Manager present*/
};

enum {
   /*gestaltDictionaryMgrAttr response bits*/
   gestaltDictionaryMgrPresent     = 0      /*Dictionary Manager present*/
};

enum {
   /*gestaltDisplayMgrAttr response bits*/
   gestaltDisplayMgrPresent        = 0      /*Display Manager Present*/
};

enum {
   /*gestaltDITLExtAttr response bits*/
   gestaltDITLExtPresent           = 0      /*Dialog Manager extensions */
                                            /* present*/
};

enum {
   /*gestaltDragMgrAttr response bits*/
   gestaltDragMgrPresent           = 0      /*Drag Manager present*/
};

enum {
   /*gestaltEasyAccessAttr response bits*/
   gestaltEasyAccessOff            = 0,     /*Easy Access present but off*/
   gestaltEasyAccessOn             = 1,     /*Easy Access on*/
   gestaltEasyAccessSticky         = 2,     /*Easy Access sticky*/
   gestaltEasyAccessLocked         = 3      /*Easy Access locked*/
};
```

```
enum {
   /*gestaltEditionMgrAttr response bits*/
   gestaltEditionMgrPresent         = 0,     /*Edition Manager present*/
   gestaltEditionMgrTranslationAware = 1     /*Edition Manager aware of */
                                             /* Translation Manager*/
};

enum {
   /*gestaltFinderAttr response bits*/
   gestaltFinderDropEvent        = 0,        /*Finder recognizes drop event*/
   gestaltFinderMagicPlacement   = 1,        /*Finder supports magic icon */
                                             /* placement*/
   gestaltFinderCallsAEProcess   = 2,        /*Finder calls */
                                             /* AEProcessAppleEvent*/
   gestaltFinderOSLCompliantFinder
                                 = 3,        /*Finder is scriptable and */
                                             /* recordable*/
   getstaltFinderSupports4GBVolumes
                                 = 4,        /*Finder handles 4GB volumes*/
   getstaltFinderHandlesCFMFailures
                                 = 5,        /*Finder handles Code */
                                             / *Fragment Manager errors*/
   getstaltFinderHasClippings    = 6         /*Finder supports Drag */
                                             /* Manager cliping files*/

enum {
   /*gestaltFindFolderAttr response bits*/
   gestaltFindFolderPresent         = 0      /*FindFolder available*/
};

enum {
   /*gestaltFontMgrAttr response bits*/
   gestaltOutlineFonts              = 0      /*outline fonts supported*/
};

enum {
   /*gestaltFPUType response values*/
   gestaltNoFPU                     = 0,     /*no FPU*/
   gestalt68881                     = 1,     /*Motorola 68881 FPU*/
   gestalt68882                     = 2,     /*Motorola 68882 FPU*/
   gestalt68040FPU                  = 3      /*built-in 68040 */
                                             /* floating-point processing*/
};
```

```
enum {
   /*gestaltFSAttr response bits*/
   gestaltFullExtFSDispatching      = 0,     /*new HFSDispatch available*/
   gestaltHasFSSpecCalls            = 1,     /*has FSSpec calls*/
   gestaltHasFileSystemManager      = 2,     /*has File System Manager*/
   gestaltHasFileSystemManager      = 3,     /*supports dynamic loading*/
   gestaltFSSupports4GBVols         = 4,     /*supports 4 gigabyte volume*/
   gestaltHasExtendedDiskInit       = 6      /*has extended disk */
                                             /* initialization calls*/
};

enum {
   /*gestaltFXfrMgrAttr response bits*/
   gestaltFXfrMgrPresent            = 0,     /*File Transfer Manager */
                                             /* present*/
   gestaltFXfrMgrMultiFile          = 1,     /*supports FTSend and */
                                             /* FTReceive*/
   gestaltFXfrMgrErrorString        = 2      /*supports FTGetErrorString*/
};

enum {
   /*gestaltHelpMgrAttr response bits/*
   gestaltHelpMgrPresent            = 0      /*Help Manager present*/
};

enum {
   /*gestaltIconUtilitiesAttr response bits*/
   gestaltIconUtilitiesPresent      =  0     /*icon utilities present*/
};

enum {
   /*gestaltKeyboardType response values*/
   gestaltMacKbd                    = 1,     /*Macintosh*/
   gestaltMacAndPad                 = 2,     /*Macintosh with keypad*/
   gestaltMacPlusKbd                = 3,     /*Macintosh Plus*/
   gestaltExtADBKbd                 = 4,     /*extended ADB*/
   gestaltStdADBKbd                 = 5,     /*standard ADB*/
   gestaltPrtblADBKbd               = 6,     /*Portable ADB */
   gestaltPrtblISOKbd               = 7,     /*Portable ISO ADB*/
   gestaltStdISOADBKbd              = 8,     /*ISO standard ADB*/
   gestaltExtISOADBKbd              = 9,     /*ISO extended ADB*/
   gestaltADBKbdII                  = 10,    /*ADB II*/
   gestaltADBISOKbdII               = 11,    /*ISO ADB II*/
   gestaltPwrBookADBKbd             = 12,    /*PowerBook ADB*/
```

```
   gestaltPwrBookISOADBKbd          = 13,    /*PowerBook ISO ADB*/
   gestaltAppleAdjustKeypad         = 14,    /*Adjustable Keypad*/
   gestaltAppleAdjustADBKbd         = 15,    /*Adjustable ADB*/
   gestaltAppleAdjustISOKbd         = 16     /*Adjustable ISO*/
};

enum {
   /*gestaltMiscAttr return bits*/
   gestaltScrollingThrottle         = 0,     /*scrolling throttle is on*/
   gestaltSquareMenuBar             = 2      /*menu bar is square*/
};

enum {
   /*gestaltMMUType return values*/
   gestaltNoMMU                     = 0,     /*no MMU*/
   gestaltAMU                       = 1,     /*Mac II address management */
                                             /* unit*/
   gestalt68851                     = 2,     /*Motorola 68851 PMMU*/
   gestalt68030MMU                  = 3,     /*built-in 68030 MMU*/
   gestalt68040MMU                  = 4,     /*built-in 68040 MMU*/
   gestaltEMMU1                     = 5      /*emulated MMU type 1*/
};

enum {
   /*gestaltNativeCPUtype response values*/
   gestaltCPU68000                  = $000,  /*Macintosh 68000 CPU*/
   gestaltCPU68010                  = $001,  /*Macintosh 68010 CPU*/
   gestaltCPU68020                  = $002,  /*Macintosh 68020 CPU*/
   gestaltCPU68030                  = $003,  /*Macintosh 68030 CPU*/
   gestaltCPU68040                  = $004,  /*Macintosh 68040 CPU*/
   gestaltCPU601                    = $101,  /*PowerPC 601 CPU*/
};

enum {
   /*gestaltNotificationMgrAttr response bits*/
   gestaltNotificationPresent       = 0      /*Notification Manager present*/
};

enum {
   /*gestaltOSAttr response bits*/
   gestaltSysZoneGrowable           = 0,     /*system heap can grow*/
   gestaltLaunchCanReturn           = 1,     /*can return from launch*/
   gestaltLaunchFullFileSpec        = 2,     /*LaunchApplication available*/
   gestaltLaunchControl             = 3,     /*Process Manager available*/
```

```
   gestaltTempMemSupport            = 4,     /*temporary memory support */
                                             /* available*/
   gestaltRealTempMemory            = 5,     /*temporary memory handles */
                                             /* are real*/
   gestaltTempMemTracked            = 6,     /*temporary memory handles */
                                             /* are tracked*/
};

enum {
   /*gestaltParityAttr response bits*/
   gestaltHasParityCapability       = 0,     /*machine can check parity*/
   gestaltParityEnabled             = 1      /*parity RAM is installed*/
};

enum {
   /*gestaltPCXAttr response bits*/
   gestaltPCXHas8and16BitFat        = 0,     /*PC exchange supports both */
                                             /* 8 and 16 bit FATs*/
   /*gestaltPCXHasProDOS            = 1      /*PC exchange supports ProDos*/
};

enum {
   /*gestaltPopupAttr response bits*/
   gestaltPopupPresent              = 0      /*pop-up 'CDEF' is present*/
};

enum {
   /*gestaltPowerMgrAttr response bits*/
   gestaltPMgrExists                = 0,     /*Power Manager is present*/
   gestaltPMgrCPUIdle               = 1,     /*CPU can idle*/
   gestaltPMgrSCC                   = 2,     /*Power Manager can stop SCC */
                                             /* clock*/
   gestaltPMgrSound                 = 3,     /*Power Manager can turn off */
                                             /* sound power*/
   gestaltPMgrDispatchExists        = 4      /*Power Mgr dispatch exists*/
};

enum {
   /* gestaltPPCToolboxAttr response bits*/
   gestaltPPCToolboxPresent         = 0x0000, /*PPC Toolbox is present; */
                                              /* PPCInit has been called*/
   gestaltPPCSupportsRealTime       = 0x1000, /*supports real-time delivery*/
   gestaltPPCSupportsIncoming       = 0x0001, /*accepts sessions from */
                                              /* remote computers*/
```

```
   gestaltPPCSupportsOutGoing       = 0x0002  /*can initiate sessions with */
                                              /* remote computers*/
};

enum {
   /*gestaltProcessorType response values*/
   gestalt68000                     = 1,     /*68000 microprocessor*/
   gestalt68010                     = 2,     /*68010 microprocessor*/
   gestalt68020                     = 3,     /*68020 microprocessor*/
   gestalt68030                     = 4,     /*68030 microprocessor*/
   gestalt68040                     = 5      /*68040 microprocessor*/
};

enum {
   /*gestaltQuickdrawFeatures response bits*/
   gestaltHasColor                  = 0,     /*Color QuickDraw present*/
   gestaltHasDeepGWorlds            = 1,     /*graphics worlds can be */
                                             /* deeper than 1 bit*/
   gestaltHasDirectPixMaps          = 2,     /*PixMaps can be direct */
                                             /* (16- or 32-bit)*/
   gestaltHasGrayishTextOr          = 3,     /*supports text mode */
                                             /* grayishTextOr*/
   gestaltSupportsMirroring         = 4      /*supports video mirroring */
                                             /* using the Display Manager*/
};

enum {
   /*gestaltQuickdrawVersion response values*/
   gestaltOriginalQD                = 0x000, /*original 1-bit QuickDraw*/
   gestalt8BitQD                    = 0x100, /*8-bit QuickDraw*/
   gestalt32BitQD                   = 0x200, /*32-Bit QuickDraw vers. 1.0*/
   gestalt32BitQD11                 = 0x210, /*32-Bit QuickDraw vers. 1.1*/
   gestalt32BitQD12                 = 0x220, /*32-Bit QuickDraw vers. 1.2*/
   gestalt32BitQD13                 = 0x230  /*32-Bit QuickDraw vers. 1.3*/
};

enum {
   /*gestaltRealtimeAttr response bits*/
   gestaltRealtimeMgrPresent        = 0      /*Realtime Manager present*/
};
```

```
enum {
   /*gestaltResourceMgrAttr response bits*/
   gestaltPartialRsrcs              = 0      /*partial resources supported*/
};

enum {
   /*gestaltScrapMgrAttr response bits*/
   gestaltScrapMgrTranslationAware  = 0,     /*aware of Translation Manager*/
   gestaltTrasnlationMgrHintOrder   = 1      /*hint order reversal present*/
};

enum {
   /*gestaltSerialAttr response bits*/
   gestaltHasGPIaToDCDa             = 0,     /*GPI connected to DCD on */
                                             /* port A*/
   gestaltHasGPIaToRTxCa            = 1,     /*GPI connected to RTxC on */
                                             /* port A*/
   gestaltHasGPIbToDCDb             = 2      /*GPI connected to DCD on */
                                             /* port B*/
};

enum {
   /*gestaltSoundAttr response bits*/
   gestaltStereoCapability          = 0,     /*stereo capability present*/
   gestaltStereoMixing              = 1,     /*stereo mixing on internal */
                                             /* speaker*/
   gestaltSoundIOMgrPresent         = 3,     /*sound input routines present*/
   gestaltBuiltInSoundInput         = 4,     /*built-in input device */
                                             /* present*/
   gestaltHasSoundInputDevice       = 5,     /*sound input device present*/
   gestaltPlayAndRecord             = 6,     /*built-in hardware can play */
                                             /* and record simultaneously*/
   getstalt16BitSoundIO             = 7,     /*sound hardware can play and */
                                             /* record 16-bit samples*/
   getstaltStereoInput              = 8,     /*sound hardware can */
                                             /* record steore*/
   getstaltSndPlayDoubleBuffer      = 10,    /*SndPlayDouble buffer present*/
   getstaltMultiChannels            = 11,    /*multiple channel support*/
   getstalt16BitAudioSuuport        = 12     /*16-bit audio data supported*/
};

enum {
   /*gestaltSpeechAttr response bits*/
   gestaltSpeechMgrPresent          = 0,     /*Speech Manager present*/
```

```
   gestaltSpeechHasPPCGlue          = 1      /*Speech Manager has native *
                                             /* PPC glue for API*/
};

enum {
   /*gestaltStandardFileAttr response bits*/
   getaltStandardFile58                = 0,  /*has functions new with 7.0*/
   gestaltStandardFileTranslationAware = 1,  /*aware of Translation Manager*/
   gestaltStandardFileHasColorIcons    = 2   /*dialog boxes use small */
                                             /* color icons*/
};

enum {
   /*gestaltStdNBPAttr response bits*/
   gestaltStdNBPPresent             = 0      /*StandardNBP is present*/
};

enum {
   /*gestaltSysArchitecture response bits*/
   gestalt68k                       = 1,     /*MC680x0 architecture*/
   gestaltPowerPC                   = 2      /*PowerPC architecture*/
};

enum {
   /*gestaltTEAttr response bits*/
   gestaltTEHasGetHiliteRgn         = 0      /*TextEdit has TEGetHiliteRgn*/
};

enum {
   /*gestaltTermMgrAttr response bits*/
   gestaltTermMgrPresent            = 0,     /*Terminal Manager present*/
   gestaltTermMgrErrorString        = 2      /*supports error string */
                                             /* function*/
};

enum {
   /*gestaltTextEditVersion response codes */
   gestaltTE1                       = 1,     /*in MacIIci ROM*/
   gestaltTE2                       = 2,     /*with 6.0.4 scripts on */
                                             /* MacIIci*/
   gestaltTE3                       = 3,     /*with 6.0.4 scripts on*/
                                             /* other machines*/
   gestaltTE4                       = 4,     /*in 6.0.5 and 7.0*/
   gestaltTE5                       = 5      /*TextWidthHook available*/
};
```

```
enum {
   /*gestaltThreadMgrAttr response bits*/
   gestaltThreadMgrPresent          = 0,     /*Thread Manager present*/
   gestaltSpecificMatchSupports     = 1      /*Thread Manager supports */
                                             /* exact match creation option*/
};

enum {
   /*gestaltTimeMgrVersion response codes*/
   gestaltStandardTimeMgr           = 1,     /*standard Time Manager*/
   gestaltRevisedTimeMgr            = 2,     /*revised Time Manager*/
   gestaltExtendedTimeMgr           = 3      /*extended Time Manager*/
};

enum {
   /*getstaltTranslationAttr response codes*/
   gestaltTranslationMgrExists      = 0      /*Translation Manager present*/
};

enum {
   /*gestaltVMAttr response bits*/
   gestaltVMPresent                 = 0      /*virtual memory present*/
};
```

Informational Selector Response Values

```
enum {
   /*gestaltHardwareAttr response bits*/
   gestaltHasVIA1                   = 0,     /*has VIA1 chip*/
   gestaltHasVIA2                   = 1,     /*has VIA2 chip*/
   gestaltHasASC                    = 3,     /*has Apple Sound Chip*/
   gestaltHasSCC                    = 4,     /*has SCC*/
   gestaltHasSCSI                   = 7,     /*has SCSI*/
   gestaltHasSoftPowerOff           = 19,    /*capable of software power */
                                             /* off*/
   gestaltHasSCSI961                = 21,    /*has 53C96 SCSI on internal */
                                             /* bus*/
   gestaltHasSCSI962                = 22,    /*has 53C96 SCSI on external */
                                             /* bus*/
   gestaltHasUniversalROM           = 24     /*has universal ROM*/
};
```

```
enum {
   /*gestaltMachineType response codes*/
   gestaltClassic                  = 1,     /*Macintosh 128K*/
   gestaltMacXL                    = 2,     /*Macintosh XL*/
   gestaltMac512KE                 = 3,     /*Macintosh 512K enhanced*/
   gestaltMacPlus                  = 4,     /*Macintosh Plus*/
   gestaltMacSE                    = 5,     /*Macintosh SE*/
   gestaltMacII                    = 6,     /*Macintosh II*/
   gestaltMacIIx                   = 7,     /*Macintosh IIx*/
   gestaltMacIIcx                  = 8,     /*Macintosh IIcx*/
   gestaltMacSE030                 = 9,     /*Macintosh SE/30*/
   gestaltPortable                 = 10,    /*Macintosh Portable*/
   gestaltMacIIci                  = 11,    /*Macintosh IIci*/
   gestaltMacIIfx                  = 13,    /*Macintosh IIfx*/
   gestaltMacClassic               = 17,    /*Macintosh Classic*/
   gestaltMacIIsi                  = 18,    /*Macintosh IIsi*/
   gestaltMacLC                    = 19,    /*Macintosh LC*/
   gestaltQuadra900                = 20,    /*Macintosh Quadra 900*/
   gestaltPowerBook170             = 21,    /*Macintosh PowerBook 170*/
   gestaltQuadra700                = 22,    /*Macintosh Quadra 700*/
   gestaltClassicII                = 23,    /*Macintosh Classic II*/
   gestaltPowerBook100             = 24,    /*Macintosh PowerBook 100*/
   gestaltPowerBook140             = 25,    /*Macintosh PowerBook 140*/
   gestaltQuadra950                = 26,    /*Macintosh Quadra 950*/
   gestaltMacLCIII                 = 27,    /*Macintosh LC III*/
   gestaltPowerBook210             = 29,    /*Macintosh PowerBook Duo 210*/
   gestaltMacCentris650            = 30,    /*Macintosh Centris 650*/
   gestaltPowerBook230             = 32,    /*Macintosh PowerBook Duo 230*/
   gestaltPowerBook180             = 33,    /*Macintosh PowerBook 180*/
   gestaltPowerBook160             = 34,    /*Macintosh PowerBook 160*/
   gestaltMacQuadra800             = 35,    /*Macintosh Quadra 800*/
   gestaltMacLCII                  = 37,    /*Macintosh LC II*/
   gestaltPowerBookDuo250          = 38,    /*Macintosh PowerBook Duo 230*/
   gestaltMacIIvi                  = 44,    /*Macintosh IIvi*/
   gestaltPerforma600              = 45,    /*Macintosh Performa 600*/
   gestaltMacIIvx                  = 48,    /*Macintosh IIvx*/
   gestaltMacColorClassic          = 49,    /*Macintosh Color Classic*/
   gestaltPowerBook165c            = 50,    /*Macintosh PowerBook 165c*/
   gestaltMacCentris610            = 52,    /*Macintosh Centris 610*/
   gestaltMacQuadra610             = 53,    /*Macintosh Quadra 610*/
   gestaltPowerBook145             = 54,    /*Macintosh PowerBook 145*/
   getstaltMacLC520                = 56,    /*Macintosh LC 520*/
   getstaltMacCentris660AV         = 60,    /*Macintosh Centris 660 AV*/
```

```
   getstaltPowerBook180c          = 71,    /*Macintosh PowerBook 180c*/
   getstaltPowerBookDuo270c       = 77,    /*Macintosh PowerBook Duo 270c*/
   getstaltMacQuadra840AV         = 78,    /*Macintosh Quadra 840 AV*/
   getstaltPowerBook165           = 84,    /*Macintosh PowerBook 165*/
   getstaltMacTV                  = 88,    /*Macintosh TV*/
   getstaltMacLC475               = 89,    /*Macintosh LC 475*/
   getstaltMacLC575               = 92,    /*Macintosh LC 575*/
   getstaltMacQuadra605           = 94,    /*Macintosh Quadra 605*/

   getstaltPowerMac8100_80        = 65,    /*Power Macintosh 8100/80*/
   getstaltPowerMac6100_60        = 75,    /*Power Macintosh 6100/60*/
   getstaltPowerMac7100_66        = 112    /*Power Macintosh 7100/66*/

};

enum {
   kMachineNameStrID              = -16395 /*'STR#' resource that */
};                                         /* contains machine names*/
```

SysEnvirons Constants

```
enum {
   curSysEnvVers                  = 2      /*current SysEnvirons version*/
};

enum {
   /*machine types*/
   envXL                          = -2,    /*Macintosh XL*/
   envMac                         = -1,    /*Macintosh with 64K ROM*/
   envMachUnknown                 = 0,     /*unknown model, after */
                                           /* Macintosh IIfx*/
   env512KE                       = 1,     /*Macintosh 512K enhanced*/
   envMacPlus                     = 2,     /*Macintosh Plus*/
   envSE                          = 3,     /*Macintosh SE*/
   envMacII                       = 4,     /*Macintosh II*/
   envMacIIx                      = 5,     /*Macintosh IIx*/
   envMacIIcx                     = 6,     /*Macintosh IIcx*/
   envSE30                        = 7,     /*Macintosh SE30*/
   envPortable                    = 8,     /*Macintosh Portable*/
   envMacIIci                     = 9,     /*Macintosh IIci*/
   envMacIIfx                     = 11,    /*Macintosh IIfx*/
   envMacClassic                  = 15,    /*Macintosh Classic*/
   envMacIIsi                     = 16,    /*Macintosh IIsi*/
   envMacLC                       = 17,    /*Macintosh LC*/
```

```
   envMacQuadra900               = 18,    /*Macintosh Quadra 900*/
   envMacPowerBook170            = 19,    /*Macintosh PowerBook 170*/
   envMacQuadra700               = 20,    /*Macintosh Quadra 700*/
   envMacClassicII               = 21,    /*Macintosh Classic II*/
   envMacPowerBook100            = 22,    /*Macintosh PowerBook 100*/
   envMacPowerBook140            = 23,    /*Macintosh PowerBook 140*/
   envMacQuadra950               = 24,    /*Macintosh Quadra 950*/
   envMacLCII                    = 35,    /*Macintosh LC II*/
   envMacPowerBook145            = 52     /*Macintosh PowerBook 145*/
};

enum {
   /*CPU types*/
   envCPUUnknown                 = 0,     /*unknown microprocessor*/
   env68000                      = 1,     /*68000 microprocessor*/
   env68010                      = 2,     /*68010 microprocessor*/
   env68020                      = 3,     /*68020 microprocessor*/
   env68030                      = 4,     /*68030 microprocessor*/
   env68040                      = 5,     /*68040 microprocessor*/
};

enum {
   /*keyboard types*/
   envUnknownKbd                 = 0,     /*Macintosh Plus with keypad*/
   envMacKbd                     = 1,     /*Macintosh*/
   envMacAndPad                  = 2,     /*Macintosh with keypad*/
   envMacPlusKbd                 = 3,     /*Macintosh Plus*/
   envAExtendKbd                 = 4,     /*Apple extended*/
   envStandADBKbd                = 5,     /*standard ADB*/
   envPrtblADBKbd                = 6,     /*Macintosh Portable ADB*/
   envPrtblISOKbd                = 7,     /*Macintosh Portable ISO*/
   envStdISOADBKbd               = 8,     /*standard ISO ADB */
   envExtISOADBKbd               = 9      /*extended ISO ADB*/
};
```

Data Types

```
struct SysEnvRec {                                /*system environment record*/
    short            environsVersion;     /*SysEnvirons version number*/
    short            machineType;         /*Macintosh model code*/
    short            systemVersion;       /*System file version number*/
    short            processor;           /*microprocessor type code*/
    Boolean          hasFPU;              /*floating-point unit flag*/
```

```
    Boolean          hasColorQD;          /*Color QuickDraw flag*/
    short            keyBoardType;        /*keyboard type code*/
    short            atDrvrVersNum;       /*AppleTalk driver version number*/
    short            sysVRefNum           /*working-directory reference */
                                          /* number of folder or volume */
                                          /* containing open System file*/
};

typedef struct SysEnvRec SysEnvRec;
```

Gestalt Manager Routines

Getting Information About the Operating Environment

```
pascal OSErr Gestalt           (OSType selector, long *response);
pascal OSErr SysEnvirons       (short versionRequested, SysEnvRec *theWorld);
```

Adding a Selector Code

```
pascal OSErr NewGestalt        (OSType selector,
                                 SelectorFunctionUUP gestaltFunction);
```

Modifying a Selector Function

```
pascal OSErr ReplaceGestalt
                               (OSType selector,
                                SelectorFunctionUUP gestaltFunction,
                                SelectorFunctionUUP *oldGestaltFunction);
```

Application-Defined Routines

```
pascal OSErr MySelectorFunc
                               (OSType selector, long *response);
```

Assembly-Language Summary

Data Structures

SysEnvRec Data Structure

0	environsVersion	word	`SysEnvirons` version number
2	machineType	word	Macintosh model code
4	systemVersion	word	System file version number
6	processor	word	microprocessor type code
8	hasFPU	byte	floating-point unit flag
9	hasColorQD	byte	Color QuickDraw flag
10	keyBoardType	word	keyboard type code
12	atDrvrVersNum	word	AppleTalk driver version number
14	sysVRefNum	word	working-directory reference number of directory or volume containing open System file

Result Codes

`noErr`	0	No error
`memFullErr`	–108	Ran out of memory
`envNotPresent`	–5500	`SysEnvirons` trap not present
`envBadVers`	–5501	Nonpositive version number passed
`envVersTooBig`	–5502	Requested version of `SysEnvirons` not available
`gestaltUnknownErr`	–5550	Could not obtain the response
`gestaltUndefSelectorErr`	–5551	Undefined selector
`gestaltDupSelectorErr`	–5552	Selector already exists
`gestaltLocationErr`	–5553	Function not in system heap

System Error Handler

Contents

This chapter describes the System Error Handler. The System Error Handler assumes control of the system when a system error occurs and is also responsible for displaying certain alert boxes in response to a system startup. The System Error Handler displays an alert box when a system error occurs and manages display of the "Welcome to Macintosh" alert box and the disk-switch alert box.

This chapter explains what the Operating System does when a system error is encountered, describes the routine and resource that the System Error Handler uses when generating a system error alert box, and discusses how you can provide code that can help your application recover from an system error.

Although your application may call the routine provided by the System Error Handler, ordinarily there is no need to do so; this routine is primarily used by the Macintosh Operating System.

This chapter also contains a list of all currently defined system errors and the conditions under which they can arise.

About the System Error Handler

The System Error Handler employs a mechanism that allows for display of simple alert boxes even when the Control Manager, Dialog Manager, and Memory Manager might not be able to function properly. System Error Handler alert boxes can therefore be displayed at times when the Dialog Manager cannot be called. This mechanism is useful at two times. First, at system startup time, the Dialog Manager may not yet have been initialized. Second, after a system error occurs, using the Dialog Manager or Memory Manager may be impossible or cause a system crash.

Because the System Error Handler cannot use Dialog Manager resources to store representations of its alert boxes, it defines its own resource, the system error alert table resource**,** to store such information. This resource type is described in "The System Error Alert Table Resource" beginning on page 2-16. The **system alert table resource** defines for each system error the contents of the system alert box to be displayed. For example, depending on the system error that occurred, the system error alert box may contain one or more buttons, typically a Restart and a Continue button.

At system startup time, the System Error Handler presents the **system startup alert box**, shown in Figure 2-1.

Figure 2-1 The system startup alert box

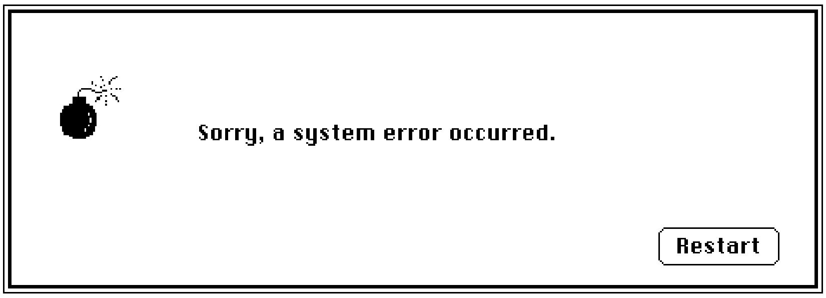
The system startup alert box can take different forms. In particular, if an error occurs during the startup process, the System Error Handler might inform the user of the error by displaying an additional line of information in the alert box. The System Error Handler also uses the system startup alert box to post special messages to inform the user about the status of the system. For example, in System 7 and later, if the user holds down the Shift key while starting up, system extensions are disabled, and the system startup alert box includes the message “Extensions off.” This is illustrated in Figure 2-2.

Figure 2-2 The system startup alert box when extensions have been disabled

Other messages that may be displayed at startup time include “Debugger Installed,” “Disassembler Installed,” and “System 7.1 needs more memory to start up.”

The System Error Handler also displays an alert box when the Operating System or some other software invokes the `SysError` procedure. Figure 2-3 illustrates a **system error alert box,** sometimes called a **bomb box.** The conditions under which a system error occur are described in the next section, “System Errors.”

Figure 2-3 The system error alert box



The system error alert box presents some information about the type of error that has occurred and also includes buttons to allow possible recovery from the error. The user may click the Restart button, in which case the System Error Handler attempts to restart the computer. (Such attempts are not always successful, and the computer may freeze, forcing the user to flip the power switch or depress the reset switch.) Some system error alert boxes have Continue buttons. If the user clicks the Continue button, the System Error Handler attempts to execute the application’s resume procedure. Resume procedures are discussed in “Resume Procedures” on page 2-11. If no resume procedure has been defined, then only the Restart button is available.

Note

The layout and form of the system error alert box have changed considerably in different versions of system software. In early versions of system software, there was always a Resume button, which had the same effect as the Continue button, but it was grayed out when no resume procedure was defined. The Resume and Restart buttons were both at the left of the alert box. In some versions of system software, information about the type of error was displayed at the bottom of the alert box, and the ID information may have been conveyed in words (“bus error”) instead of numbers (“ID = 1”). However, your application should not need to be familiar with the layout of the system error alert box. u

A close examination of the button in Figure 2-3 reveals that the button has a different appearance from that of buttons displayed by the Control Manager. This is because the System Error Handler does not use the Control Manager to create buttons. Instead, it draws the buttons itself and highlights them when the mouse is clicked within the button area.

System Errors

A **system error** is the result of the detection of a problem by the microprocessor or the Operating System. For example, if your application attempts to execute a system software routine that is not available on a certain Macintosh computer, the microprocessor detects the exception. The Operating System then calls the `SysError` procedure to produce a system error alert box. Similarly, the Operating System itself might detect a problem; for example, it might detect that a menu record that is needed has been purged. In this case, the Operating System calls `SysError` directly.

Your application can also call `SysError` if it detects that something that never should happen actually has happened. Ordinarily, it is more graceful for an application to use the Dialog Manager to warn the user that an error has occurred. You should call the `SysError` procedure only if there is reason to believe that an abnormal condition could prevent the Dialog Manager from working correctly. The Dialog Manager is described in the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials.*

Associated with each type of system error is a **system error ID.** This ID is typically presented to the user in the system error alert box. Although the system error IDs are meaningless to most users, a user can report the ID to you, thus possibly making it easier for you to track down the problem and provide the user with a solution.

Table 2-1 lists and briefly describes the system error IDs that are currently defined. Note, however, that sometimes system error IDs may be misleading. For example, your application might make an invalid memory reference that does not cause a system error immediately. However, the effects of that reference could cause another problem leading to a system error of a different type.

Note also that some system errors occur in the ordinary course of an application’s execution but are handled by the Operating System with no need to display an error message to the user. For example, when virtual memory is in operation and an application attempts to access memory that has been paged out, a bus error is generated. Because the Virtual Memory Manager intercepts the bus error and determines that memory needs to be paged in, this error is generated transparently to the user. If possible, when a system error occurs, the System Error Handler stops execution of the application that caused the error and displays an alert box with the message “Application has unexpectedly quit.” (See Figure 2-4 on page 2-12 for an example of this alert box.)

Table 2-1 System error IDs

ID and name	Explanation
1 (Bus error)	A memory reference was invalid. This is the most common type of system error. An application might have tried to access memory in another application's partition or in a portion of memory not accessible to the application. Typically, this error occurs if your application uses a handle or pointer reference that is no longer valid or was never valid. For example, if your application does not initialize a variable of type `Handle` or `Ptr` to the correct value and then tries to use that value as a memory reference, a bus error could occur. Or if you have made an error in performing pointer arithmetic, a bus error could occur. This error could also occur if your application attempts to access a block of memory that has been moved or disposed of. Once your application disposes of a block of memory, either directly or indirectly, all pointer and handle references to that block of memory are invalid and could cause bus errors. If your application dereferences a handle, calls a routine that could move or purge memory, and then relies on the master pointer value, a bus error could occur. See *Inside Macintosh: Memory* for more information. If your application is careless in using the Memory Manager's `BlockMove` procedure or another technique to copy bytes directly, data structures used by the Memory Manager could be altered and a bus error generated.
2 (Address error)	A reference to a word (2 bytes) or long word (4 bytes) was not on a word boundary. An address error is often simply a bus error in which the memory reference happens to be odd. Thus, any programming errors that could cause a bus error might result in an address error as well. Indeed, sometimes the same programming error can generate both types of errors if you execute the offending code several times. Address errors are often microprocessor-specific. That is, code that executes correctly on MC68030 microprocessors might generate an address error on MC68000 microprocessors. This is most likely to be a problem for assembly-language programmers.

continued

Table 2-1 System error IDs (continued)

ID and name	Explanation
3 (Illegal instruction)	The microprocessor attempted to execute an instruction not defined for that version of the microprocessor. This might occur if you set a compiler to generate MC68030 code and then attempt to execute that code on a MC68000 microprocessor. Attempting to execute PowerPC code on a MC680x0 microprocessor could also cause this problem. Typically, this problem occurs only if you are programming in assembly language or if your compiler generates illegal instructions. If your application (either intentionally or unintentionally) modifies its own code while executing, then this problem could also occur.
4 (Zero divide)	The microprocessor received a signed divide (`DIVS`) or unsigned divide (`DIVU`) instruction, but the divisor was 0. When you write code that performs the division operation, you should ensure that the divisor can never be 0, unless you are using Operating System or SANE numeric types that support division by 0.
5 (Check exception)	The microprocessor executed a check-register-against-bounds (`CHK`) instruction and detected an out-of-bounds value. If you are programming in a high-level language, this might occur if you have enabled range-checking and a value is out of range (for example, you attempt to access the sixth element of a five-element array).
6 (TrapV exception)	The microprocessor executed a trap-on-overflow (`TRAPV`) instruction and detected an overflow. If you are programming in a high-level language, this might occur if you have enabled integer-arithmetic overflow checking and an overflow occurs.
7 (Privilege violation)	The Macintosh computer was in a mode that did not allow execution of the specified microprocessor instruction. This should not happen because the Macintosh computer always runs in supervisor mode. However, if you are programming in assembly language, this error could occur if you execute an erroneous return-from-execution (`RTE`) instruction.
8 (Trace exception)	The trace bit in the status register is set. Debuggers use this error to force code execution to stop at a certain point. If you are programming in a high-level language, this system error should always be intercepted by your low-level debugger.
9 (A-line exception)	The trap dispatcher failed to execute the specified system software routine. This error might occur if you attempt to execute a Toolbox routine that is not defined in the version of the system software that is running.
10 (F-line exception)	Your application executed an illegal instruction.

Table 2-1 System error IDs (continued)

ID and name	Explanation
11 (Miscellaneous exception)	The microprocessor invoked an exception not covered by system error IDs 1 to 10. This exception might be generated in the case of a hardware failure.
12 (Unimplemented core routine)	The Operating System encountered an unimplemented trap number.
13 (Spurious interrupt)	The interrupt vector table entry for a particular level of interrupt is `NIL`. This error usually occurs with level 4, 5, 6, or 7 interrupts. Typically, this error should affect only developers of low-level device drivers, NuBus cards, and other expansion devices.
14 (I/O system error)	A Device Manager or Operating System queue operation failed. This might occur if the File Manager attempts to remove an entry from an I/O request queue, but the queue entry has an invalid queue type (perhaps the queue entry is unlocked). Or this might occur as a result of a call to `Fetch` or `Stash`, but the `dCtlQHead` field was `NIL`. This error can also occur if your driver has purged a needed device control entry (DCE).
15 (Segment loader error)	A call was made to load a code segment, but a call to `GetResource` to read the segment into memory failed. This could occur if your application attempts to load a segment that does not exist, or if your application attempts to load a segment but there is not enough memory for it in the application heap. When an attempt to load a code resource with resource ID 0 fails, a system error with ID 26 is generated instead.
16 (Floating-point error)	The halt bit in the floating-point environment word was set.
17–24 (Can't load package)	The Package Manager attempted to load a package into memory, but the call to `GetResource` failed. This could occur because the system file is corrupted, or because there is not enough memory for the package to be loaded. For example, if you call a List Manager routine when memory is very low, the `SysError` procedure could be executed.
25 (Out of memory)	The requested memory block could not be allocated in the heap because there is insufficient free space. Typically, a Toolbox routine generates this system error if it requires heap space to run but there is insufficient space. Your application should prevent this from occurring by ensuring that it always leaves enough memory for Toolbox operations. See *Inside Macintosh: Memory* for more details. You can also get this error if the Package Manager was unable to load the Apple Event Manager (Pack 8). See the chapter "Package Manager" in this book for an explanation of this error.

continued

Table 2-1 System error IDs (continued)

ID and name	Explanation
26 (Segment loader error)	A call was made to load a code segment with resource ID 0, but the call to `GetResource` failed. This usually occurs if your application attempts to execute a nonexecutable file. You can also get this error if the Package Manager was unable to load the Program-to-Program Communications (PPC) Toolbox package (Pack 9). See the chapter "Package Manager" in this book for an explanation of this error.
27 (File map destroyed)	The File Manager encountered a paradox. 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. The disk is probably corrupted.
28 (Stack overflow error)	The Operating System detected that the application's stack collided with its heap. This could happen when a deeply nested routine is executed or when interrupt routines use more stack space than available. If your application relies on recursion, it should monitor the size of the stack to prevent such an error from occurring. If this error occurs simply because your application attempted to execute a deeply nested routine, you can prevent this from occurring by increasing the minimum size of the stack at application startup. Because the size of the stack may differ from one Macintosh model to another, an application might encounter no problems on a Macintosh LC but crash on a Macintosh Plus, for example. For more information, see *Inside Macintosh: Memory*. You can also get this error if the Package Manager was unable to load the Edition Manager (Pack 11). See the chapter "Package Manager" in this book for an explanation of this error.
30 (Disk insertion required)	A necessary disk is not available. The System Error Handler responds to this error by requesting that the user insert the requested disk. Often, the user can cancel this alert box by pressing Command-period repeatedly; in certain circumstances, however, pressing Command-period repeatedly can lead to a system crash. You can also get this error if the Package Manager was unable to load the Data Access Manager (Pack 13). See the chapter "Package Manager" in this book for an explanation of this error.
31 (Wrong disk inserted)	The user inserted the incorrect disk in response to a disk-insertion request. The System Error Handler ejects the disk and allows the user to insert another. You can also get this error if the Package Manager was unable to load the Help Manager (Pack 14). See the chapter "Package Manager" in this book for an explanation of this error.

Table 2-1 System error IDs (continued)

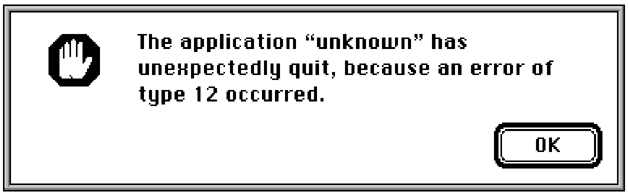
ID and name	Explanation
33 (Negative `zcbFree` value)	The Memory Manager's calculation of the number of bytes free in a heap zone (that is, the value of the `zcbFree` field) resulted in a negative number. Your application might have used up too much memory in the heap zone, or the heap is corrupted
41 (Finder not found)	The Operating System could not locate the Finder on the disk. The disk might be corrupted.
84 (Menu purged)	The Menu Manager attempted to access information about a menu, but the menu record was purged. You should ensure that all menus stored in your application's resource file are marked as unpurgeable.
100 (Can't mount system startup volume)	The Operating System could not mount the system startup volume and thus is unable to read the system resource file into memory. The startup volume could be corrupted or broken. Your application can force startup on another volume by clearing parameter RAM, as discussed in the chapter "Parameter RAM Utilities" in this book.
32767 (Default system error)	This is the default system error that executes when an undefined problem occurs. Your application can call the `SysError` procedure with this value.

Resume Procedures

The Operating System supports a mechanism that allows your application to resume execution after a system error if the user clicks the Continue button (or the Resume button in earlier versions of system software). When initializing the Dialog Manager using the `InitDialogs` procedure, your application passes a pointer to a resume procedure or passes `NIL` if no resume procedure is desired. A resume procedure takes no parameters.

In general, you should not write code to allow an application to continue to execute normally after a system error has occurred. Because current versions of system software allow multiple applications to be open at once, a system error could affect other processes than the one that is executing. Indeed, the System Error Handler often simply stops execution of the application that caused the error rather than present the system error alert box. In this case, the Finder reports that the application has unexpectedly quit, as shown in Figure 2-4.

Figure 2-4 Handling of a nonfatal system error in System 7



An application that attempts to resume execution after a system error is likely to encounter the same problem again and might even encounter more serious problems. In early versions of system software, such an attempt constituted a harmless last-ditch effort by an application to salvage itself. In current versions of system software, such an attempt may cause a **fatal system error**—that is, a system error that crashes the entire system—even if the initial system error was nonfatal.

If your application is designed to work with System 7 only, you should always pass `NIL` to `InitDialogs` and forego a resume procedure. You might alternatively pass a pointer to a simple resume procedure that simply quits the program, as illustrated in Listing 2-1.

Listing 2-1 A simple resume procedure

```
PROCEDURE MyResumeProc;
BEGIN
   ExitToShell;
END;
```

If you wish, you might write a custom resume procedure that you install only on Macintosh computers running versions of system software prior to System 7. Typically, such resume procedures simply jump to the beginning of the application’s main event loop and hope for the best. Because Pascal does not permit a procedure to include a `GOTO` statement that references a label outside its scope, resume procedures typically are written in assembly language.

s **WARNING**

Implementing a resume procedure is not an adequate substitute for quality assurance. Your application should not, for example, allow the user to open so many documents that memory runs out, causing a system error. Calling the System Error Handler’s `SysError` procedure to report a problematic condition to the user might cause a system crash even if no crash would have otherwise occurred and even if your application uses the simple resume procedure defined in Listing 2-1. s

System Error Handler Reference

This section describes the routine and resource that the System Error Handler uses when generating a system error. Although your application may use the routine, ordinarily there is no need to do so. The system error alert table resource is private to the System Error Handler and documented for completeness only.

System Error Handler Routines

The Operating System calls the `SysError` procedure to force display of the system error alert box.

SysError

You can use the `SysError` procedure to simulate a system error. Ordinarily, however, only the Operating System invokes this procedure.

```
PROCEDURE SysError (errorCode: Integer);
```

`errorCode` The system error ID corresponding to the system error condition identified.

DESCRIPTION

The `SysError` procedure generates a system error with the system error ID specified by the `errorCode` parameter. The value of the system error ID determines the exact response of the System Error Handler (for example, whether it can intercept the error) and determines the contents of the system error alert box displayed for the error.

The `SysError` procedure begins by saving all registers and the stack pointer and by storing the system error ID in a global variable (named `DSErrCode`). The Finder uses this global variable when reporting that an application unexpectedly quit.

If there is not a system error alert table in memory, `SysError` loads it in. (The global variable `DSAlertTab` stores a pointer to the current system error alert table. If no system error alert table is in memory, `DSAlertTab` is `NIL`.) If there is no table in memory (indicating that the error likely occurred at the beginning of system startup), the System Error Handler draws the “sad Macintosh” icon and plays appropriate ominous tones through the Macintosh speaker. Different tones correspond to different problems that the `SysError` procedure determines have occurred.

After allocating memory for QuickDraw global variables on the stack and initializing QuickDraw, `SysError` initializes a graphics port in which the alert box is drawn.

The SysError procedure draws the alert box (in the rectangle specified by the global variable DSAlertRect) unless the errorCode parameter contains a negative value. *Note that the system error alert box is not a Dialog Manager modal dialog box.* Negative values are used to force the SysError procedure to display a sequence of consecutive messages in a system startup alert box without redrawing the entire alert box. If the value in the errorCode parameter does not correspond to an entry in the system error alert table, the default alert box definition at the start of the table is used, displaying the message “Sorry, a system error occurred.”

The SysError procedure uses the value in the errorCode parameter to determine the contents of the system error alert box. It looks in the system error alert table resource for an alert definition whose definition ID matches the errorCode parameter. It then draws the text and icon of the alert box according to that alert definition in the system error alert table.

System error alert tables include procedures and button definitions. (See the description of the system error alert table resource in the section “The System Error Alert Table Resource” beginning on page 2-16, for details.) If the procedure definition ID in the table is not 0, SysError invokes the procedure with the specified ID. If the button definition ID in the table is 0, SysError returns control to the procedure that called it. This mechanism allows the disk-switch alert box to return control to the File Manager after the “Please insert the disk:” message has been displayed.

If a resume procedure has been defined, the button definition ID is incremented by 1. This mechanism allows the System Error Handler to use one of two layouts depending on whether a resume procedure has been defined. After drawing the buttons using QuickDraw rather than the Control Manager, SysError performs hit-testing on the buttons, highlighting them appropriately. When a button is pressed, the appropriate procedure is invoked. If there is no procedure code defined for a button, the SysError procedure returns to the routine that called it. The resume procedure is described in the next section.

SPECIAL CONSIDERATIONS

Calling the SysError procedure might cause a system crash even if no condition that would have caused a system crash existed prior to the invocation of SysError.

SysError works correctly only if the following conditions are met:

- The trap dispatcher is operative. (See the chapter “Trap Manager” in this book for information about the trap dispatcher.)
- The Font Manager procedure InitFonts has been called. Ordinarily, it is called when the system starts up.
- Register A7 points to a reasonable place in memory (for example, not to video RAM).
- A few important system data structures do not appear to be too badly damaged.

SEE ALSO

A list of system error IDs is provided in Table 2-1 on page 2-7.

Application-Defined Routines

The System Error Handler calls your application's resume procedure when the user clicks the Continue button (or the Resume button on earlier versions of system software) in the system error alert box.

MyResumeProc

When you call the Dialog Manager procedure `InitDialogs`, your application can pass a pointer to a resume procedure. If you don't want to install a resume procedure, pass `NIL`. A resume procedure has the following syntax:

```
PROCEDURE MyResumeProc;
```

DESCRIPTION

If your application is the current process, your application's resume procedure is called when the user responds to a system error alert box by clicking the Continue button. No parameters are passed to a resume procedure.

In System 7, the System Error Handler intercepts many system errors and stops execution of the process, causing an error rather than calling the application's resume procedure.

SPECIAL CONSIDERATIONS

In general, you should not write code to allow your application to continue to execute normally after a system error has occurred. An application that attempts to resume execution after a system error is likely to encounter the same problem again and might even encounter more serious problems. In early versions of system software, such an attempt constituted a harmless last-ditch effort by an application to salvage itself. In current versions of system software, such an attempt may cause a fatal system error—that is, a system error that crashes the entire system—even if the initial system error was nonfatal.

SEE ALSO

For more information about resume procedures, see the section "Resume Procedures" on page 2-11.

Resources

This section describes the system error alert table ('DSAT') resource. The System Error Handler uses resources of this type to determine what to display in the system startup alert box and the system error alert box. You should never need to access or change these resources; the information is provided for completeness only.

The System Error Alert Table Resource

The System Error Handler stores system error alert tables in resources with resource type `'DSAT'`. During system startup, the system error alert table resource with resource ID 0 is loaded. This resource describes the "Welcome to Macintosh" alert box. Immediately thereafter, that table is disposed of and replaced with the system error alert table resource with resource ID 2.

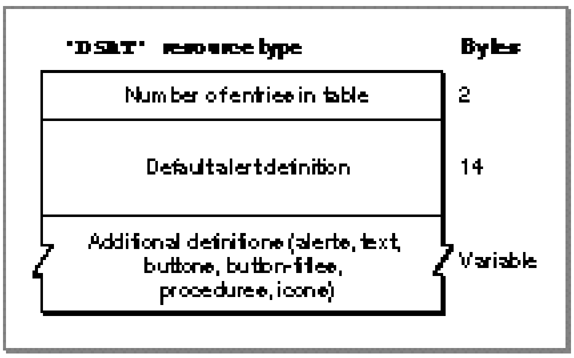
Note

In early versions of system software the system error alert table was called the "user alert table" and its resource type was of type `'INIT'`. u

A system error alert table consists of a group of alert definitions, text definitions, icon definitions, procedure definitions, button definitions, and button-title definitions. These definitions provide information about the alert box as a whole: the text, icon, buttons, and titles for those buttons to be displayed in the alert box, and the procedures to be executed. The first word (2 bytes) of any definition contains a definition ID, which must be unique across all definitions. Some definitions reference other definitions. For example, a button definition includes a word to reference a button-title definition and a word to reference a procedure definition. This section describes the format of the system error alert table as a whole and of the various types of definitions.

A system error alert table's first word indicates the number of entries in the table. Following these 2 bytes is a 14-byte alert definition that defines an alert box to be used for all system errors that do not have their own alert box definitions. This alert box definition is followed by additional definitions, which need not be in any particular order. For example, a system alert table could contain all alert box definitions before any other definitions, but this might not be the case. Figure 2-5 illustrates the overall structure of a system error alert table.

Figure 2-5 The structure of a system error alert table



All definitions in a system error alert table contain a 4-byte definition header. The first word of the header is the unique definition ID for that definition, which corresponds to the appropriate system error for alert box definitions, and the second word is a number indicating the length in bytes of the remainder of the definition.

Figure 2-6 shows the format of an alert definition.

Figure 2-6 The structure of an alert definition

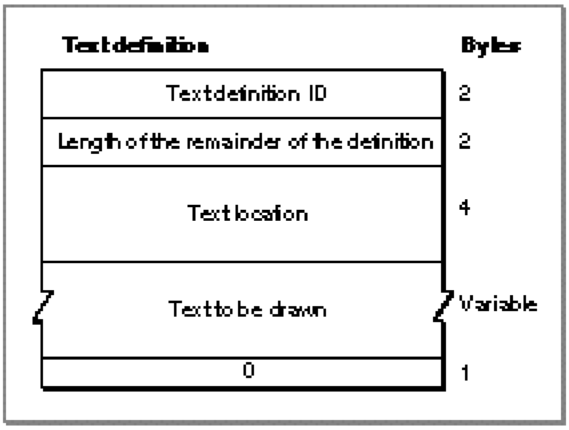
Alert definition	Bytes
System error ID	2
Length of the remainder of the definition	2
Primary text definition ID	2
Secondary text definition ID	2
Icon definition ID	2
Procedure definition ID	2
Button definition ID	2

Following the definition header, the alert definition consists of five word-length fields containing the definition IDs for a primary text definition, a secondary text definition, an icon definition, a procedure definition, and a button definition. For each alert definition, two button definitions must be defined with consecutive numbers. The lower of these numbers is specified in the button definition ID field. When an application specifies a resume procedure, the `SysError` procedure uses the button definition with the higher ID.

A definition ID of 0 is used for any field to which no definition corresponds. For example, if a system error alert box contains only one text string, the field for the secondary text definition ID contains 0. A button definition ID of 0 indicates that `SysError` should return to the procedure that called it; this is used for disk-insertion alerts. If the procedure definition ID is 0, `SysError` does not invoke an alert procedure (which should not be confused with a resume procedure).

A text definition specifies the text that is to be drawn in the system error alert box. Because an alert box can have up to two lines of text, the alert definition allows for two text definitions. The primary text definition specifies the first line of text in the system error alert box and the secondary text definition specifies the second line of text. Figure 2-7 illustrates the format of a text definition.

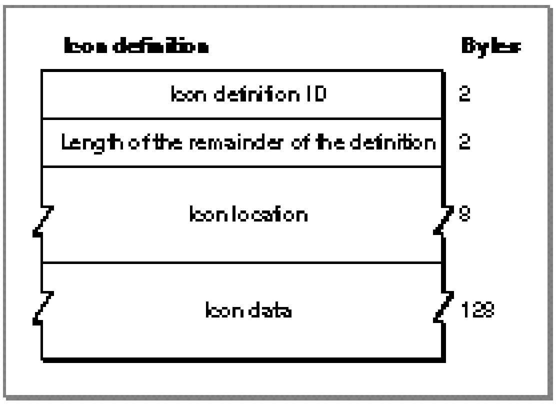
Figure 2-7 The structure of a text definition



Following the definition header, a text definition includes a 4-byte field indicating the point, specified in global coordinates, at which the text is to be drawn. Following this field is a variable-length field consisting of the text to be drawn. The System Error Handler responds to the slash (/) character by advancing to the beginning of the next line. This mechanism allows a single text definition to consist of a multiline message. The last byte of the definition must contain `0` to indicate the end of the text.

An icon definition specifies what icon the System Error Handler draws in the system error alert box, where to draw it, whether the icon is black-and-white or color, the bit depth of the icon, and other data as necessary. Figure 2-8 shows the format of an icon definition.

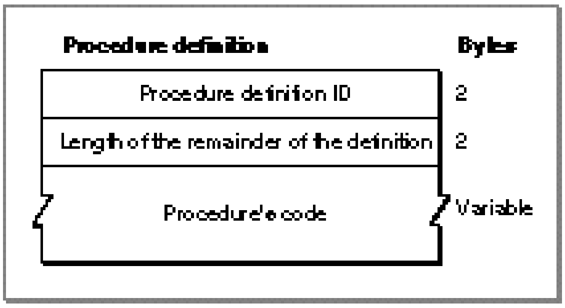
Figure 2-8 The structure of an icon definition



Following the definition header, the icon definition contains an 8-byte field indicating the rectangle, specified in global coordinates, in which to draw the icon. The following 128 bytes consist of icon data.

An alert definition uses a procedure definition to specify a procedure to be executed whenever the `SysError` procedure draws a system error alert box. Button definitions (described next) use procedure definitions to specify an action to be taken when the user presses a particular button. Figure 2-9 illustrates the format of a procedure definition.

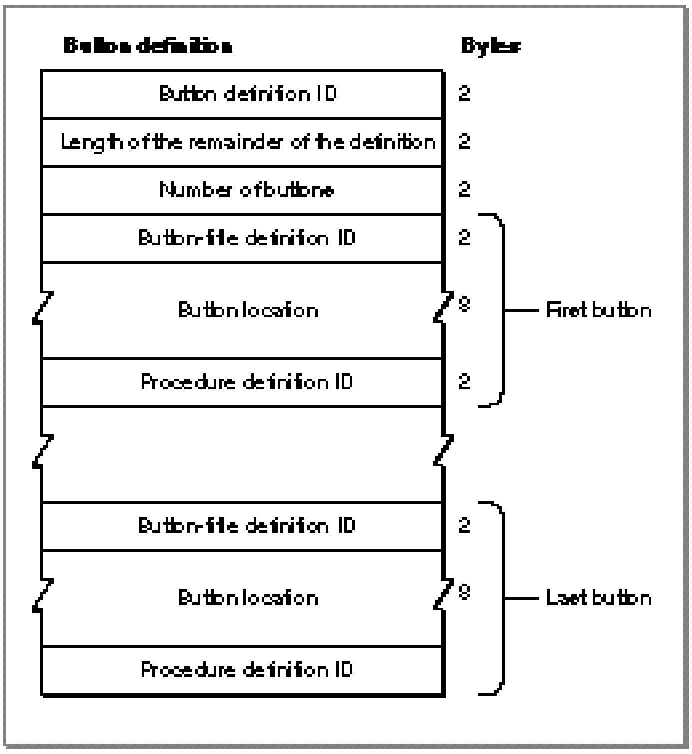
Figure 2-9 The structure of a procedure definition



After the definition header, a procedure definition consists only of a variable-length field that contains the procedure's code. The procedure takes no parameters.

A button definition specifies the buttons that the System Error Handler should draw in the system error alert box. A button definition may reference 0, 1, 2, or more buttons. Figure 2-10 shows the format of a button definition.

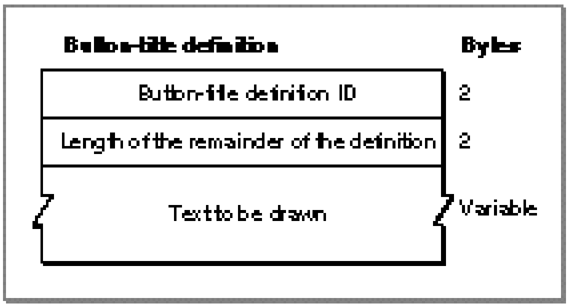
Figure 2-10 The structure of a button definition



Following the definition header is a word indicating the number of buttons in the button definition. Following this is 12 bytes for each defined button. Each of these 12-byte groups consists of a word containing the button-title definition ID for the text within the button, 8 bytes containing a rectangle, in global coordinates, that specifies the location of the button, and a word containing the procedure definition ID for the procedure to be executed when the button is pressed.

A button-title definition specifies the text to be drawn within a button. Figure 2-11 shows a button-title definition. Following the definition header of the button-title definition are the actual characters in the string.

Figure 2-11 The structure of a button-title definition



Summary of the System Error Handler

Pascal Summary

System Error Handler Routines

```
PROCEDURE SysError          (errorCode: Integer);
```

Application-Defined Routines

```
PROCEDURE MyResumeProc;
```

C Summary

System Error Handler Routines

```
pascal void SysError        (short errorCode);
```

Application-Defined Routines

```
pascal void MyResumeProc;
```

Assembly-Language Summary

Global Variables

`DSErrCode`	The system error ID of the last system error.
`DSAlertTab`	A pointer to the system error alert table in memory, or `NIL` if none has been loaded.
`DSAlertRect`	The rectangle, in global coordinates, in which to draw the system error alert box.

C H A P T E R 3

Mathematical and Logical Utilities

Contents

This chapter describes a number of utility routines that you can use to perform mathematical and logical operations supported directly by the Macintosh Operating System. In particular, this chapter discusses how you can

- perform low-level logical manipulation of bits and bytes when using a compiler that does not directly support such manipulations
- save disk space by using simple compression and decompression routines
- obtain a pseudorandom number
- perform mathematical operations with two fixed-point data types supported directly by the Operating System
- convert numeric variables of different types

You need to read this chapter only if you need access to any of these features. With the exception of the mathematical operations and conversions, the routines this chapter describes are intended for programmers who occasionally need to access some of these features and do not require that the algorithms used to implement them be sophisticated. For example, if you are developing an advanced mathematical application, the pseudorandom number generator built into the Operating System might be too simplistic to fit your needs. Similarly, if you wish to access individual bits of memory in a time-critical loop, the Operating System routines that perform these operations are probably too slow to be practical.

You do not need any prior knowledge of the Operating System to read this chapter, which begins by describing the building blocks of memory in any operating system: bits, bytes, words, and long words. After subsequent discussions of the built-in compression and decompression routines provided by the Operating System, this chapter illustrates how you can use the Operating System's Mathematical and Logical Utilities. The chapter concludes with a reference to all mathematical and logical routines supported by the Operating System. If you are an experienced programmer, you might be able to skip directly to that section to determine which routine you need.

This chapter does not describe the numeric data types supported by the Standard Apple Numerics Environment (SANE) that the Operating System does not support directly. For more information on such data types, consult the *Apple Numerics Manual* and *Inside Macintosh: PowerPC Numerics*.

About the Mathematical and Logical Utilities

This section begins by introducing the building blocks of memory and then discusses some low-level routines the Mathematical and Logical Utilities provide, such as routines that compress data and generate pseudorandom numbers. Finally, the section concludes by introducing two fixed-point data types the Operating System supports.

Bits, Bytes, Words, and Long Words

This section describes the fundamental memory units used in all computer systems and discusses some of the operations that you can perform on them using the Mathematical and Logical Utilities. If you already know what bits, bytes, words, and long words are, you can skip this section.

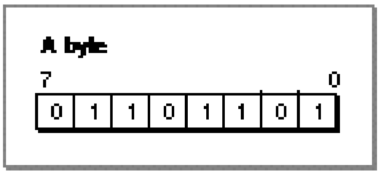
A **bit** is the atomic memory unit. Each bit can be set to one of two values. Often these values are called 0 and 1. A bit is said to be cleared when its value is 0 and set when its value is 1.

Eight bits form a single **byte.** The first bit in a byte is bit number 7, and the last bit is bit number 0. Bit number 7 is called the **most significant bit** or the **high-order bit,** and bit number 0 is the **least significant bit** or the **low-order bit.** A byte can thus store 2^8, or 256, different possible values. In Pascal, a byte is thus defined like this:

```
TYPE
   Byte = 0..255;
```

Figure 3-1 illustrates a byte set to the base-10 value 109.

Figure 3-1 A byte set to 109 ($6D)



The base-10 value 109 is equivalent to the binary value 01101101. This sequence of binary digits exactly corresponds to the status of each bit in the byte illustrated in Figure 3-1. A byte value is typically represented by two hexadecimal digits. The value in Figure 3-1, for example, is equivalent to $6D.

Sometimes it is useful to quickly convert between hexadecimal and binary number formats during debugging when examining the values of individual bits in a byte. Table 3-1 provides an easy way to do this on a digit-by-digit basis.

Table 3-1 Converting hexadecimal digits to binary values

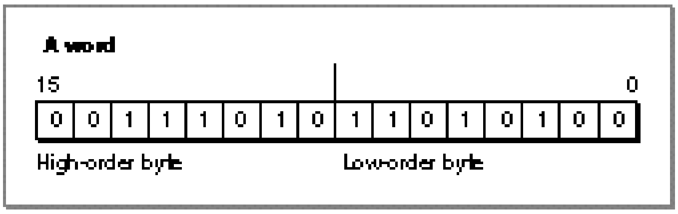
Hexadecimal	Binary
$0	0000
$1	0001
$2	0010
$3	0011
$4	0100
$5	0101
$6	0110
$7	0111
$8	1000
$9	1001
$A	1010
$B	1011
$C	1100
$D	1101
$E	1110
$F	1111

For example, the hexadecimal value $A8 is equivalent to the binary value 10101000 because the hexadecimal digit $A is equivalent to 1010 and the digit $8 is equivalent to 1000. You can use Table 3-1 to convert numbers in both directions.

While you can always think of a byte as a particular value from $00 to $FF, sometimes that value is irrelevant. For example, an application might use a byte simply as a way to store eight flag bits; in this case, the application cares about only individual bits within the byte and not the value of the byte as a whole. Also, bytes are often used to store signed values, in which case a byte can be considered equivalent to values from –$80 to +$7F. If you use a low-level debugger like MacsBug to examine individual bytes in memory, you should also be aware that different compilers might use bytes in different ways.

Two bytes form a **word.** A word is thus a 16-bit quantity and can be used to store 2^{16} (or 65,536) possible values. A **word boundary** is the memory location that divides two words. The first byte in a word is known as the high-order byte, and the second byte is known as the low-order byte. A pointer to a word points to the high-order byte. Figure 3-2 illustrates a word.

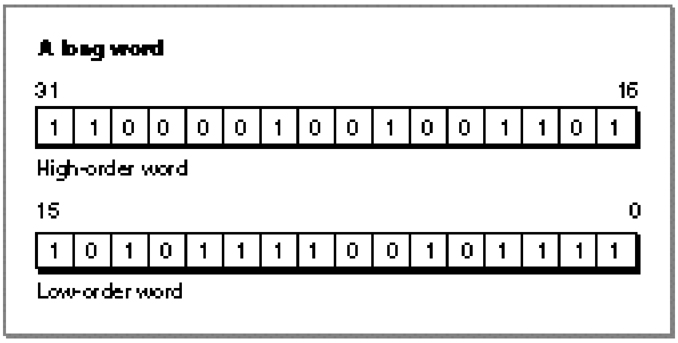
Figure 3-2 A word set to $3AD4



In Figure 3-2, the high-order byte is set to $3A. The low-order byte is set to $D4. The word thus has the value $3AD4.

Two words form a **long word.** A long word is thus a 32-bit quantity and can be used to store 2^{32} (or 4,294,967,296) values. A **long-word boundary** is the memory location that divides two long words. A long word consists of a high-order word and a low-order word, as illustrated in Figure 3-3.

Figure 3-3 A long word set to $C24DAF2F



In Figure 3-3, the high-order word is set to $C24D. The low-order word is set to $AF2F. The long word thus has the value $C24DAF2F.

Variables of type `Integer` are signed words, and variables of type `LongInt` are signed long words. On current versions of the Operating System, a memory address is stored using all 32 bits of a long word.

Typically, Macintosh compilers align all values on word boundaries (and in some cases on long-word boundaries). This means that when you declare a variable of type `Byte` in Pascal, the compiler is in fact likely to allocate 2 bytes of memory to store the byte; the extra byte is called a **pad byte.** In this case, when you attempt to test bits in a byte you have allocated, the compiler might test the corresponding bit in the wrong byte.

In Pascal, there are two easy ways to avoid this problem. One is to aggregate variables of type `Boolean` and of type `Byte` in a packed record. In this case, as long as the packed record's size is a number of bytes that is a multiple of 4, no pad bytes are added. The second technique is, for variables in which you wish to test individual bits, to allocate 2 or 4 bytes for the variable (using a variable of type `Integer` or `LongInt`, respectively).

Bit Manipulation and Logical Operations

The Mathematical and Logical Utilities provide a number of routines that provide bit-level and byte-level control over memory, as described in “Performing Low-Level Manipulation of Memory” beginning on page 3-14. Given a pointer and offset, these routines can manipulate any specific bit in a stream of bits.

The `BitTst`, `BitSet`, and `BitClr` routines allow you to test and clear individual bits within a byte. These functions are introduced in “Testing and Manipulating Bits” on page 3-14.

Note
The `BitTst`, `BitSet`, and `BitClr` routines use a bit-numbering scheme that is opposite that of the MC680x0 microprocessor. This reversed bit-numbering scheme is described in the next section. u

The `BitAnd`, `BitOr`, `BitXor`, and `BitNot` functions allow you to perform logical operations on long words, and the `BitShift` function allows you to shift the bits in a long word to the right or to the left. These functions are introduced in “Performing Logical Operations on Long Words” on page 3-16.

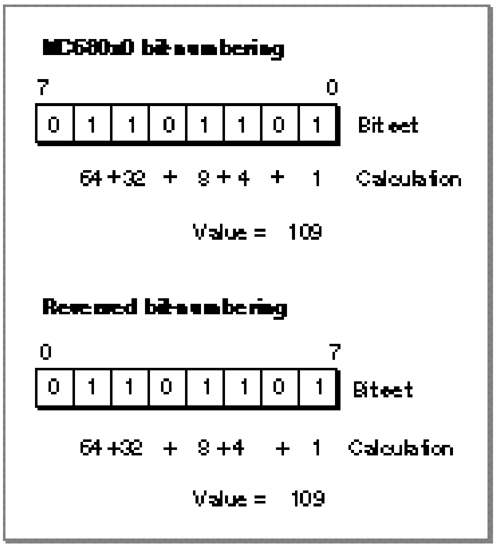
You might also need to extract one of a long word’s words. The `HiWord` and `LoWord` functions allow you to do this and are described in “Extracting a Word From a Long Word” on page 3-18. Finally, you might need to set a group of bytes’ values directly. The `StuffHex` procedure enables you to hardcode hexadecimal values to bytes anywhere in memory and is described in “Hardcoding Byte Values” on page 3-19.

Reversed Bit-Numbering

Three of the routines described in this chapter (the `BitTst`, `BitSet`, and `BitClr` routines) use a bit-numbering scheme that is opposite from that of the bit-numbering scheme used by the MC680x0 microprocessor.

The `BitTst`, `BitSet`, and `BitClr` routines count the bit numbers from left to right. That is, the most significant bit has the bit number 0. The MC680x0 bit number notation counts the bit numbers from right to left. (That is, the most significant bit has the biggest bit number.) Figure 3-1 illustrates these bit-numbering schemes.

Figure 3-4 Bit-numbering schemes



When using routines other than the `BitTst`, `BitSet`, and `BitClr` routines or if you are an assembly-language programmer, you should use the MC680x0 bit-numbering scheme.

To convert from MC680x0 bit notation to the scheme described in this section, subtract the MC680x0 bit number from the highest bit number. For example, to clear bit number 3 in a byte, you must clear bit number 4 (7–3 = 4).

Data Compression

The Mathematical and Logical Utilities include two procedures, `PackBits` and `UnpackBits`, that allow you to provide rudimentary data compression and decompression, respectively. The procedures are not powerful enough to provide effective compression for applications that primarily concern themselves with data compression. Also, if you are compressing sound, image, or video data, the Sound Manager (described in *Inside Macintosh: Sound*) and the Image Compression Manager (described in *Inside Macintosh: QuickTime*) provide far more effective compression algorithms.

You can use the `PackBits` and `UnpackBits` procedures to conserve memory both in RAM and on disk. However, because decompressing data is time consuming, typically you compress data using the `PackBits` procedure before saving a file or resource to disk and decompress data using the `UnpackBits` procedure after reading the data back from disk. Because the time required for compression and decompression using `PackBits` and `UnpackBits` is usually trivial compared to the time it takes to access a typical hard disk, the routines provide a simple, low-overhead way for an application to minimize the size of its data files.

The `PackBits` procedure is effective when an uncompressed buffer of data is likely to have many consecutive bytes containing the same value. For example, some applications use data structures that include fields that the application reserves for future use. These fields are typically all set to 0. The `PackBits` procedure senses that there is a long string of consecutive bytes containing the same value and compresses the string of bytes by using 1 byte to indicate that the subsequent compressed byte represents a number of consecutive uncompressed bytes.

`PackBits` was originally intended as an easy way to compress black-and-white image data, such as MacPaint documents. However, because each pixel of a color picture is typically represented by multiple bytes of data, `PackBits` is unlikely to provide effective compression for such pictures.

If there is no reason to think that your data format might contain long strings of consecutive bytes, then the `PackBits` procedure is probably not useful and might even increase the size of your files. The `PackBits` procedure packs data 127 bytes at a time. If within the 127 bytes there is no series of 3 consecutive bytes containing the same value, then there are no gains to be made from compression. In this case, the `PackBits` procedure must use an initial byte to specify that the 127 subsequent bytes contain uncompressed data. You can compute the worst-case performance of `PackBits` (that is, the maximum number of output bytes) by using the following formula:

```
maxDstBytes := srcBytes + (srcBytes+126) DIV 127;
```

where `maxDstBytes` stands for the maximum number of destination bytes and `srcBytes` stands for the number of bytes in the uncompressed source data.

You can, if desired, pack a buffer of data, and then pack the packed buffer again. However, packing data twice not only is slower than packing data once, but also is likely to result in a larger output buffer than just packing data once. If your application does pack data twice, it should unpack the data twice.

Note

In current versions of system software, you can request that `PackBits` pack up to 32,767 bytes. The `PackBits` procedure then processes the input buffer in 127-byte chunks. In versions of system software prior to version 6.0.2, however, you should pass to `PackBits` only buffers up to 127 bytes in length. u

Pseudorandom Number Generation

Because digital computers continuously execute instructions, it is impossible for a computer to select a truly random number. To force the computer to output a number, the programmer must create an algorithm, but because algorithms always execute in the same way, the numbers an algorithm produces cannot be truly random. Random numbers are often necessary in software applications, however. For example, an entertainment software application might need to ensure that the user is not faced with the exact same game every time. Or a spreadsheet application might offer a randomization function for business users attempting to simulate various possible scenarios.

To get around the impossibility of producing truly random numbers, computer scientists rely on pseudorandom number generation algorithms. These are complex numeric algorithms used to produce a series of numbers. All such series eventually repeat, but typically not until the pseudorandom number generation algorithm has been executed millions or even billions of times. Because the series is generated by an algorithm, it is possible to discern a pattern; given the first few numbers of a series, a clever user might be able to guess the next number. Typically, however, these algorithms are complicated enough to make the numbers appear random, at least to the casual observer.

Of course, because pseudorandom number generation algorithms are algorithms, they produce the same series of numbers every time. However, you can seed the pseudorandom number generator to force it to start somewhere in the middle of the series. By seeding the generator to a constantly changing variable when your application starts up, your application can produce different results each time. The value typically used to seed the pseudo-random number generator is the current date and time. Of course, time isn't random—it moves forward at a constant linear rate—but in the absence of a stopped system clock, the user will never launch your application at the same time twice, so you can be confident that your application will produce different results each time it is executed.

The Macintosh Operating System's pseudorandom number generation algorithm is accessible through the `Random` function. The `Random` function returns a pseudorandom integer from –32767 to 32767. The value that the `Random` function produces depends on the `randSeed` global variable. The `Random` function changes `randSeed` while generating a pseudorandom number, thus enabling a subsequent call to `Random` to produce the next number in the series. You only need to seed the global variable once, at the start of your program.

The pseudorandom number generation algorithm is designed so that as the number of times `Random` is executed approaches infinity, the percentage difference in the number of times any two integers in the range –32767 to 32767 are produced approaches 0. Thus, the pseudorandom number generator is said to produce pseudo-random numbers that are uniformly distributed in the range –32767 to 32767.

This chapter does not describe the algorithm that `Random` uses to generate pseudorandom numbers. While the algorithm is sufficiently complex for most applications, applications that perform mathematical or statistical analysis might require a better pseudo-random number generator. Consult the computer science literature for information on sophisticated pseudorandom number generation algorithms.

Fixed-Point Data Types

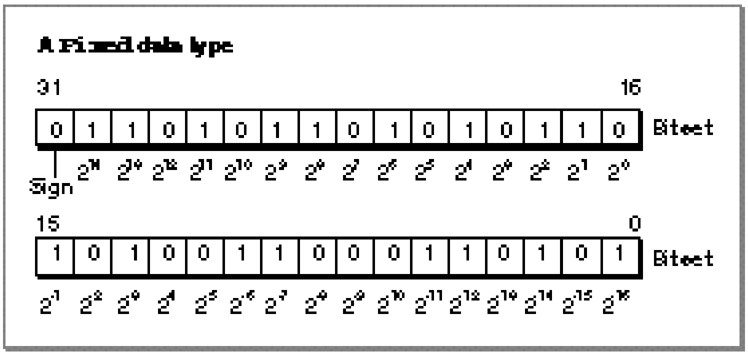
The Operating System supports two fixed-point data types, that is, numeric types that consist of integral and fractional components. Depending on the type of information you are representing with a fixed-point data type, these might be better suited for your needs than the types `Integer`, `LongInt`, and the many floating-point types supported by the Standard Apple Numerics Environment.

A variable of type `Fixed` is defined like this:

```
TYPE
   Fixed = LongInt;
```

A variable of type `Fixed` is a 32-bit signed quantity containing an integer part in the high-order word and a fractional part in the low-order word. Figure 3-5 illustrates the format for `Fixed`.

Figure 3-5 The `Fixed` data type



The high-order word consists of the integral component of the fixed-point number, and the low-order word consists of the fractional component of the fixed-point number. Each bit, other than the most significant bit, represents a power of 2, as indicated in Figure 3-5.

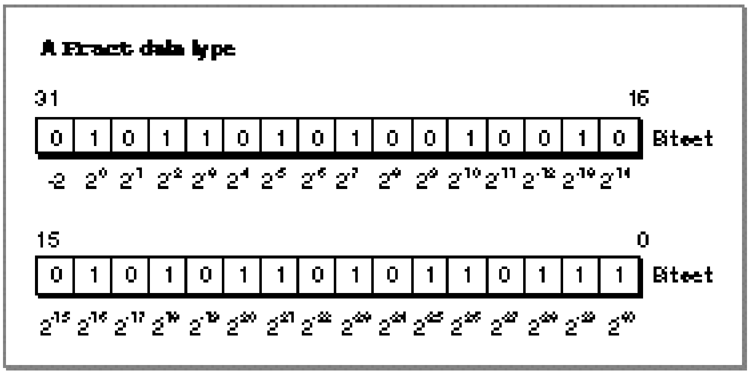
Negative numbers of type `Fixed` are the two's complement; that is, the negative numbers are formed by treating the fixed-point number as a long integer, inverting each bit, and adding 1 to the least significant bit.

The `Fract` data type is useful for allowing accurate representation of small numbers, that is, numbers between –2 and 2. It is defined just like `Fixed`:

```
TYPE
   Fract = LongInt;
```

Figure 3-6 illustrates the format for `Fract`.

Figure 3-6 The `Fract` data type



Like a `Fixed` number, a `Fract` number is a 32-bit quantity, but its implicit binary point is to the right of bit 30 of the number; that is, a `Fract` number has 2 integer bits and 30 fraction bits. As with the type `Fixed`, a number is negated by taking its two's complement. Thus, `Fract` values range between –2 and 2 – (2^{-30}), inclusive.

All routines that operate on fixed-point numbers handle boundary cases uniformly. Results are rounded by adding half a unit in magnitude in the last place of the stored precision and then chopping toward zero. Overflows are set to the maximum representable value with the correct sign ($80000000 for negative results and $7FFFFFFF for positive results). Division by zero results in $8000000 if the numerator is negative and $7FFFFFFF otherwise; thus, the special case 0/0 yields $7FFFFFFF.

Angle-Slope Conversion

The Mathematical and Logical Utilities provide two functions for applications that need to draw lines at particular angles. For example, a mathematical plotting application might need to draw a 30-degree line. The `SlopeFromAngle` and `AngleFromSlope` functions provide simple conversion between slope and angle values. Slopes and angles are defined in such a way as to be convenient to a computer programmer rather than correspond to the conventional mathematical interpretation.

Note

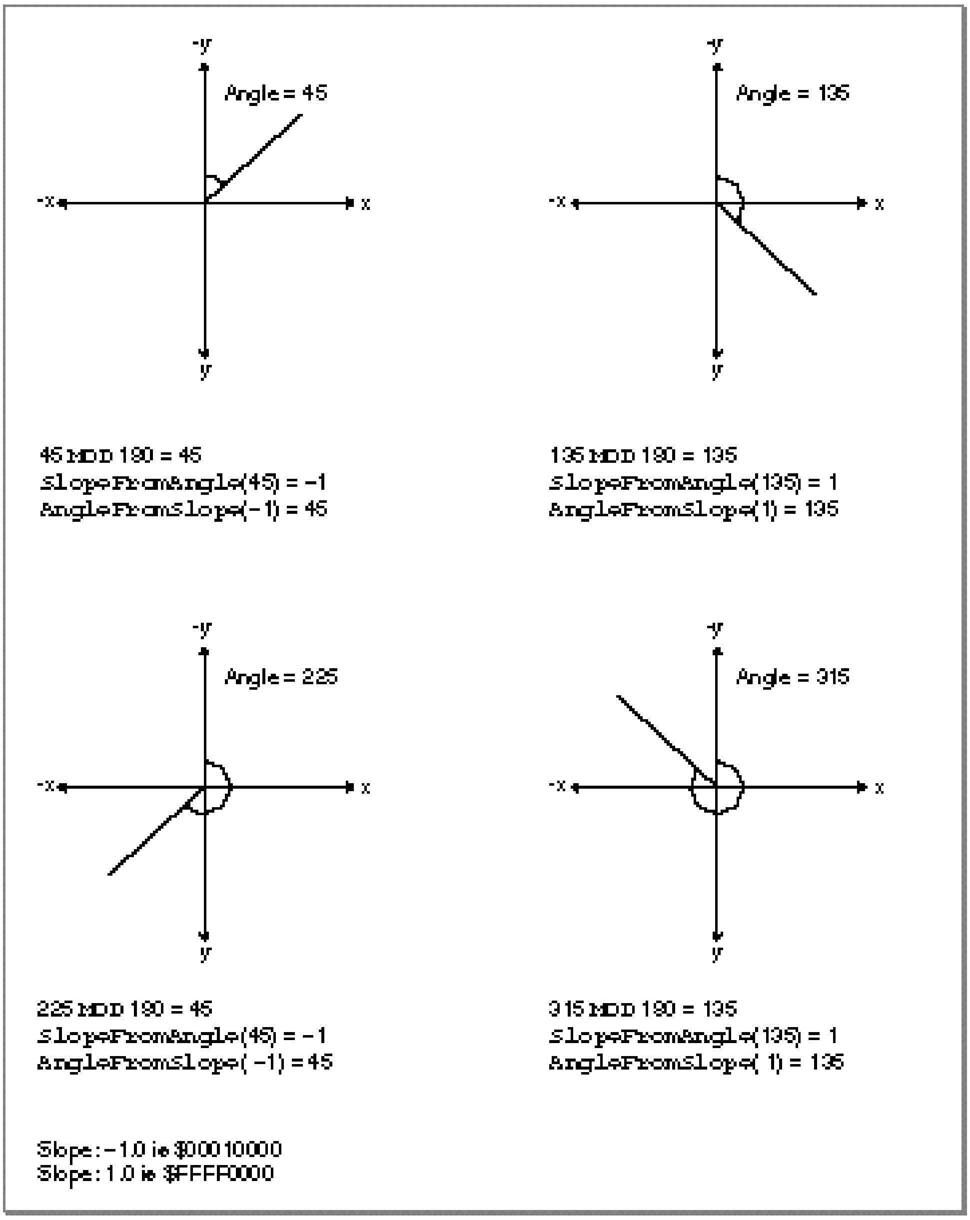
You should not rely on the `SlopeFromAngle` and `AngleFromSlope` functions to produce values that will allow you to draw lines at a precise angle on the screen. The functions do not take into account the size of pixels on a screen. If pixels on a screen are not perfect squares, a 30-degree angle might appear to be a different angle to the user. u

Since QuickDraw and other computer imaging schemes typically invert the y-axis (making positive down and negative up), the angle-slope conversion routines use this convention as well. Angles are measured clockwise relative to the negative y-axis (that is, relative to 12 o'clock), and are taken `MOD` 180, so that a 270-degree angle is considered to be equivalent to a 90-degree angle.

Slopes are defined as $\Delta x / \Delta y$, the horizontal change divided by the vertical change for any two points on a line with the slope. Note that mathematicians typically measure slopes $\Delta y / \Delta x$. The convention of angle-slope conversion is convenient for applications that plot a number of lines in a graph one horizontal line at a time.

Figure 3-7 shows some equivalencies between angle and slope values for the angle-slope conversion routines.

Figure 3-7 Some slope and line equivalencies using the conventions of the angle-slope conversion routines



The `AngleFromSlope` function is useful primarily only when speed is more important than accuracy because the function might return an angle off by as much as 1 degree from the actual angle. The function returns values between 1 and 180 (inclusive), and thus never returns an angle value between 0 and 1 degrees. If your application is likely to need precise differentiation in angles, you should probably develop alternative routines to handle angle-slope conversions.

`SlopeFromAngle(0)` is 0, and `AngleFromSlope(0)` is 180. For all `x` except for 0, however, `AngleFromSlope(SlopeFromAngle(x)) = x` is true. But the reverse, `SlopeFromAngle(AngleFromSlope(x)) = x` is not necessarily true.

Using the Mathematical and Logical Utilities

This section describes how you can take advantage of the Mathematical and Logical Utilities supported by the Operating System, it describes how you can

- test and set individual bits, perform logical operations on long words, divide a long word into its high word and low word, and set memory values directly.
- use the `PackBits` and `UnpackBits` procedures to compress and decompress data.
- seed the pseudo-random number generator and obtain random integers or long integers within a given range.
- perform simple calculations involving fixed-point numbers and convert fixed-point numbers to other numeric types.

Performing Low-Level Manipulation of Memory

The Mathematical and Logical Utilities provide several routines to perform bit-level and byte-level manipulation of memory. These routines are provided primarily for Pascal programmers. C and assembly-language programmers can use these routines also; however, in general it is easier and more efficient to achieve the same effects as these routines by using built-in C or assembly constructs.

Testing and Manipulating Bits

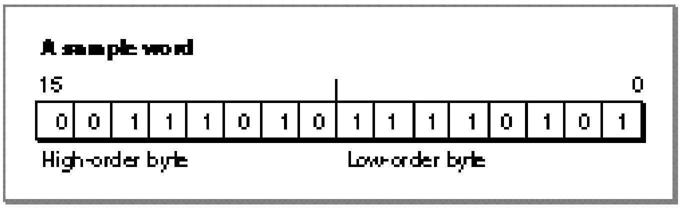
The `BitTst` function lets you test whether a given bit is set. The function requires that you specify a bit through an offset from a pointer. Listing 3-1 is an example of an application-defined function that tests a specified bit.

Listing 3-1 Testing bits

```
FUNCTION MyTestBit (bytePtr: Ptr; bitNum: LongInt): Boolean;
BEGIN
   MyTestBit := BitTst(bytePtr, bitNum);
END;
```

The `bytePtr` parameter specifies a pointer to a byte in memory. The `bitNum` parameter specifies the number of the bit to be tested as an offset from `bytePtr`. For example, you can use the application-defined function `MyTestBit` to test specific bits of the word specified in Figure 3-8.

Figure 3-8 A sample word (in MC680x0 notation)



Using the word in Figure 3-8, the call `BitTst(myPtr, 0)` returns FALSE because bit number 0 in the first byte is not set. But the call `BitTst(myPtr, 11)` returns TRUE because bit number 3 in the second byte is set.

When using the `BitTst` function, be sure to specify bits as positive offsets from the high-order bit rather than using the normal MC680x0 notation (see "Reversed Bit-Numbering" on page 3-7). Listing 3-2 illustrates a use of the `BitTst` function in conjunction with a bit traditionally identified with MC680x0 notation.

Listing 3-2 Determining whether a handle is purgeable using the `BitTst` function

```
FUNCTION MyHandleIsPurgeable (myHandle: Handle): Boolean;
CONST
   kMyBitNum68000 = 6;
VAR
   propertiesByte: SignedByte;
BEGIN
   propertiesByte := HGetState(myHandle);
   MyHandleIsPurgeable := BitTst(@propertiesByte,
                                 7 - kMyBitNum68000);
END;
```

The `MyHandleIsPurgeable` function defined in Listing 3-2 determines whether a handle references a relocatable block by examining the properties byte for that handle. The purgeable bit is, in MC680x0 notation, bit number 6 of the properties byte; because `BitTst` uses reverse numbering, so bit number 7 – 6 = 1 is tested.

The `BitSet` and `BitClr` procedures require that you specify bits using the same scheme as with the `BitTst` procedure (see "Reversed Bit-Numbering" on page 3-7). The `BitSet` procedure sets a bit (that is, sets its value to 1), while `BitClr` clears a bit (that is, sets its value to 0). For example, if you issue the following two calls to the `BitSet` procedure

```
BitSet(bytePtr, 5);
BitClr(bytePtr, 7);
```

bit 5 (using the reversed bit-numbering scheme) of the byte in memory pointed to by the `bytePtr` parameter is set to 1, and bit 7 (using reversed bit-numbering) of the same byte is cleared.

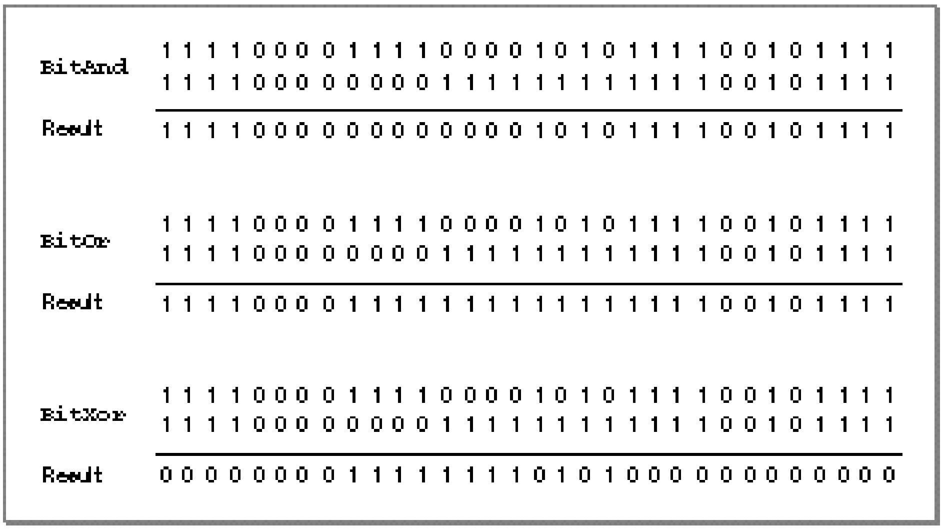
Note

In C, you can test bits by using the & operator. You can set and clear bits by using the |= and &= operators, respectively. In all three cases, one operand should be the byte (or word or long word you wish to manipulate), and the other should be a value in which only the relevant bit is set or cleared. Many Pascal compilers also support built-in operations that accomplish these tasks efficiently. Note that C uses the MC680x0 bit-numbering scheme (normal bit-numbering). u

Performing Logical Operations on Long Words

The Macintosh Operating System provides routines that allow you to perform basic bitwise logical operations, including the AND, OR, and XOR operations on long words. Each of the functions takes two long integers as parameters and returns another long integer. You can use these functions on other 32-bit data types, as long as you cast values to `LongInt` as required by your compiler. The functions that perform the AND, OR, and XOR operations are `BitAnd`, `BitOr`, and `BitXor` respectively. Figure 3-9 illustrates these functions.

Figure 3-9 The `BitAnd`, `BitOr`, and `BitXor` functions



As shown in Figure 3-9, the `BitAnd` function returns a long word in which each bit is set if and only if the corresponding bit is set in both long words passed in. The `BitOr` function returns a long word in which each bit is set if and only if the corresponding bit is set in either long word passed in. The `BitXor` function returns a long word in which each bit is set if and only if one but not both of the corresponding bits in the long words passed in is set.

Note

In C, you can achieve the same effects as the `BitAnd`, `BitOr`, and `BitXor` functions by using the &, |, and ^ operators, respectively, in conjunction with the = assignment operator. Many Pascal compilers also support built-in operations that accomplish these tasks more efficiently. u

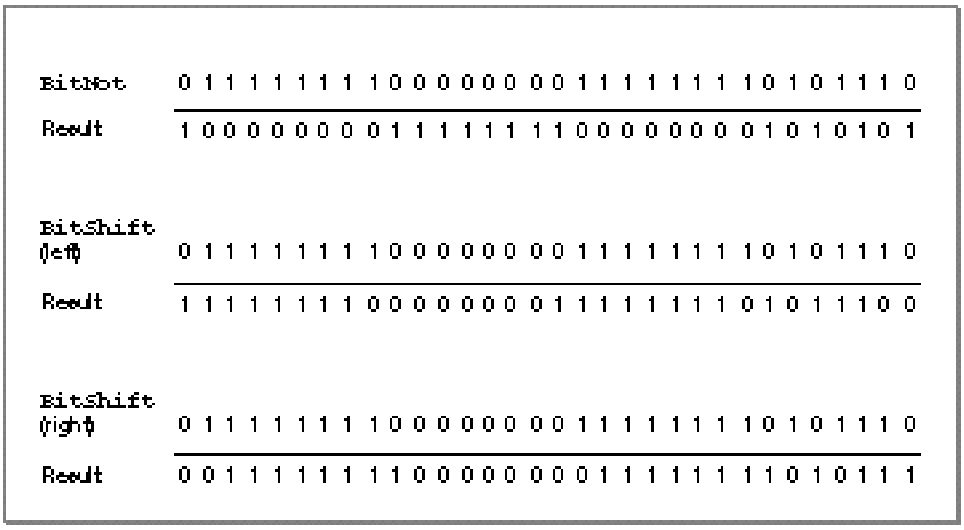
A common use of the `BitAnd` function is to mask out certain bytes within a long word (that is, clear all bits in those bytes). For example, to mask out the second byte of a long word stored in a variable `value`, you could write the following code:

```
value := BitAnd(value, $FF00FFFF);
```

The Macintosh Operating System also offers two bit-manipulation routines that simulate unary operators, the `BitNot` and the `BitShift` functions, which perform the `NOT` operation and bit-shifting, respectively. You specify the long integer on which to perform the operation as a parameter to the `BitNot` and `BitShift` functions. In addition, you specify how to shift the bits as a parameter to the `BitShift` function.

Figure 3-10 illustrates `BitNot` and `BitShift`.

Figure 3-10 The `BitNot` and `BitShift` functions



As shown in Figure 3-10, the `BitNot` function returns a long word in which each bit is set if and only if the corresponding bit in the long word passed in is not set. The `BitShift` function shifts bits—to the left if the `count` parameter is greater than 0 and to the right if the `count` parameter is less than 0. (Shifting to the left means shifting towards the high-order bit.) When shifting `count` bits to the left, the `count` low-order bits are set to 0; when shifting `count` bits to the right, the `count` high-order bits are set to 0.

Note

In C, you can achieve the same effect as the `BitNot` function more efficiently by using the ^ operator on the value whose bits are to be inverted and the value $FFFFFFFF. You can achieve the same effect as the `BitShift` function more efficiently by using the >> operator for shifting to the right and the << operator for shifting to the left. Many Pascal compilers support built-in operations that accomplish these tasks efficiently. u

Extracting a Word From a Long Word

Often a long word stored as a variable of type `LongInt` is used to hold two different pieces of information in its two different words. For example, when a disk-inserted event occurs, the `message` field of the event record contains the drive number in the low-order word and a result code in the high-order word. To access these two types of information, you can use the `HiWord` and `LoWord` functions. For example:

```
VAR
   x: LongInt;
   high, low: Integer;
   high := HiWord(x);
   low := LoWord(x);
```

The `HiWord` function returns the high-order word of the long word passed in, and the `LoWord` function returns the low-order word of the long word passed in. You can use these functions with types other than `LongInt` and `Integer`, as long as they are 4 bytes and 2 bytes, respectively, and, if you are using Pascal, you cast the quantities to the correct types.

The Operating System does not provide any routines that allow you to set the high-order or low-order words of a long integer. It might seem that you could set the low-order word by calling the `BitAnd` function with the original long integer and the low-order word as parameters, and set the high-order word by calling `BitAnd` with the original long integer and the high-order word shifted left 16 bytes as parameters. The problem with this approach is that when you pass an integer variable to `BitAnd`, the compiler automatically casts the variable to a long integer. But for both integers and long integers, it is the leftmost byte that indicates the sign of the number. So when a negative integer is cast to a long integer, the low-order word of the long integer is not equal to the original integer.

However, you can use the Memory Manager's `BlockMove` procedure to directly copy the bytes of a word to the high-order or low-order word of a long word. See *Inside Macintosh: Memory* for more information. Or, if you wish to set both the high-order word and the low-order word of a long integer at once, you can define the following type:

```
TYPE MyLongWordType =
PACKED RECORD
   myHiWord:      Integer;       {high-order word}
   myLoWord:      Integer;       {low-order word}
END;
```

Then you can define a variable of this type and set the high-word and low-word fields. By casting a long integer to `MyLongWordType`, you could also extract a word from a long word more efficiently than you can using the `HiWord` and `LoWord` functions.

Hardcoding Byte Values

Occasionally, you might need to set a group of bytes in memory to specific hexadecimal values. For example, suppose your application uses a data structure with a 16-byte flags field and you wish to initialize each of the bytes in the flags field to particular values. While there are a number of ways that you might do this, the `StuffHex` procedure provides a simple, though usually inefficient, option.

You provide a pointer to any data structure in memory, and a string of hexadecimal digits as parameters to the `StuffHex` procedure. For example:

```
StuffHex(@x, 'D34E0F29');
```

Of course, it would in this case be just as easy—and more efficient—to write the following code:

```
x := $D34E0F29;
```

The `StuffHex` procedure is perhaps most useful when you wish to assign a large or odd number of bytes or set the values of particular bytes within a variable. For example, to set the low-order word of a long integer `x` to $64B5, you could use the following code:

```
StuffHex(Ptr(ORD4(@x) + 2), '64B5');
```

You could use this code rather than use the techniques described in the previous section, "Extracting a Word From a Long Word."

Note that `Ptr` and `ORD4` are used here simply to satisfy Pascal type-casting rules.

The `StuffHex` procedure might also be useful if you are developing a calculator or other application that allows users to enter hexadecimal values directly.

Compressing Data

The PackBits and UnpackBits procedures, introduced in “Data Compression” on page 3-8, allow you to compress (or decompress) data stored in RAM. Typically, you use PackBits before writing data to disk and UnpackBits immediately after writing data from disk.

Both procedures require that you pass in the srcPtr and dstPtr parameters values that point to the beginning of the source buffer and the destination buffer, respectively. The PackBits procedure compresses the data in the source buffer and stores the result in the destination buffer; the UnpackBits procedure decompresses the data in the source buffer and stores the result in the destination buffer. You must also pass to the PackBits procedure and the UnpackBits procedure a value that specifies the size of the original, uncompressed data. Because you must pass this information to UnpackBits, you typically use these procedures only to compress a data structure with a fixed size, so that this size can be passed as a parameter to PackBits.

Your application is responsible for allocating memory for both the source and the destination buffers. When PackBits and UnpackBits complete operation, the srcPtr and dstPtr parameter are incremented so that srcPtr points to the memory immediately following the source bytes, and dstPtr points to the data immediately following the destination bytes. This feature was originally designed to allow you to pack large buffers of data at once in chunks, although PackBits can automatically chunk large data buffers in versions of system software 6.0.2 and later. In any case, your application must store copies of srcPtr and dstPtr to access the start of the source or destination buffer after calling PackBits or UnpackBits.

One use of the compression routines might be to compress resources in your application’s resource fork. Many types of resources can be made significantly smaller by compression. Listing 3-3 shows how you can pack data stored in a handle to a specified resource.

Listing 3-3 Packing data to a resource

```
PROCEDURE MyAddPackedResource (srcData: Handle; theType: ResType;
                               theID: Integer; name: Str255);
VAR
   srcBytes:        Integer;                {bytes of unpacked data}
   maxDstBytes:     LongInt;                {maximum length of packed data}
   dstData:         Handle;                 {packed data}
   srcPtr:          Ptr;                    {pointer to unpacked data}
   dstPtr:          Ptr;                    {pointer to packed data}
   srcProperties:   SignedByte;             {properties of source handle}
BEGIN
   srcBytes := GetHandleSize(srcData);      {find size of source}
                                            {calculate maximum possible }
                                            { size of packed data}
```

```
   maxDstBytes := srcBytes + (srcBytes + 126) DIV 127;
   dstData := NewHandle(maxDstBytes + 2);    {allocate memory for source, }
                                             { plus length info}
   IF dstData <> NIL THEN                    {check for NIL handle}
   BEGIN
      BlockMove(@srcBytes, dstData^, 2);     {copy source into buffer}
      srcPtr := srcData^;                    {copy source pointer}
      dstPtr := Ptr(ORD4(dstData^) + 2);     {copy destination pointer}
      PackBits(srcPtr, dstPtr, srcBytes);    {pack source to destination}
                                             {shrink destination data}
      SetHandleSize(dstData, ORD4(dstPtr) - ORD4(dstData^));
      srcProperties := HGetState(srcData);   {get source handle properties}
      IF BitTst(@srcProperties, 2) THEN      {is source a real resource?}
         RemoveResource(srcData);            {remove current resource}
                                             {add to resource file}
      AddResource(dstData, theType, theID, name);
      WriteResource(dstData);                {write resource data}
      DetachResource(dstData);               {detach from resource map}
      DisposeHandle(dstData);                {dispose of destination data}
   END;
END;
```

The `MyAddPackedResource` procedure declared in Listing 3-3 initially allocates a destination buffer to hold compressed data that is big enough to hold the compressed data in a worst-case scenario, plus 2 bytes to store information at the beginning of the resource about the size of the source data. Because `PackBits` does not move memory, the handle storing the destination buffer does not need to be locked. However, to prevent the `PackBits` procedure from changing the value of a master pointer, you should only pass copies of the dereferenced handle to the procedure. After `PackBits` returns, `MyAddPackedResource` determines how much memory the compressed data takes up by computing how much the `dstPtr` variable has changed. `MyAddPackedResource` then resizes the handle containing the compressed data to the appropriate size. Finally, `MyAddPackedResource` writes the new resource, after first removing the existing resource if the source handle is a handle to a resource. For more information on resources, see *Inside Macintosh: More Macintosh Toolbox*.

Having used the `MyAddPackedResource` procedure to compress resource data, your application needs to be able read the resource and decompress it using the `UnpackBits` procedure. Listing 3-4 shows how you might accomplish this.

Listing 3-4 Decompressing data from a packed resource

```
FUNCTION MyGetPackedResource (theType: ResType; theID: Integer): Handle;
VAR
   srcData:           Handle;                   {handle to packed data}
```

```
   dstData:          Handle;                    {handle to unpacked data}
   srcPtr:           Ptr;                       {pointer to packed data}
   dstPtr:           Ptr;                       {pointer to unpacked data}
   dstBytes:         Integer;                   {number of unpacked bytes}
BEGIN
   srcData := GetResource(theType, theID);      {get the resource}
   BlockMove(srcData^, @dstBytes, 2);           {read number of bytes of }
                                                { unpacked data}
   dstData := NewHandle(dstBytes);              {allocate memory for }
                                                { unpacked data}
   IF dstData <> NIL THEN
   BEGIN
      srcPtr := Ptr(ORD4(srcData^) + 2);        {copy source pointer}
      dstPtr := dstData^;                       {copy destination pointer}
      UnpackBits(srcPtr, dstPtr, dstBytes);     {unpack source to }
                                                { destination}
   END;
   IF srcData <> NIL THEN                       {if there was a resource}
   BEGIN
      DetachResource(srcData);                  {detach from resource map}
      DisposeHandle(srcData);                   {dispose the resource}
   END;
   MyGetPackedResource := dstData;              {return destination handle}
END;
```

The `MyGetPackedResource` function reads in a resource that has previously been packed, determines the size of the unpacked data by copying the first 2 bytes of the resource data, and allocates a relocatable block of this size. The remainder of the data is unpacked using the `UnpackBits` procedure, and the original packed resource data is disposed of.

Obtaining Pseudorandom Numbers

The `Random` function makes it easy to obtain pseudorandom numbers. Before you use `Random`, however, you should seed the pseudo-random number generator. Listing 3-5 shows a common technique for doing this.

Listing 3-5 Seeding the pseudo-random number generator

```
PROCEDURE MySeedGenerator;
BEGIN
   GetDateTime(randSeed);
END;
```

The MySeedGenerator procedure defined in Listing 3-5 simply uses the Date and Time Utilities' GetDateTime procedure to copy the number of seconds since midnight, January 1, 1904, to the global variable randSeed. You might use some other volatile long-word value—such as the mouse location—to seed the pseudo-random number generator, or you might even take a word from one source and a word from another. However, just using GetDateTime is sufficient for most applications.

Sometimes you wish to obtain a pseudo-random integer from a large range of integers; for example, you might need a pseudo-random integer in the range of –20,000 to 20,000. Listing 3-6 shows how you might do this.

Listing 3-6 A simple way of obtaining a large random integer from a range of pseudo-random numbers

```
FUNCTION MyRandomLargeRange (min, max: Integer): Integer;
VAR
   randInt:       Integer;
BEGIN
   REPEAT
      randInt := Random
   UNTIL (randInt >= min) AND (randInt <= max);
   MyRandomLargeRange := randInt;
END;
```

The MyRandomLargeRange function defined in Listing 3-6 simply calls the Random function until it returns an acceptable value. This approach is efficient when you need a random integer from a range of integers that is wide, though not quite as wide as the range the Random function returns by default. However, if you need a random number from a small range—for example, a random number from 1 to 10—the MyRandomLargeRange function is inefficient. Listing 3-7 shows an alternative approach.

Listing 3-7 Obtaining a pseudo random integer from a small range of numbers

```
FUNCTION MyRandomRange (min, max: Integer): Integer;
CONST
   kMinRand = -32767.0;
   kMaxRand = 32767.0;
VAR
   myRand:     Integer;
   x:          Real;          {Random scaled to [0..1]}
BEGIN
   {find random number, and scale it to [0.0..1.0]}
   x := (Random - kMinRand) / (kMaxRand + 1.0 - kMinRand);
```

```
   {scale x to [min, max + 1.0], truncate, and return result}
   MyRandomRange := TRUNC(x * (max + 1.0 - min) + min);
END;
```

The `MyRandomRange` function defined in Listing 3-7 first scales the integral value returned by the `Random` function to a floating-point value from 0 up to, but not including, 1. The function then scales the result to a real number greater than or equal to `min` but less than `max + 1`. By truncating extra decimal places, the correct result is achieved. Note that to force the compiler to perform floating-point calculations, all constants in the function are expressed as real numbers rather than as integers.

Sometimes an application might require a pseudo-random long integer. Listing 3-8 shows how you can do this.

Listing 3-8 Obtaining a pseudo-random long integer

```
FUNCTION MyRandomLongInt: LongInt;
TYPE
   MyLongWordType = PACKED RECORD
      myHiWord:   Integer;             {high-order word}
      myLoWord:   Integer;             {low-order word}
   END;
VAR
   myLongWord:    MyLongWordType;      {random long word}
BEGIN
   {obtain random high-order word}
   myLongWord.myHiWord := Random;
   {obtain random low-order word}
   myLongWord.myLoWord := Random;
   {cast and return result}
   MyRandomLongInt := LongInt(myLongWord);
END;
```

The `MyRandomLongInt` function defined in Listing 3-8 uses a technique discussed in "Extracting a Word From a Long Word" on page 3-18 to stuff a pseudo-random number in the high-order word of a long integer and another pseudo-random number in the low-order word of the long integer. If you need to obtain a long integer within a specified range, you can define routines analogous to Listing 3-6 and Listing 3-7 but use the `MyRandomLongInt` function in place of the `Random` function.

Using Fixed-Point Data Types

Most high-level language compilers include built-in support for the `Fixed` and `Fract` data types so that you can perform regular mathematical operations with fixed-point variables. Also, the algorithms for performing addition and subtraction on `Fixed` and

Fract variables are the same as the algorithms for performing such operations on variables of type LongInt.

The Operating System, however, includes several routines that allow you to convert Fixed and Fract variables to other formats, including SANE's Extended data type, and allow you to perform some simple operations on Fixed and Fract variables. If you need more sophisticated numeric functions, consult the *Apple Numerics Manual*.

To perform multiplication and division of fixed-point numbers, you can use the FixMul, FixDiv, FracMul, and FracDiv functions, which allow you to multiply Fixed point numbers with each other or with other long integers.

You can multiply and divide 32-bit quantities of different types using these functions. The format of the result in this case depends on the particular function being used. See descriptions of the individual functions in "Multiplying and Dividing Fixed-Point Numbers" beginning on page 3-38 for more information.

Using the FracSqrt, FracCos, FracSin, and FixATan2 functions, you can perform a few special arithmetic operations involving variables of type Fixed and Fract.

The FracSqrt function allows you to obtain the square root of a variable of type Fract, interpreting bit 0 as having weight 2 rather than –2. The FracCos and FracSin provide support for the trigonometric cosine and sine functions. The FixATan2 function provides support for the arctangent function. The arguments to all of these functions should be expressed in radians, not in degrees.

Note

To provide fast trigonometric approximations, these trigonometric functions use values of π correct only to 4 decimal places. You should thus use alternative SANE routines when you require better precision. u

To convert among 32-bit numeric types, you can use the Long2Fix, Fix2Long, Fix2Frac, and Frac2Fix functions.

Each of the functions returns its parameter converted into the appropriate format.

You can also convert fixed-point values to and from the SANE Extended floating-point type using the Fix2X, X2Fix, Frac2X, and X2Frac functions.

Two additional functions, FixRatio and FixRound, allow you to perform special conversions on variables of type Fixed.

The FixRatio function returns the fixed-point quotient of the numer and denom parameters. The FixRound function rounds a variable of type Fixed to the nearest integer. If the value is halfway between two integers (0.5), it is rounded to the integer with the higher absolute value. To round a negative fixed-point number, negate it, round it, and then negate it again.

Note

To convert a variable of type Fixed to a variable of type Integer simply use the HiWord function to extract the integral component of the fixed-point number. u

The Operating System also provides the LongMul procedure that allows you to multiple two 32-bit quantities and obtain a 64-bit quantity.

Table 3-2 summaries the routines that perform operations on the Fixed and Fract data types.

Table 3-2 Routines for fixed-point data types

Routine	Description
FixMul	Multiply a variable of type Fixed with another variable of type Fixed or with a variable of type Fract or LongInt
FixDiv	Divide two variables of the same type (Fixed, Fract, or LongInt) or divide a LongInt or Fract number by a Fixed number
FracMul	Multiply a variable of type Fract with another variable of type Fract or with a variable of type Fixed or LongInt
FracDiv	Divide two variables of the same type (Fixed, Fract, or LongInt) or divide a LongInt or Fixed number by a Fract number
FracSqrt	Compute the square root of a variable of type Fract
FracCos	Obtain the cosine of a variable of type Fixed
FracSin	Obtain the sine of a variable of type Fixed
FixATan2	Obtain the arctangent of a variable of type Fixed, Fract, or LongInt
Long2Fix	Convert a variable of type LongInt to Fixed
Fix2Long	Convert a variable of type Fixed to LongInt
Fix2Frac	Convert a variable of type Fixed to Fract
Frac2Fix	Convert a variable of type Fract to Fixed
Fix2X	Convert a variable of type Fixed to Extended
X2Fix	Convert a variable of type Extended to Fixed
Frac2X	Convert a variable of type Fract to Extended
X2Frac	Convert a variable of type Extended to Fract
FixRatio	Obtain the Fixed equivalent of a fraction
FixRound	Round a fixed-point number to the nearest integer
LongMul	Multiply two 32-bit quantities and obtain a 64-bit quantity

Mathematical and Logical Utilities Reference

This section provides a complete reference to the Mathematical and Logical Utilities routines provided by the Macintosh Operating System. The section “Data Structures” describes the 64-bit integer record. The section “Routines” describes the routines that the Operating System includes to allow you to perform simple mathematical and logical operations.

Data Structures

This section describes the 64-bit integer record. For information on the numeric formats of fixed-point numbers, see “Fixed-Point Data Types” beginning on page 3-11. For information on the format of other numeric data types, consult the *Apple Numerics Manual*.

64-Bit Integer Record

By using the `LongMul` procedure, you can multiply two 32-bit quantities and obtain a 64-bit quantity stored in a 64-bit integer record. The `Int64Bit` data type defines a 64-bit integer record.

```
TYPE Int64Bit =
RECORD
   hiLong: LongInt;
   loLong: LongInt;
END;
```

Field descriptions

`hiLong`	The high-order long integer of the 64-bit integer.
`loLong`	The low-order long integer of the 64-bit integer.

Routines

This section describes the Mathematical and Logical Utilities supported directly by the Macintosh Operating System. Note that none of the routines in this section moves memory; therefore, all of the described routines in this section can be called at interrupt time.

Testing and Setting Bits

This section describes the `BitTst` function and the `BitSet` and `BitClr` procedures. You can test a bit using `BitTst` and specify a bit's value using `BitSet` and `BitClr`. All three of these procedures use the reversed bit-numbering scheme described in the section "Reversed Bit-Numbering" on page 3-7.

BitTst

You can use the `BitTst` function to determine whether a given bit is set.

```
FUNCTION BitTst (bytePtr: Ptr; bitNum: LongInt): Boolean;
```

`bytePtr` A pointer to a byte in memory.

`bitNum` The bit to be tested, specified as a positive offset from the high-order bit of the byte pointed to by the `bytePtr` parameter. The bit being tested need not be in the byte pointed to by `bytePtr`.

DESCRIPTION

The `BitTst` function returns `TRUE` if the bit specified by the `bytePtr` and `bitNum` parameters is set (that is, has a value of 1) and returns `FALSE` if the specified bit is cleared (that is, has a value of 0).

SPECIAL CONSIDERATIONS

The bit-numbering scheme used by the `BitTst` function is the opposite of MC680x0 bit numbering. To convert an MC680x0 bit number to the format required by the `BitTst` function, subtract the MC680x0 bit number from the highest bit number.

SEE ALSO

For an example of the use of the `BitTst` function, see Listing 3-2 on page 3-15. For more information about reversed bit-numbering see, "Reversed Bit-Numbering" on page 3-7.

BitSet

You can use the `BitSet` procedure to set a particular bit.

```
PROCEDURE BitSet (bytePtr: Ptr; bitNum: LongInt);
```

`bytePtr` A pointer to a byte in memory.

bitNum — The bit to be set, specified as a positive offset from the high-order bit of the byte pointed to by the bytePtr parameter. The bit being set need not be in the byte pointed to by bytePtr.

DESCRIPTION

The BitSet procedure sets (to a value of 1) the bit specified by the bytePtr and bitNum parameters.

SPECIAL CONSIDERATIONS

The bit-numbering scheme used by the BitSet procedure is the opposite of MC680x0 bit numbering. To convert an MC680x0 bit number to the format required by the BitSet procedure, subtract the MC680x0 bit number from the highest bit number.

SEE ALSO

For an example of the use of the BitSet procedure, see page 3-16. For more information about reversed bit-numbering see “Reversed Bit-Numbering” on page 3-7.

BitClr

You can use the BitClr procedure to clear a particular bit.

```
PROCEDURE BitClr (bytePtr: Ptr; bitNum: LongInt);
```

bytePtr — A pointer to a byte in memory.

bitNum — The bit to be cleared, specified as a positive offset from the high-order bit of the byte pointed to by the bytePtr parameter. The bit being cleared need not be in the same byte pointed to by bytePtr.

DESCRIPTION

The BitClr procedure clears (to a value of 0) the bit specified by the bytePtr and bitNum parameters.

SPECIAL CONSIDERATIONS

The bit-numbering scheme used by the BitClr procedure is the opposite of MC680x0 bit numbering. To convert an MC680x0 bit number to the format required by the BitClr procedure, subtract the MC680x0 bit number from the highest bit number.

SEE ALSO

For an example of the use of the `BitClr` procedure, see page 3-16. For more information about reversed bit-numbering, see "Reversed Bit-Numbering" on page 3-7.

Performing Logical Operations

The Operating System supports five functions to support bit-level logical operations. The `BitAnd`, `BitOr`, `BitXor`, `BitNot`, and `BitShift` functions perform AND, OR, XOR, NOT, and bit-shifting operations, respectively. These routines are intended primarily for Pascal programmers. If you are programming in C, you can typically use C operators to perform the same logical operations more efficiently.

BitAnd

You can use the `BitAnd` function to perform the AND logical operation on two long words.

```
FUNCTION BitAnd (value1, value2: LongInt): LongInt;
```

`value1` A long word.

`value2` A long word.

DESCRIPTION

The `BitAnd` function returns a long word that is the result of performing the AND operation on the long words specified by the `value1` and `value2` parameters. Each bit in the returned value is set if and only if the corresponding bit is set in both `value1` and `value2`.

SEE ALSO

For an illustration of the result of performing an operation using the `BitAnd` function, see Figure 3-9 on page 3-16.

BitOr

You can use the `BitOr` function to perform the OR logical operation on two long words.

```
FUNCTION BitOr (value1, value2: LongInt): LongInt;
```

`value1` A long word.

`value2` A long word.

DESCRIPTION

The `BitOr` function returns a long word that is the result of performing the `OR` operation on the long words specified by the `value1` and `value2` parameters. Each bit in the returned value is set if and only if the corresponding bit is set in `value1` or `value2`, or in both `value1` and `value2`.

SEE ALSO

For an illustration of the result of performing an operation using the `BitOr` function, see Figure 3-9 on page 3-16.

BitXor

You can use the `BitXor` function to perform the `XOR` logical operation on two long words.

```
FUNCTION BitXor (value1, value2: LongInt): LongInt;
```

`value1` A long word.

`value2` A long word.

DESCRIPTION

The `BitXor` function returns a long word that is the result of performing the `XOR` operation on the long words specified by the `value1` and `value2` parameters. Each bit in the returned value is set if and only if the corresponding bit is set in either `value1` or `value2`, but not in both `value1` and `value2`.

SEE ALSO

For an illustration of the result of performing an operation using the `BitXor` function, see Figure 3-9 on page 3-16.

BitNot

You can use the `BitNot` function to perform the `NOT` logical operation on a long word.

```
FUNCTION BitNot (value: LongInt): LongInt;
```

`value` A long word.

DESCRIPTION

The `BitNot` function returns a long word that is the result of performing the `NOT` operation on the long word specified by the `value` parameter. Each bit in the returned value is set if and only if the corresponding bit is not set in `value`.

SEE ALSO

For an illustration of the result of performing an operation using the `BitNot` function, see Figure 3-10 on page 3-17.

BitShift

You can use the `BitShift` function to shift bits in a long word.

```
FUNCTION BitShift (value: LongInt; count: Integer): LongInt;
```

`value`	A long word.
`count`	The number of bits to shift. If this number is positive, `BitShift` shifts this many positions to the left; if this number is negative, `BitShift` shifts this many positions to the right. The value in this parameter is converted to the result of `MOD` 32.

DESCRIPTION

The `BitShift` function returns a long word that is the result of shifting the bits in the long word specified by the `value` parameter. The shift's direction and extent are determined by the `count` parameter. Zeroes are shifted into empty positions regardless of the direction of the shift.

SEE ALSO

For an illustration of the result of performing an operation using the `BitShift` function, see Figure 3-10 on page 3-17.

Getting and Setting Memory Values

The `HiWord` and `LoWord` functions allow you to extract a word from a long word. The `StuffHex` procedure provides a quick way to convert hexadecimal values stored in a string into byte values in memory.

To copy a range of bytes from one memory location to another, you should ordinarily use the Memory Manager's `BlockMove` procedure, which is described in *Inside Macintosh: Memory*.

HiWord

You can use the `HiWord` function to obtain the high-order word of a long word. One use of this function is to obtain the integral part of a fixed-point number.

```
FUNCTION HiWord (x: LongInt): Integer;
```

`x` The long word whose high word is to be returned.

DESCRIPTION

The `HiWord` function returns the high-order word of the long word specified by the `x` parameter.

LoWord

You can use the `LoWord` function to obtain the low-order word of a long word. One use of this function is to obtain the fractional part of a fixed-point number.

```
FUNCTION LoWord (x: LongInt): Integer;
```

`x` The long word whose low word is to be returned.

DESCRIPTION

The `LoWord` function returns the low-order word of the long word specified by the `x` parameter.

StuffHex

You can use the `StuffHex` procedure to hardcode byte values into memory.

```
PROCEDURE StuffHex (thingPtr: Ptr; s: Str255);
```

`thingPtr` A pointer to any data structure in memory. If `thingPtr` is an odd address, then `thingPtr` is interpreted as pointing to the next word boundary.

`s` A string of characters representing hexadecimal digits. Be sure that all characters in this string are hexadecimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F). Otherwise, `StuffHex` may set bytes in the data structure pointed to by `thingPtr` to arbitrary values. If there are an odd number of characters in the string, the last character is ignored.

DESCRIPTION

The StuffHex procedure sets bytes in memory beginning with that byte specified by the parameter thingPtr. The total number of bytes set is equivalent to s[0] DIV 2 (that is, half the length of the string, ignoring the last character if the number of characters is odd).

Each byte to be set corresponds to two characters in the string. These characters should represent hexadecimal digits. For example, the string 'D41A' results in 2 bytes being set to the values $D4 and $1A, respectively.

Although the StuffHex procedure sets the value of individual bytes, it does not move relocatable blocks. Thus, you can call it at interrupt time.

SPECIAL CONSIDERATIONS

The StuffHex procedure does no range checking to ensure that bytes being set are within the bounds of a certain data structure. If you do not use StuffHex carefully, you may change memory in the partition of your application or another application in unpredictable ways.

SEE ALSO

For examples of the use of the StuffHex procedure, see page 3-19.

Compressing and Decompressing Data

You can use the PackBits function to compress a source buffer of data into a destination buffer and the UnpackBits function to decompress a source buffer of PackBits-compressed data into a destination buffer.

PackBits

You can use the PackBits procedure to compress a data buffer stored in RAM.

```
PROCEDURE PackBits (VAR srcPtr, dstPtr: Ptr; srcBytes: Integer);
```

srcPtr	On entry, a pointer to the first byte of a buffer of data to be compressed. On exit, a pointer to the first byte following the bytes compressed.
dstPtr	On entry, a pointer to the first byte in which to store compressed data. On exit, a pointer to the first byte following the compressed data.
srcBytes	The number of bytes of uncompressed data to be compressed. In versions of software prior to version 6.0.2, this number must be 127 or less.

DESCRIPTION

The PackBits procedure compresses srcBytes bytes of data beginning at the location specified by the srcPtr parameter and stores it at the location specified by the dstPtr parameter. It then modifies the srcPtr and dstPtr variables to point to the first bytes after the uncompressed and compressed data, respectively.

Your application must allocate memory for the destination buffer itself. In general, you should allocate enough memory for a worst-case scenario. In the worst case, the destination buffer is 128 bytes long for each block of source data up to 127 bytes. Thus, you can use the following formula to determine how much space to allocate for the destination buffer:

```
maxDstBytes := srcBytes + (srcBytes+126) DIV 127;
```

where maxDstBytes stands for the maximum number of destination bytes.

The PackBits algorithm is most effective on data buffers in which there are likely to be series of bytes containing the same value. For example, resources of many formats often contain many consecutive zeros. If you have a data buffer in which there are only likely to be series of words or long words containing the same value, PackBits is unlikely to be effective.

Because your application must allocate memory for the source and destination buffers, PackBits does not move relocatable blocks. Thus, you can call it at interrupt time.

SPECIAL CONSIDERATIONS

Because PackBits changes the values of the srcPtr and dstPtr parameters, you should pass to PackBits only copies of pointers to the source and destination buffers. This allows you to access the beginning of the source and destination buffers after PackBits returns. Also, if the source or destination buffer is stored in an unlocked, relocatable block, this technique prevents PackBits from changing the value of a master pointer, which would make the original handle invalid.

SEE ALSO

For an example of the use of the PackBits procedure, see Listing 3-3 on page 3-20.

UnpackBits

You can use the UnpackBits procedure to decompress a data buffer containing data compressed by PackBits.

```
PROCEDURE UnpackBits (VAR srcPtr, dstPtr: Ptr; dstBytes: Integer);
```

srcPtr — On entry, a pointer to the first byte of a buffer of data to be decompressed. On exit, a pointer to the first byte following the compressed data.

`dstPtr` On entry, a pointer to the first byte in which to store decompressed data. On exit, a pointer to the first byte following the decompressed data.

`dstBytes` The number of bytes of the data before compression. In general, you should either use `PackBits` to compress data structures of a fixed size that you can then pass in this parameter to `UnpackBits`, or store with the compressed data the original size of the uncompressed data.

DESCRIPTION

The `UnpackBits` procedure decompresses `srcBytes` bytes of data beginning at the location specified by the `srcPtr` parameter and stores it at the location specified by the `dstPtr` parameter. It then modifies the `srcPtr` and `dstPtr` variables to point to the first bytes after the compressed and decompressed data, respectively.

Because your application must allocate memory for the source and destination buffers, `UnpackBits` does not move relocatable blocks. Thus, you can call it at interrupt time.

SPECIAL CONSIDERATIONS

Because `UnpackBits` changes the values of the `srcPtr` and `dstPtr` parameters, you should pass to `UnpackBits` only copies of pointers to the source and destination buffers. This allows you to access the beginning of the source and destination buffers after `UnpackBits` returns. Also, if the source or destination buffer is stored in an unlocked, relocatable block, this technique prevents `UnpackBits` from changing the value of a master pointer, which would make the original handle invalid.

SEE ALSO

For an example of the use of the `UnpackBits` procedure, see Listing 3-4 on page 3-21.

Obtaining a Pseudorandom Number

You can gain access to the Operating System's pseudorandom number generator by using the `Random` function.

Random

You can use the `Random` function to obtain a pseudorandom integer.

```
FUNCTION Random: Integer;
```

DESCRIPTION

The `Random` function returns a pseudorandom integer, uniformly distributed in the range –32767 to 32767.

The value Random returns depends solely on the global variable randSeed, which the QuickDraw InitGraf procedure initializes to 1. Each time the Random function executes, it uses a numerical algorithm to change the value of randSeed to prevent it from returning the same value each time it is called.

To prevent your application from generating the same sequence of pseudo-random numbers each time it is executed, initialize the randSeed global variable, when your application starts up, to a volatile long word variable such as the current date and time. If you would like to generate the same sequence of pseudo-random numbers twice, on the other hand, simply set randSeed to the same value before calling Random for each sequence.

ASSEMBLY-LANGUAGE INFORMATION

You can access the global variable randSeed through the system global variable RndSeed.

SEE ALSO

Listing 3-5 on page 3-22, Listing 3-6 on page 3-23, Listing 3-7 on page 3-23, and Listing 3-8 on page 3-24 for examples of how to use the Random function.

Converting Between Angle and Slope Values

You can use the SlopeFromAngle and AngleFromSlope functions to convert between angle and slope values.

SlopeFromAngle

You can convert an angle value to a slope value using the SlopeFromAngle function.

```
FUNCTION SlopeFromAngle (angle: Integer): Fixed;
```

angle — The angle, expressed in clockwise degrees from 12 o'clock and treated MOD 180. (90 degrees is thus at 3 o'clock and –90 degrees is at 9 o'clock.)

DESCRIPTION

The SlopeFromAngle function returns the slope corresponding to the angle specified in the angle parameter. Slopes are defined as $\Delta x/\Delta y$, the horizontal change divided by the vertical change between any two points on a line with the given angle. The negative y-axis is defined as being at 12 o'clock, and the positive y-axis at 6 o'clock. The x-axis is defined as usual, with the positive side defined as being at 3 o'clock.

SEE ALSO

For an example of the use of the `SlopeFromAngle` function, see Figure 3-7 on page 3-13.

AngleFromSlope

You can convert a slope value to an angle value using the `AngleFromSlope` function.

```
FUNCTION AngleFromSlope (slope: Fixed): Integer;
```

`slope` The slope, defined as Δx / Δy, which is the horizontal change divided by the vertical change between any two points on a line with the slope.

DESCRIPTION

The `AngleFromSlope` function returns the angle corresponding to the slope specified in the `slope` parameter treated `MOD` 180. Angles are defined in clockwise degrees from 12 o'clock. The negative y-axis is defined as being at 12 o'clock, and the positive y-axis at 6 o'clock. The x-axis is defined as usual, with the positive side defined as being at 3 o'clock.

SPECIAL CONSIDERATIONS

The `AngleFromSlope` function is most useful when you require speed more than accuracy in performing the calculation. The integer result is within 1 degree of the correct answer, but not necessarily within half a degree.

SEE ALSO

For an example of the use of the `AngleFromSlope` function, see Figure 3-7 on page 3-13.

Multiplying and Dividing Fixed-Point Numbers

The `FixMul` and `FracMul` functions allow you to multiply fixed-point numbers. The `FixDiv` and `FracDiv` functions allow you to divide fixed-point numbers. By performing appropriate type casting, you can multiply or divide a fixed-point number of one type with a fixed-point number of another type or a long integer.

FixMul

You can use the `FixMul` function to multiply a variable of type `Fixed` with another variable of type `Fixed` or with a variable of type `Fract` or `LongInt`.

```
FUNCTION FixMul (a, b: Fixed): Fixed;
```

a — The first operand, which can be a variable of type Fixed or a variable of type Fract or LongInt.

b — The second operand, which can be a variable of type Fixed or a variable of type Fract or LongInt.

DESCRIPTION

The FixMul function returns the product of the numbers specified in the a and b parameters. At least one of a and b should be a variable of type Fixed.

The returned value is in the format of a LongInt if one of a or b is a LongInt. It is a Fract number if one of a or b is Fract. It is a Fixed number if both a and b are Fixed numbers.

Overflows are set to the maximum representable value with the correct sign ($80000000 for negative results and $7FFFFFFF for positive results).

SEE ALSO

For a summary of the routines that perform operations on the Fixed and Fract data type, see Table 3-2 on page 3-26.

FixDiv

You can use the FixDiv function to divide two variables of the same type (Fixed, Fract, or LongInt) or to divide a LongInt or Fract number by a Fixed number.

```
FUNCTION FixDiv (a, b: Fixed): Fixed;
```

a — The first operand, which can be a variable of type Fixed or a variable of type Fract or LongInt.

b — The second operand, which can be a variable of type Fixed or it can be a variable of the same type as the variable in parameter a.

DESCRIPTION

The FixDiv function returns the quotient of the numbers specified in the a and b parameters. If the b parameter is in the format of a Fixed number, then the a parameter can be in the format of a Fixed, Fract, or LongInt number. If the b parameter is in the format of a Fract or LongInt number, then the a parameter must be in the same format.

The returned value is in the format of a Fixed number if both a and b are both Fixed numbers, both Fract numbers, or both LongInt numbers. Otherwise, the returned value is the same type as the number in the a parameter.

Division by zero results in $8000000 if `a` is negative, and $7FFFFFFF otherwise; thus the special case 0/0 yields $7FFFFFFF.

SEE ALSO

For a summary of the routines that perform operations on the Fixed and Fract data type, see Table 3-2 on page 3-26.

FracMul

You can use the `FracMul` function to multiply a variable of type `Fract` with another variable of type `Fract` or with a variable of type `Fixed` or `LongInt`.

```
FUNCTION FracMul (a, b: Fract): Fract;
```

`a` The first operand, which can be a variable of type `Fract` or a variable of type `Fixed` or `LongInt`.

`b` The second operand, which can be a variable of type `Fract` or a variable of type `Fixed` or `LongInt`.

DESCRIPTION

The `FracMul` function returns the product of the numbers specified in the `a` and `b` parameters. At least one of `a` or `b` should be a variable of type `Fract`.

The returned value is in the format of a `LongInt` number if one of `a` and `b` is a `LongInt` number. It is a `Fixed` number if one of `a` or `b` is a `Fixed` number. It is a `Fract` number if both `a` and `b` are `Fract` numbers.

Overflows are set to the maximum representable value with the correct sign ($80000000 for negative results and $7FFFFFFF for positive results).

SEE ALSO

For a summary of the routines that perform operations on the Fixed and Fract data type, see Table 3-2 on page 3-26.

FracDiv

You can use the `FracDiv` function to divide two variables of the same type (`Fract`, `Fixed`, or `LongInt`) or to divide a `LongInt` or `Fixed` number by a `Fract` number.

```
FUNCTION FracDiv (a, b: Fract): Fract;
```

a — The first operand, which can be a variable of type `Fract` or a variable of type `Fixed` or `LongInt`.

b — The second operand, which can be a variable of type `Fract` or a variable of the same type as the variable in parameter `a`.

DESCRIPTION

The `FracDiv` function returns the quotient of the numbers specified in the `a` and `b` parameters. If the `b` parameter is in the format of a `Fract` number, then the `a` parameter can be in the format of a `Fract`, a `Fixed`, or a `LongInt` number. If the `b` parameter is in the format of a `Fixed` or a `LongInt` number, then the `a` parameter must be in the same format.

The returned value is in the format of a `Fract` number if `a` and `b` are both `Fract` numbers, both `Fixed` numbers, or both `LongInt` numbers. Otherwise, the returned value is in the same format as the number in the `a` parameter.

Division by zero results in $8000000 if `a` is negative, and $7FFFFFFF otherwise; thus the special case 0/0 yields $7FFFFFFF.

Performing Calculations on Fixed-Point Numbers

The Operating System provides four functions that you can use to perform a few common calculations on fixed-point numbers. The `FracSqrt` function allows you to obtain the square root of a number. The `FracCos`, `FracSin`, and `FixATan2` functions allow you to obtain fast approximations of trigonometric functions on fixed-point numbers.

FracSqrt

You can use the `FracSqrt` function to obtain the square root of a `Fract` number.

```
FUNCTION FracSqrt (x: Fract): Fract;
```

x — The `Fract` number to obtain a square root of. This parameter is interpreted as being unsigned in the range 0 through $4 - 2^{-30}$, inclusive. That is, the bit of a `Fract` number that ordinarily has weight –2 is instead interpreted as having weight 2.

DESCRIPTION

The `FracSqrt` function returns the square root of the `Fract` number you supply in the `x` parameter. The result is unsigned in the range 0 through 2, inclusive.

FracCos

You can use the `FracCos` function to obtain a fast approximation of the cosine of a `Fixed` number.

```
FUNCTION FracCos (x: Fixed): Fract;
```

`x` The `Fixed` number expressed in radians, whose cosine is to be calculated.

DESCRIPTION

The `FracCos` function returns the cosine, expressed in radians, of the `Fixed` number `x`.

The approximation of $\pi/4$ used to compute the cosine is the hexadecimal value 0.C910, making the approximation of π equal to 3.1416015625, while π itself equals 3.14159265.... Despite the approximation of π, the cosine value obtained is usually correct to several decimal places.

FracSin

You can use the `FracSin` function to obtain a fast approximation of the sine of a `Fixed` number.

```
FUNCTION FracSin (x: Fixed): Fract;
```

`x` The `Fixed` number expressed in radians, whose sine is to be calculated.

DESCRIPTION

The `FracSin` function returns the sine, expressed in radians, of the `Fixed` number `x`.

The approximation of $\pi/4$ used to compute the sine is the hexadecimal value 0.C910, making the approximation of π equal to 3.1416015625, while π itself equals 3.14159265.... Despite the approximation of π, the sine value obtained is usually correct to several decimal places.

FixATan2

You can use the `FixATan2` function to obtain a fast approximation of the arctangent of a fraction.

```
FUNCTION FixATan2 (x, y: LongInt): Fixed;
```

x The numerator of the fraction whose arctangent is to be obtained. This variable can be a LongInt, Fixed, or Fract number.

y The denominator of the fraction whose arctangent is to be obtained. The number supplied in this variable must be of the same type as that of the number supplied in the x parameter.

DESCRIPTION

The FixATan2 function returns, in radians, the arctangent of y/x.

The approximation of π/4 used to compute the arctangent is the hexadecimal value 0.C910, making the approximation of π equal to 3.1416015625, while π itself equals 3.14159265.... Thus FixATan2(1, 1) equals the equivalent of the hexadecimal value 0.C910. Despite the approximation of π, the arctangent value obtained will usually be correct to several decimal places.

Converting Among 32-Bit Numeric Types

The Operating System includes functions that allow you to convert among variables of type LongInt, Fixed, and Fract. The Long2Fix and Fix2Long functions convert between LongInt variables and Fixed variables. The Fix2Frac functions and Frac2Fix functions convert between Fixed and Fract variables. Ordinarily, there is no need to convert between LongInt and Fract variables, because Fract variables are used only to represent very small numbers. If you wish to do so, however, you can combine functions shown in this section.

Long2Fix

You can use the Long2Fix function to convert a LongInt number to a Fixed number.

```
FUNCTION Long2Fix (x: LongInt): Fixed;
```

x The long integer to be converted to a Fixed number.

DESCRIPTION

The Long2Fix function returns the Fixed number equivalent to the long integer you supply in the x parameter. If x is greater than the maximum representable fixed-point number, the Long2Fix function returns $7FFFFFFF. If x is less than the negative number with the highest absolute value, Long2Fix returns $80000000.

Fix2Long

You can use the `Fix2Long` function to convert a `Fixed` number to a `LongInt` number.

```
FUNCTION Fix2Long (x: Fixed): LongInt;
```

`x` The `Fixed` number to be converted to a long integer.

DESCRIPTION

The `Fix2Long` function returns the long integer nearest to the `Fixed` number you supply in the `x` parameter. If `x` is halfway between two integers (0.5), it is rounded to the integer with the higher absolute value.

Fix2Frac

You can use the `Fix2Frac` function to convert a `Fixed` number to a `Fract` number.

```
FUNCTION Fix2Frac (x: Fixed): Fract;
```

`x` The `Fixed` number to be converted to a `Fract` number.

DESCRIPTION

The `Fix2Frac` function returns the `Fract` number equivalent to the `Fixed` number x. If `x` is greater than the maximum representable `Fract` number, the `Fix2Frac` function returns $7FFFFFFF. If `x` is less than the negative number with the highest absolute value, `Fix2Frac` returns $80000000.

Frac2Fix

You can use the `Frac2Fix` function to convert a `Fract` number to a `Fixed` number.

```
FUNCTION Frac2Fix (x: Fract): Fixed;
```

`x` The `Fract` number to be converted to a `Fixed` number.

DESCRIPTION

The `Frac2Fix` function returns the `Fixed` number that best approximates the `Fract` number you supply in the `x` parameter.

Converting Between Fixed-Point and Floating-Point Values

The Mathematical and Logical Utilities provide four functions that allow you to convert between fixed-point and floating-point values represented using SANE's `Extended` floating-point data type. The `Fix2X` function and the `X2Fix` function convert between `Fixed` and `Extended` numbers. The `Frac2X` and `X2Frac` functions convert between `Fract` and `Extended` numbers. See *Apple Numerics Manual* for information about numeric data types supported by SANE.

Fix2X

You can use the `Fix2X` function to convert a `Fixed` number to an `Extended` number.

```
FUNCTION Fix2X (x: Fixed): Extended;
```

`x` The `Fixed` number to be converted to an `Extended` number.

DESCRIPTION

The `Fix2X` function returns the `Extended` equivalent of the `Fixed` number you supply in the `x` parameter.

SPECIAL CONSIDERATIONS

Because the `Fix2X` function does not move memory, you can call it at interrupt time.

X2Fix

You can use the `X2Fix` function to convert an `Extended` number to a `Fixed` number.

```
FUNCTION X2Fix (x: Extended): Fixed;
```

`x` The `Extended` number to be converted to a `Fixed` number.

DESCRIPTION

The `X2Fix` function returns the best `Fixed` approximation of the `Extended` number you supply in the `x` parameter. If `x` is greater than the maximum representable `Fixed` number, the `X2Fix` function returns $7FFFFFFF. If `x` is less than the negative number with the highest absolute value, `X2Fix` returns $80000000.

Frac2X

You can use the `Frac2X` function to convert a `Fract` number to an `Extended` number.

```
FUNCTION Frac2X (x: Fract): Extended;
```

`x` The `Fract` number to be converted to an `Extended` number.

DESCRIPTION

The `Frac2X` function returns the `Extended` equivalent of the `Fract` number you supply in the `x` parameter.

X2Frac

You can use the `X2Frac` function to convert an `Extended` number to a `Fract` number.

```
FUNCTION X2Frac (x: Extended): Fract;
```

`x` The `Extended` number to be converted to a `Fract` number.

DESCRIPTION

The `X2Frac` function returns the best `Fract` approximation of the `Extended` number you supply in the `x` parameter. If `x` is greater than the maximum representable `Fract` number, the `X2Frac` function returns $7FFFFFFF. If `x` is less than the negative number with the highest absolute value, `X2Frac` returns $80000000.

Converting Between Fixed-Point and Integral Values

To convert the quotient of two integers to a `Fixed` number, you can use the `FixRatio` function. To obtain the integral portion of a number of type `Fixed`, typically you just use the `HiWord` function, described on page 3-33. However, you can also use the `FixRound` function to obtain the integer nearest a fixed-point number.

FixRatio

You can use the `FixRatio` function to obtain the `Fixed` equivalent of a fraction.

```
FUNCTION FixRatio (numer, denom: Integer): Fixed;
```

`numer` The numerator of the fraction.
`denom` The denominator of the fraction.

DESCRIPTION

The `FixRatio` function return the `Fixed` equivalent of the fraction `numer/denom`.

FixRound

You can use the `FixRound` function to round a fixed-point number to the nearest integer.

```
FUNCTION FixRound (x: Fixed): Integer;
```

`x` The `Fixed` number to be rounded.

DESCRIPTION

The `FixRound` function returns the `Integer` number nearest the `Fixed` number you supply in the `x` parameter. If the value is halfway between two integers (0.5), it is rounded up. Thus, 4.5 is rounded to 5, and –3.5 is rounded to –3.

To round a negative `Fixed` number so that values halfway between two integers are rounded to the number with the higher absolute value, negate the number, round it, and then negate it again.

Multiplying 32-bit values

To multiply a 32-bit value and return a 64-bit value, you can use the `LongMul` procedure.

LongMul

You can use the `LongMul` procedure to multiply two 32-bit quantities and obtain a 64-bit quantity.

```
Procedure LongMul (a, b: LongInt; VAR result: Int64Bit);
```

`a` The first operand, which is a variable of type `LongInt`.

`b` The second operand, which is a variable of type `LongInt`.

`result` A pointer to the returned value.

DESCRIPTION

Given two variables of type `LongInt`, the `LongMul` procedure multiplies the two variables specified in parameter `a` and `b`, and returns the value in the variable specified by the `result` parameter.

Summary of the Mathematical and Logical Utilities

Pascal Summary

Data Types

```
TYPE
   Fixed          = LongInt;        {fixed-point number}
   Fract          = LongInt;        {fractional number}

   Int64Bit =                       {64-bit integer record}
   RECORD
      hiLong:     LongInt;          {high-order long integer}
      loLong:     LongInt;          {low-order long integer}
   END;
```

Routines

Testing and Setting Bits

```
FUNCTION BitTst            (bytePtr: Ptr; bitNum: LongInt): Boolean;
PROCEDURE BitSet           (bytePtr: Ptr; bitNum: LongInt);
PROCEDURE BitClr           (bytePtr: Ptr; bitNum: LongInt);
```

Performing Logical Operations

```
FUNCTION BitAnd            (value1, value2: LongInt): LongInt;
FUNCTION BitOr             (value1, value2: LongInt): LongInt;
FUNCTION BitXor            (value1, value2: LongInt): LongInt;
FUNCTION BitNot            (value: LongInt): LongInt;
FUNCTION BitShift          (value: LongInt; count: Integer): LongInt;
```

Getting and Setting Memory Values

```
FUNCTION HiWord            (x: LongInt): Integer;
FUNCTION LoWord            (x: LongInt): Integer;
PROCEDURE StuffHex         (thingPtr: Ptr; s: Str255);
```

Compressing and Decompressing Data

```
PROCEDURE PackBits          (VAR srcPtr, dstPtr: Ptr; srcBytes: Integer);
PROCEDURE UnpackBits        (VAR srcPtr, dstPtr: Ptr; dstBytes: Integer);
```

Obtaining a Pseudorandom Number

```
FUNCTION Random             : Integer;
```

Converting Between Angle and Slope Values

```
FUNCTION SlopeFromAngle     (angle: Integer): Fixed;
FUNCTION AngleFromSlope     (slope: Fixed): Integer;
```

Multiplying and Dividing Fixed-Point Numbers

```
FUNCTION FixMul             (a, b: Fixed): Fixed;
FUNCTION FixDiv             (a, b: Fixed): Fixed;
FUNCTION FracMul            (a, b: Fract): Fract;
FUNCTION FracDiv            (a, b: Fract): Fract;
```

Performing Calculations on Fixed-Point Numbers

```
FUNCTION FracSqrt           (x: Fract): Fract;
FUNCTION FracCos            (x: Fixed): Fract;
FUNCTION FracSin            (x: Fixed): Fract;
FUNCTION FixATan2           (x, y: LongInt): Fixed;
```

Converting Among 32-Bit Numeric Types

```
FUNCTION Long2Fix           (x: LongInt): Fixed;
FUNCTION Fix2Long           (x: Fixed): LongInt;
FUNCTION Fix2Frac           (x: Fixed): Fract;
FUNCTION Frac2Fix           (x: Fract): Fixed;
```

Converting Between Fixed-Point and Floating-Point Values

```
FUNCTION Fix2X              (x: Fixed): Extended;
FUNCTION X2Fix              (x: Extended): Fixed;
FUNCTION Frac2X             (x: Fract): Extended;
FUNCTION X2Frac             (x: Extended): Fract;
```

Converting Between Fixed-Point and Integral Values

```
FUNCTION FixRatio           (numer, denom: Integer): Fixed;
FUNCTION FixRound           (x: Fixed): Integer;
```

Multiplying 32-bit Values

```
Procedure LongMul           (a, b: LongInt; VAR result: Int64Bit);
```

C Summary

Data Types

```
typedef long Fixed;                     /*fixed-point number*/
typedef long Fract;                     /*fractional number*/

struct Int64Bit {                       /*64-bit integer record*/
   long hiLong;                         /*high-order long integer*/
   long loLong;                         /*low-order long integer*/
};
typedef struct Int64Bit Int64Bit;
```

Routines

Testing and Setting Bits

```
pascal Boolean BitTst       (const void *bytePtr, long bitNum);
pascal void BitSet          (void *bytePtr, long bitNum);
pascal void BitClr          (void *bytePtr, long bitNum);
```

Performing Logical Operations

```
pascal long BitAnd          (long value1, long value2);
pascal long BitOr           (long value1, long value2);
pascal long BitXor          (long value1, long value2);
pascal long BitNot          (long value);
pascal long BitShift        (long value, short count);
```

Getting and Setting Memory Values

```
pascal short HiWord         (long x);
pascal short LoWord         (long x);
pascal void StuffHex        (void *thingPtr, ConstStr255Param s);
```

Compressing and Decompressing Data

```
pascal void PackBits        (Ptr *srcPtr, Ptr *dstPtr, short srcBytes);
```

```
pascal void UnpackBits      (Ptr *srcPtr, Ptr *dstPtr, short dstBytes);
```

Obtaining a Pseudorandom Number

```
pascal short Random         (void);
```

Converting Between Angle and Slope Values

```
pascal Fixed SlopeFromAngle
                            (short angle);
pascal short AngleFromSlope
                            (Fixed slope);
```

Multiplying and Dividing Fixed-Point Numbers

```
pascal Fixed FixMul         (Fixed a, Fixed b);
pascal Fixed FixDiv         (Fixed a, Fixed b);
pascal Fract FracMul        (Fract a, Fract b);
pascal Fract FracDiv        (Fract a, Fract b);
```

Performing Calculations with Fixed-Point Numbers

```
pascal Fract FracSqrt       (Fract x);
pascal Fract FracCos        (Fixed x);
pascal Fract FracSin        (Fixed x);
pascal Fixed FixATan2       (long x, long y);
```

Converting Among 32-Bit Numeric Types

```
pascal Fixed Long2Fix       (long x);
pascal long Fix2Long        (Fixed x);
pascal Fract Fix2Frac       (Fixed x);
pascal Fixed Frac2Fix       (Fract x);
```

Converting Between Fixed-Point and Floating-Point Values

```
pascal Extended Fix2X       (Fixed x);
pascal Fixed X2Fix          (Extended x);
pascal Extended Frac2X      (Fract x);
pascal Fract X2Frac         (Extended x);
```

Converting Between Fixed-Point and Integral Values

```
pascal Fixed FixRatio       (short numer, short denom);
pascal short FixRound       (Fixed x);
```

```
Multiplying 32-bit values
Pascal void LongMul          (long a, long b, Int64Bit *result);
```

Global Variables

`randSeed`	The seed to the pseudorandom number generator.

CHAPTER 4

Date, Time, and Measurement Utilities

Contents

This chapter describes a set of utility routines that you can use to operate on dates and times. You can use these routines to get and change information about the current date, time, geographic location, time zone, and units of measurement.

The routines described in this chapter return this information in a format that is best suited to the current script. As a result, you can facilitate localization of your application by using these date, time, and measurement utilities.

To understand the material in this chapter, you need to be familiar with the international resources, especially the numeric-format and long-date-format resources, and the Script Manager. These topics are described in *Inside Macintosh: Text*. In addition, the chapter "Text Utilities" in *Inside Macintosh: Text* describes how to convert date and time information into strings of text.

Many of the Date, Time, and Measurement Utilities were previously associated with other managers in the Macintosh system software, and several of these routines have been renamed. Table 4-4 on page 4-33 shows the original names and locations of the modified Date, Time, and Measurement Utilities routines.

The next section provides an introduction to the Date, Time, and Measurement Utilities.

About the Date, Time, and Measurement Utilities

You can use the Date, Time, and Measurement Utilities to manipulate the date-time information and geographic location data used by a Macintosh computer. A Macintosh computer contains a battery-operated **clock chip** that maintains

- the current date-time information
- the geographic location and related time-zone information

The date-time information is stored in a 4-byte value located on the clock chip.The geographic location and related time-zone information is stored in extended parameter RAM. For information on extended parameter RAM, see the chapter "Parameter RAM Utilities" in this book.

You can use the routines provided by the Date, Time, and Measurement Utilities to manipulate this information. Specifically, the Date, Time, and Measurement Utilities provide routines that you can use to

- get the current date and time
- set the current date and time, if necessary
- convert between internal date-time structures
- get and set the geographic location and time-zone information
- determine the current measurement system
- determine the number of elapsed microseconds since system startup

The following sections give an overview of these utilities.

Date and Time

A Macintosh computer contains a battery-operated clock chip that maintains the current date-time information. This date-time information is expressed, using 4 bytes, as the number of seconds elapsed since midnight, January 1, 1904. At system startup the date-time information is copied into low memory and is accessible through the system global variable `Time`. System software updates the value of the global variable `Time` each second. Doing this is faster than manipulating the clock chip directly.

The Date, Time, and Measurement Utilities provide four data structures that you can use to access date-time information. You can access date-time information through

- a **standard date-time value** that consists of a 32-bit long integer indicating the total number of seconds elapsed since midnight, January 1, 1904
- a **date-time record** that contains fields to indicate the year, month, day, hour, minute, second, and day of the week
- a **long date-time record** that extends the date-time record format by adding fields for era, day of the year, week of the year, and morning / evening designations (for example, A.M. and P.M.)
- a **long date-time value** that consists of a 64-bit integer, in SANE `comp` (computational) format, which also maintains the total number of seconds relative to midnight on January 1, 1904

To access date-time information as a date and time, you can use a date-time record or a long date-time record. A date-time record is defined by a data structure of type `DateTimeRec`

```
TYPE DateTimeRec =
RECORD
   year:       Integer;    {year, ranging from 1904 to 2040}
   month:      Integer;    {month, 1 = January and 12 = December}
   day:        Integer;    {day, from 1 to 31}
   hour:       Integer;    {hour, from 0 to 23}
   minute:     Integer;    {minute, from 0 to 59}
   second:     Integer;    {second, from 0 to 59}
   dayOfWeek:  Integer;    {day of the week, 1 = Sunday, }
                           { 7 = Saturday}
END;
```

The `year` field contains the year of the date, ranging from 1904 to 2040. The `month` field contains the month of the year, where a value of 1 equals January and 12 equals December. The `day` field contains the number of the day, ranging from day 1 to day 31. The `hour` field contains the hour, where the value of 0 equals midnight and 23 equals 11 P.M. The `minute` field contains the number of minutes, ranging from 0 to 59 minutes. The `second` field contains the number of seconds, ranging from 0 to 59 seconds. The `dayOfWeek` field specifies the name of the day; a value of 1 equals Sunday and a value of 7 equals Saturday. For additional information about the fields in a date-time record, see “The Date-Time Record” beginning on page 4-23.

Note
The date-time record can be used to hold date and time values only for a Gregorian calendar. The long date-time record, described next, can be used for a Gregorian calendar as well as other calendar systems. u

Because the values in a date-time record are simply a translation of the long integer containing the number of seconds since midnight, January 1, 1904, the data structure suffers the same limitation as the long integer representation: after the long integer has reached its maximum value of $FFFFFFFF, it resets to 0. Therefore, the date-time record can track dates and times only between midnight on January 1, 1904 and 6:28:15 A.M. on February 6, 2040.

For some applications, this range might be inadequate. For example, a hotel management application might need to let managers book reservations for customers who think ahead to 2050, or a history multimedia application might need to track dates in the first century B.C. If your application needs to track dates and times beyond the range supported by the date-time record, you must use a long date-time record. A long date-time record is defined by a data structure of type `LongDateRec`

```
TYPE LongDateRec =
RECORD
   CASE Integer OF
   0:
      (era:        Integer;    {era}
      year:        Integer;    {year, from 30081 B.C. to 29940 A.D}
      month:       Integer;    {month, 1 = January and }
                               { 12 = December}
      day:         Integer;    {day, from 1 to 31}
      hour:        Integer;    {hour, from 0 to 23}
      minute:      Integer;    {minute, from 0 to 59}
      second:      Integer;    {second, from 0 to 59}
      dayOfWeek:   Integer;    {day of the week, 1= Sunday, }
                               { 7 = Saturday}
      dayOfYear:   Integer;    {day of the year, 1 to 365}
      weekOfYear:  Integer;    {week of the year, 1 to 52}
      pm:          Integer;    {which half of day--0 for }
                               { morning, 1 for evening}
      res1:        Integer;    {reserved}
      res2:        Integer;    {reserved}
      res3:        Integer);   {reserved}
   1:
                               {index by LongDateField}
      (list:       ARRAY [0..13] OF Integer);
   2:
      (eraAlt:     Integer;    {era}
```

```
                              {date-time record}
      oldDate:    DateTimeRec);

END;
```

You can use a long date-time record for three purposes: to access a date and time, to specify which of the fields in a long date-time record to verify, and to convert a date and time represented by a date-time record into a date and time represented by a long date-time record.

IMPORTANT

The long date-time record covers a much longer time span (30,000 B.C. to 30,000 A.D.) than the date-time record. In addition, the long date-time record allows conversions to different calendar systems, such as a lunar calendar. s

A long date time-record includes all of the fields available in a date-time record in addition to fields that describe the era, day of the year, week of the year, and morning / evening designations (for example, A.M. and P.M.). The `era` field contains the era: a value of 0 represents A.D., and –1 represents B.C. The `dayOfYear` field contains a number that represents a day of a year. For example, the value 300 equals the 300th day of a year. The `weekOfYear` field contains a week number. The `pm` field contains the morning or evening half of the 24-hour day cycle, where a value of 0 represents the morning (for example, A.M.) and 1 represents the evening (for example, P.M.).

The `list` field contains an array of values that indicate which of the fields in a long date-time record need to be verified.

The `eraAlt` field, which indicates the era, and the `oldDate` field, which contains a date-time record, are used only for conversion from a date-time record to a long date-time record. For additional information about the fields in the long date-time record, see “The Long Date-Time Record” beginning on page 4-26.

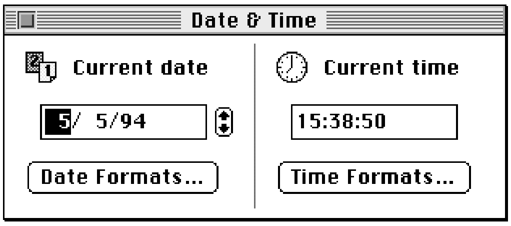
Note that if you specify, in either record, a value in the `month`, `day`, `hour`, `minute`, or `second` field that exceeds the maximum value allowed for that field (for example, a value larger than 23 for the `hour` field), the result is a wraparound to a future date and time when you modify the date-time format. Suppose you set the `year` field in a date-time record to a value greater than 2040, for example 2045. When you modify the date-time format, you get a value of 1909, because the value 2045 caused a wraparound to 1904 plus 5, the number of years over 2040. See “Calculating Dates” beginning on page 4-14 to see how you can use a wraparound to calculate and retrieve information about a specific date.

Note

To present a date and time value as a date and time text string, you need to use the Text Utilities routines. For a complete description of these routines, see *Inside Macintosh: Text*. u

A user can set the current date-time information by using the General Controls control panel, the Date & Time control panel, or the Alarm Clock. After the user sets the new date and time, this new date and time is written to the clock chip, and the global variable `Time` is updated to reflect the new date and time. Figure 4-1 illustrates how a user might change the date, using the Date & Time control panel.

Figure 4-1 The Date & Time control panel



Geographic Location and Time Zone

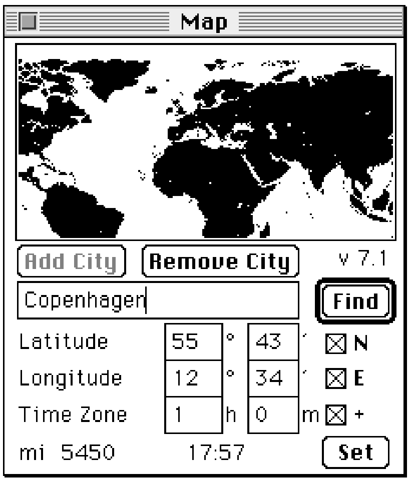
Geographic location and related time-zone information are stored in the Macintosh parameter RAM (extended parameter RAM). System software provides routines that allow you to read this information and, if necessary, make changes to it and then store the new settings in the parameter RAM (extended parameter RAM).

You can read and store values for

- latitude
- longitude
- daylight saving time (DST)
- Greenwich mean time (GMT)

The Map control panel allows the user to get geographic location and time-zone information. Figure 4-2 shows the Map control panel.

Figure 4-2 The Map control panel



The Map control panel specifies latitude and longitude, computation of Greenwich mean time for international time specification (shown as the Time Zone information), and computation of the distance and time difference between the current location (in this case, the location of the user's computer is Cupertino, California) and an arbitrary city (in this case, Copenhagen, Denmark).

See "Handling Geographic Location and Time-Zone Data" beginning on page 4-18, to see how you can use Date, Time, and Measurement Utilities routines to work with the geographic location and time-zone information.

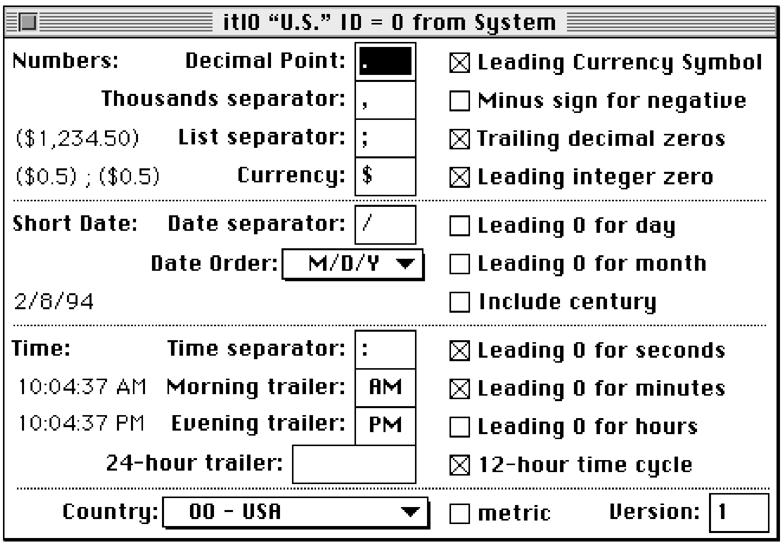
System of Measurement

The Date, Time, and Measurement Utilities provide a routine (the `IsMetric` function) that you can use to determine the type of measurement used by the current script system. The system software supports two types of measurement systems:

- the International System of Units (also called the metric system)—for example centimeters, kilometers, milligrams, degrees Celsius, and so on.
- the English system of measurement (also called the British or British imperial system)—for example, inches, miles, ounces, degrees Fahrenheit, and so on.

The measurement information is stored in the numeric-format resource (resource type `'itl0'`) of a script system. The `IsMetric` function determines whether the current script system uses the International System of Units or the English system of measurement by examining the '`itl0`' resource. Figure 4-3 depicts the window ResEdit displays for a numeric-format resource. Note that in the bottom of the figure the metric box is unchecked, indicating that the script system associated with this '`itl0`' resource uses the English system of measurement.

Figure 4-3 The numeric-format resource (resource type '`itl0`')



Time Measurement

The Date, Time, and Measurement Utilities provide a routine (the `Microseconds` procedure) that you can use to measure the number of microseconds that have elapsed since system startup. The `Microseconds` procedure is not effected by any user-specified changes to the date and time information, that is, a user can modify the current date-time information without effecting the value returned by the `Microseconds` procedure.

The number of microseconds elapsed is returned in a 64-bit unsigned integer, specified by the unsigned wide record. An unsigned wide record is defined by a data structure of type `UnsignedWide.`

```
TYPE UnsignedWide =
PACKED RECORD
   hi:   LongInt;          {high-order 32 bits}
   lo:   LongInt;          {low-order 32 bits}
END;
```

Using the Date, Time, and Measurement Utilities

This section describes how to

- get the current date and time
- set the current date and time
- calculate days and dates mathematically
- convert between date-time formats
- convert to different calendar systems
- read and store geographic location and time-zone data
- determine which measurement system to use
- determine the number of elapsed microseconds

Getting the Current Date and Time

The Date, Time, and Measurement Utilities provide

- a function—`ReadDateTime`—that system software uses at system startup time to copy the current date-time information from the clock chip into low memory. This low-memory copy of the current date-time is accessible through the global variable `Time`. You application should never need to use this function.
- two procedures —`GetDateTime` and `GetTime`—that allow you to access the current date-time information stored in the global variable `Time`.

You can access the date-time information through a date-time record, representing the date and time, or you can access the date-time information through a standard date-time value, a 32-bit integer representing the number of seconds since midnight, January 1, 1904.

To obtain the current date-time information, you can use the `GetDateTime` and `GetTime` procedures. The `GetDateTime` procedure requires that you pass it a standard date-time value as a parameter. Listing 4-1 shows how you can get the current date-time information, expressed as a number of seconds. The application-defined procedure `MyCurrentDateTimeInt` returns in the long integer the number of seconds elapsed since midnight, January 1, 1904.

Listing 4-1 Getting the current date and time with the `GetDateTime` procedure

```
PROCEDURE MyCurrentDateTimeInt (VAR myStandardDateTime: LongInt);
BEGIN
   GetDateTime(myStandardDateTime);
END;
```

The `GetTime` procedure requires that you pass it a date-time record as a parameter, and it fills in the fields of this record appropriately. Listing 4-2 shows how you can get the current date-time information, expressed as a date and time. The application-defined procedure `MyCurrentDateTimeRec` returns in the fields of the date-time record the current date and time.

Listing 4-2 Getting the current date and time with the `GetTime` procedure

```
PROCEDURE MyCurrentDateTimeRec (VAR myDateTime: DateTimeRec);
BEGIN
   GetTime(myDateTime);
END;
```

If you need to access the date-time information through a long date-time value or a long date-time record, see “Converting Date-Time Formats” beginning on page 4-12 for more information about converting date-time formats.

Setting the Current Date and Time

Your application can change the current date-time information stored in both the system global variable `Time` and in the clock chip by calling either the `SetDateTime` function or the `SetTime` procedure. The `SetDateTime` function requires a 32-bit integer as a parameter. The `SetTime` procedure requires a date-time record as a parameter.

Note
If you are using formats other than a date-time value or a date-time record to access date-time information, you must first convert these formats into a standard date-time value or a date-time record before you can write the new date-time information to the clock chip. See "Converting Date-Time Formats" beginning on page 4-12 for more information about converting date-time formats. u

Listing 4-3 shows an application-defined function that uses the `SetDateTime` function to change the current date and time to 5:50 A.M. on April 5, 1994.

Listing 4-3 Changing the current date and time with the `SetDateTime` function

```
FUNCTION MyChangeDateTimeInt: OSErr;
VAR
   myDateTimeInt: LongInt;
   myErr:         OSErr;
BEGIN
   myDateTimeInt := $A9C6AC88;
   myErr := SetDateTime(myDateTimeInt);
END;
```

Listing 4-4 shows an application-defined procedure that uses the `SetTime` function to change the current date and time to 5:50 A.M. on April 5, 1994.

Listing 4-4 Changing the current date and time with the `SetTime` function

```
PROCEDURE MyChangeDateTimeRec;
VAR
   myDateTimeRec: DateTimeRec;
   myErr:         OSErr;
BEGIN
   WITH myDateTimeRec DO
   BEGIN
      year := 1994;
      month := 4;
      day := 5;
      hour := 5;
      minute := 50;
      second := 0;
      dayOfWeek := 3;
   END;
   SetTime(myDateTimeRec);
END;
```

IMPORTANT

Users can change the current date and time stored in both the system global variable Time and in the clock chip by using the General Controls control panel, Date & Time control panel, or the Alarm Clock desk accessory. In general, your application should not directly change the current date-time information. If your application does need to modify the current date-time information, it should instruct the user how to change the date and time. s

Converting Date-Time Formats

The Date, Time, and Measurement Utilities provide four routines—the DateToSeconds, SecondsToDate, LongDateToSeconds, and LongSecondsToDate procedures—that you can use to convert date-time formats. You can convert a date and time to a number of seconds and a number of seconds to a date and time.

Note that when you call one of these routines, system software uses the DateToSeconds, SecondsToDate, LongDateToSeconds, and LongSecondsToDate procedures provided by the current script system.

Note

The routines that convert between time formats assume that each day contains 86,400 seconds. Occasionally (approximately once each two years) astronomers add a second to either June 31 or December 31 to compensate for imperfections in the earth's rotation. If you need to compute the exact number of seconds between two points in time, you might need to take these occasional additions into account. The routines that convert between formats are designed not to provide astronomical accuracy, but merely to convert data between one data structure and another. u

If you use a standard date-time value or a date-time record to access date-time information, you can use the SecondsToDate procedure to convert a number of seconds to a date and time, and the DateToSeconds procedure to convert a date and time to a number of seconds. Listing 4-6 shows an application-defined procedure, MyConvertSecondsAndDates, that uses the SecondsToDate and DateToSeconds procedures to manipulate the date-time information. After calling the GetDateTime procedure, MyConvertSecondsAndDates calls the SecondsToDate procedure to convert the number of seconds (returned by the GetDateTime procedure) to a date and time. The MyConvertSecondsAndDates procedure manipulates the year field in the date-time record and then calls DateToSeconds to convert the date and time back into a number of seconds. The SetDateTime procedure writes the new date-time information to the clock chip.

Listing 4-5 Manipulating date-time information

```
PROCEDURE MyConvertSecondsAndDates;
VAR
   myDateTimeRec:       DateRec;
   mySeconds:           DateTime;
   myErr:               OSErr;
BEGIN
   GetDateTime(mySeconds);
   SecondsToDate(mySeconds, myDateTimeRec);
   WITH myDateTimeRec DO
      year := year + 1;
   DateToSeconds (myDateTimeRec, mySeconds);
   myErr := SetDateTime(mySeconds);
END;
```

If you access date-time information through a long date-time value or a long date-time record, you can use the `LongSecondsToDate` procedure to convert a number of seconds to a date and time and use the `LongDateToSeconds` procedure to convert a date and time to a number of seconds.

If the type of data structure that you are using to access date-time information is insufficient, you can use a different date-time structure.

- To access a number of seconds through a long date-time value instead of a standard date-time value, set the `lHigh` field of a long date-time conversion record (described on page 4-25) to 0 and the `lLow` field to the total number of seconds since midnight, January 1, 1904. Then copy the value of the `c` field into a variable of type `LongDateTime`.
- To access a date and time through a long date-time record instead of a date-time record, set the `oldDate` field of the `LongDateRec` to the date-time record, and set the `eraAlt` field to 0, indicating that the date you have specified is A.D.
- To access a number of seconds through a standard date-time value instead of a long date-time value, truncate the long date-time value to just the low-order 32 bits. The year of the date being converted must fall within 1904 to 2040 of the Gregorian calendar.

  This type of conversion is important when you work with a script system that uses a calendar system other than the Gregorian. Because you cannot write a long date-time value to the clock chip, you must first convert the long date-time value (if possible) to a standard date-time value. See "Working With Different Calendar Systems" beginning on page 4-16 for more information about calendar systems.
- To access a date and time through a date-time record instead of a long date-time record, truncate the long date-time record so just the `year` through `dayOfWeek` fields are left. Once again, the year of the date being converted must fall within 1904 to 2040 of the Gregorian calendar.

- To access date-time information through a long date-time value instead of a date-time record, use the `DateToSeconds` procedure to convert the date and time to a number of seconds. Then set the `lHigh` field of a long date-time conversion record (described on page 4-25) to 0 and the `lLow` field to the total number of seconds since midnight, January 1, 1904.
- To access date-time information through a long date-time record (described on page 4-26) instead of a standard date-time value, use the `SecondsToDate` procedure to translate the number of seconds to a date and time. Then set the `oldDate` field of the long date-time record to the date-time record, and set the `eraAlt` field to 0.
- To access date-time information through a date-time value instead of long date-time record, use the `LongDateToSeconds` procedure to translate the date and time to a number of seconds. Then truncate the long date-time value (returned by the `LongDateToSeconds` procedure) to just the low-order 32 bits. The year of the date being converted must fall within 1904 to 2040 in the Gregorian calendar.

The Gregorian calendar is the default for converting to and from the long date-time forms. The current range allowed in conversion is roughly 30,000 B.C. to 30,000 A.D.

To present a date and time value as a date and time text string, you need to use Text Utilities routines, such as the `DateString`, `TimeString`, `StringToDate`, `StringToTime`, `LongDateString`, and `LongTimeString` routines. (Note that the date-string conversion routines do not append strings for A.D. or B.C.) For a complete description of these routines, see *Inside Macintosh: Text.*

Calculating Dates

In the date-time record and long date-time record, any value in the `month`, `day`, `hour`, `minute`, or `second` field that exceeds the maximum value allowed for that field, will cause a wraparound to a future date and time when you modify the date-time format.

- In the `month` field, values greater than 12 cause a wraparound to a future year and month.
- In the `day` field, values greater than the number of days in a given month cause a wraparound to a future month and day.
- In the `hour` field, values greater than 23 cause a wraparound to a future day and hour.
- In the `minute` field, values greater than 59 cause a wraparound to a future hour and minute.
- In the `seconds` field, values greater than 59 cause a wraparound to a future minute and seconds.

You can use these wraparound facts to calculate and retrieve information about a specific date. For example, you can use a date-time record and the `DateToSeconds` and `SecondsToDate` procedures to calculate the 300th day of 1994. Set the `month` field of the date-time record to 1 and the `year` field to 1994. To find the 300th day of 1994, set the `day` field of the date-time record to 300. Initialize the rest of the fields in the record to values that do not exceed the maximum value allowed for that field. (Refer to the description of the date-time record on page 4-23 for a complete list of possible values).

To force a wrap-around, first convert the date and time (in this example, January 1, 1994) to the number of seconds elapsed since midnight, January 1, 1904 (by calling the `DateToSeconds` procedure). Once you have converted the date and time to a number of seconds, you convert the number of seconds back to a date and time (by calling the `SecondsToDate` procedure). The fields in the date-time record now contain the values that represent the 300th day of 1994. Listing 4-6 shows an application-defined procedure that calculates the 300th day of the Gregorian calendar year using a date-time record.

Listing 4-6 Calculating the 300th day of the year

```
PROCEDURE MyCalculate300Day;
VAR
   myDateTimeRec:    DateTimeRec;
   mySeconds:        LongInt;
BEGIN
   WITH myDateTimeRec DO
   BEGIN
      year := 1994;
      month := 1;
      day := 300;
      hour := 0;
      minute := 0;
      second := 0;
      dayOfWeek := 1;
   END;
   DateToSeconds (myDateTimeRec, mySeconds);
   SecondsToDate (mySeconds, myDateTimeRec);
END;
```

The `DateToSeconds` procedure converts the date and time to the number of seconds elapsed since midnight, January 1, 1904, and the `SecondsToDate` procedure converts the number of seconds back to a date and time. After the conversions, the values in the `year`, `month`, `day`, and `dayOfWeek` fields of the `myDateTimeRec` record represent the year, month, day of the month, and day of the week for the 300th day of 1994. If the values in the `hour`, `minute`, and `second` fields do not exceed the maximum value allowed for each field, the values remain the same after the conversions (in this example, the time is exactly 12:00 A.M.).

Similarly, you can use a long date-time record and the `LongDateToSeconds` and `LongSecondsToDate` procedures to compute the day of the week corresponding to a given date. Listing 4-7 shows an application-defined procedure that computes and retrieves the name of the day for July 4, 1776. Note that because the year is prior to 1904, it is necessary to use a long date-time record.

Listing 4-7 Computing the day of the week

```
PROCEDURE DoDayCalc;
VAR
   myLongDateRec: LongDateRec;
   myLongSeconds: LongDateTime;
   myDayOfWeek:   Integer;
BEGIN
   WITH myLongDateRec DO
   BEGIN
      era := 0;         /*initialize era field*/
      year := 1776;
      month := 7;
      day := 4;
      hour := 0;        /*initialize hour field*/
      minute := 0;      /*initialize minute field*/
      second := 0;      /*initialize second field*/
      dayOfWeek := 1;   /*initialize dayOfWeek field*/
      dayOfYear := 1;   /*initialize dayOfYear field*/
      weekOfYear := 1;  /*initialize weekOfYear field*/
      pm := 1;          /*initialize pm field*/
   END;
   LongDateToSeconds (myLongDateRec, myLongSeconds);
   LongSecondsToDate (myLongSeconds, myLongDateRec);
   myDayOfWeek := myLongDateRec.dayOfWeek;
END;
```

The `LongDateToSeconds` procedure converts the date and time to the number of seconds, and the `LongSecondsToDate` procedure converts the number of seconds back to a date and time. After the conversions, the value in the `dayOfWeek` field of the `myLongDateRec` record represent the day of the week corresponding to July 4, 1776. If the values in the `hour`, `minute`, and `second` fields do not exceed the maximum value allowed for each field, the values remain the same after the conversions (in this example, the time is exactly 12:00 A.M.). The values in the `dayOfYear`, `weekOfYear`, and `pm` fields correspond to the date July 4, 1776 and the time 12:00 A.M.

Working With Different Calendar Systems

The additional fields and wider ranges allowed by the long date-time record can help you to do calculations and conversions for different calendar systems. For example, the date January 1, 1993 in the Gregorian calendar year converts to 7 Rajab 1413 in the Arabic Civil Lunar Calendar (CLC) and 4 Tevet 5753 in the Jewish calendar; the years 1413 and 5753 are outside of the `year` field's range in the date-time record.

Note

Depending on the country, the change from the Julian calendar to the Gregorian calendar occurred in different years. In western European countries, the change occurred in 1582; in Russia, the calendar changed in 1918. In these countries, dates before the calendar change should use the Julian calendar for conversion. (The Julian calendar differs from the Gregorian calendar by three days every four centuries.) u

In addition, the beginning of the year for one calendar system falls on different dates in other calendar systems. Table 4-1 shows the equivalent dates for the first day of the calendar year in the Gregorian, Arabic CLC, and Jewish calendars.

Table 4-1 Equivalent dates in the Gregorian, Arabic CLC, and Jewish calendars

Gregorian calendar	Arabic CLC	Jewish calendar
January 1, 1993	7 Rajab 1413	4 Tevet 5753
June 20, 1993	1 Muharram 1414	1 Tammuz 5753
September 16, 1993	29 Rabi I 1414	1 Tishri 5754

Converting from one calendar system to another produces different values in the `dayOfYear` and `weekOfYear` fields of a long date-time record. For example, assuming all the data for the date 1 Muharram 1414 is correctly put into a long date-time record, the `dayOfYear` field value is 1, and the `weekOfYear` value is also 1. Converting this date to the Gregorian calendar results in June 20, 1993. The `dayOfYear` field value is then 171, and the `weekOfYear` value is 26. Table 4-2 shows these values.

Table 4-2 Values for the `dayOfYear` and `weekOfYear` fields for the date 1 Muharram 1414 and equivalent values in the Gregorian calendar

LongDateRec field	Arabic CLC	Gregorian calendar
`dayOfYear`	1	171
`weekOfYear`	1	26

Note

Language-specific information, such as the name of the day, name of the month, and so on, are stored in the international resources. The international resources are provided by a script system, and the information in these resources varies according to the language associated with the script system. u

Table 4-3 shows how some of the fields in the long date-time record are set to show the first day of the year 1414 in the Arabic CLC and the equivalent dates in the Gregorian and Jewish calendars.

Table 4-3 Comparison of settings in fields of the long date-time record for Arabic CLC, Gregorian, and Jewish calendars

Field of a long date-time record	Arabic CLC calendar	Gregorian calendar	Jewish calendar
era	0	0	0
year	1413	1993	5753
month	1	6	
day	1	21	
...			
dayOfWeek	4	2	3
dayOfYear	1	172	
weekOfYear	1	26	

Note

The Arabic script system supports two lunar calendars: the astronomical lunar calendar (ALC) and the civil lunar calendar (CLC). The Macintosh user may choose either of the Arabic calendars or the Gregorian calendar by clicking buttons in the Arabic Calendar control panel.

The Hebrew script system supports the Jewish calendar besides the Gregorian calendar.

For more information on the different calendar systems supported by localized versions of the Macintosh system software, see *Guide to Macintosh Software Localization*. u

For calendars that have more than seven day names and 12 month names (for example, the Jewish calendar sometimes has 13 months), you use the 'itl1' resource, defined by the Itl1ExtRec data type. To get more information on the format of the 'itl1' resource, see the appendix “International Resources” in *Inside Macintosh: Text*.

Handling Geographic Location and Time-Zone Data

Geographic locations and time zones can affect date and time information. For example, time-zone information can be used to derive the Greenwich mean time (GMT) at which a document or mail message was created. With this information, when the document is received by an application or user in a different time zone, the creation date and time are correct. Otherwise, documents can appear to be created after they are read (for example, a user creates a message in Tokyo on Tuesday and sends it to San Francisco, where it is received and read on Monday). Geographic location information can also be used by applications that require it.

The geographic location and time-zone information for a particular Macintosh computers are stored in parameter RAM. You can work with this information through the ReadLocation and WriteLocation procedures. These procedures use the geographic location record (of date type `MachineLocation`) to help you read and store latitude, longitude, daylight saving time (DST), and GMT values.

```
TYPE MachineLocation =              {geographic location record}
RECORD
   latitude:         Fract;         {latitude}
   longitude:        Fract;         {longitude}
   CASE Integer OF
   0:
      (dlsDelta:     SignedByte);   {daylight saving time}
   1:
      (gmtDelta:     LongInt);      {Greenwich mean time}
END;
```

The daylight savings time value is a signed byte value that you can use to specify the offset for the `hour` field—whether to add 1 hour, subtract 1 hour, or make no change at all.

The Greenwich mean time value is in seconds east of GMT. For example, San Francisco is at –28,800 seconds (8 hours * 3,600 seconds per hour) east of GMT.

If the geographic location record has never been set, all fields contain 0.

Generally, latitude and longitude are measured in degrees. These values also can be thought of as fractions of a great circle.

Latitude and longitude information is stored in the geographic location record as values of type `Fract`. These values give accuracy to within 1 foot, which should be sufficient for most purposes. For example, the `Fract` value 1.0 equals 90 degrees; –1.0 equals –90 degrees; and –2.0 equals –180 degrees.

To store latitude and longitude values, you need to convert them first to the `Fixed` data type, then to the `Fract` data type. You can use the Operating System Utilities routines `Long2Fix` and `Fix2Fract` to accomplish this task. Listing 4-8 is an application-defined procedure that converts San Francisco's latitude and longitude to `Fract` values, then writes the `Fract` values to parameter RAM using the `WriteLocation` procedure.

Listing 4-8 Converting latitude and longitude to `Fract` values

```
PROCEDURE MyConvertLatLong;
VAR
   myLatitude, myLongitude:         LongInt;
   fixedLatitude, fixedLongitude:   Fixed;
   latFract, longFract:             Fract;
   myLocation:                      MachineLocation;
BEGIN
   myLatitude:= 37.48;                    {degrees latitude}
   myLongitude:= 122.24;                  {degrees longitude}
```

```
   {convert from long to fixed data type}
   fixedLatitude:= Long2Fix(myLatitude);
   fixedLongitude:= Long2Fix(myLongitude);

   {convert from fixed to Fract data type}
   latFract:= Fix2Frac(fixedLatitude);
   longFract:= Fix2Frac(fixedLongitude);

   {write latitude and logitude to myLocation}
   myLocation.latitude:= latFract;
   myLocation.longitude:= longFract;

   {write latitude and longitude to parameter RAM}
   WriteLocation(myLocation);

END;
```

To read the latitude and longitude values from parameter RAM, you use the `ReadLocation` procedure. To convert these values to a degrees format, you need to convert the `Fract` values first to the `Fixed` data type, then to the `LongInt` data type. You can use the Mathematical and Logical Utilities routines `Fract2Fix` and `Fix2Long` to accomplish this task. (For more information on the `Fract` data type and the conversion routines `Long2Fix`, `Fix2Fract`, `Fract2Fix`, and `Fix2Long`, see the chapter "Mathematical and Logical Utilities" in this book.)

The `gmtDelta` field of the geographic location record is a 3-byte value contained in a long word, so you must take care to get and set it properly. Listing 4-9 shows an application-defined function for obtaining the value of `gmtDelta`.

Listing 4-9 Getting `gmtDelta`

```
FUNCTION MyGetGmtDelta (myLocation: MachineLocation): LongInt;
VAR
   internalGmtDelta: LongInt;
BEGIN
   WITH myLocation DO
   BEGIN
      internalGmtDelta := BitAnd(gmtDelta, $00FFFFFF);
      IF BitTst(internalGmtDelta, 23) THEN
                                       {test sign extend bit}
         internalGmtDelta := BitOr(internalGmtDelta, $FF000000);
      MyGetGmtDelta := internalGmtDelta;
   END;
END;
```

When writing `gmtDelta`, you should preserve the value of `dlsDelta`. Listing 4-10 shows an application-defined procedure that writes `gmtDelta` while preserving the value of `dlsDelta`.

Listing 4-10 Setting `gmtDelta`

```
PROCEDURE MySetGmtDelta (VAR myLocation: Location;
                          myGmtDelta: LongInt);
VAR
   tempSignedByte: SignedByte;
BEGIN
   WITH myLocation DO
   BEGIN
      tempSignedByte := dlsDelta;         {preserve dlsDelta}
      gmtDelta := myGmtDelta;             {write gmtDelta}
      dlsDelta := tempSignedByte;         {restore dlsDelta}
   END;
END;
```

Note that you should mask off the top byte of the long word containing `gmtDelta` because it is reserved.

Determining the Measurement System

To implement measuring devices in applications, such as rulers in a word processor or in drawing applications, you need to determine which measurement system your application should use. You can use the `IsMetric` function to determine if the measurement system needs to be the metric system or the English system. The `IsMetric` function reads the numeric-format resource (resource type `'itl0'`) of the current script system to determine whether the user is using the metric system or the English system.

Listing 4-11 shows an application-defined procedure that uses the result of the `IsMetric` function to determine which application-defined ruler setup to use for a document window.

Listing 4-11 Getting the current units of measurement

```
PROCEDURE DoRuler (window: WindowPtr);
VAR
   myMeasure: BOOLEAN;            {response returned by IsMetric}
BEGIN
   myMeasure := IsMetric;
   IF myMeasure = TRUE THEN       {metric system is default}
```

```
      DoMetricRulerSetup         {set up metric system ruler}
   ELSE
      DoEnglishRulerSetup;       {set up English system ruler}
END;
```

If you want to use a measurement system different from that of the current script, you need to override the value of the `metricSys` field in the current numeric-format resource (resource type `'itl0'`). You can do this by using your own version of the numeric-format resource instead of the current script system's default international resources. See the chapter "Script Manager" in *Inside Macintosh: Text* for information on how to replace a script system's default international resources.

Determining the Number of Elapsed Microseconds

Your application can use the `Microseconds` procedure to obtain the number of elapsed microseconds since system startup time. You can use the value returned by the `Microseconds` procedure to time an event. For example, Listing 4-11 shows an application-defined function `MyEventTimer` that computes and returns the time it takes to execute an application-defined procedure `DoMyEvent`. The application-defined function `MyCalulateElapsedTime` function uses the returned value of the `Microseconds` procedure to compute the time it takes to execute the `DoMyEvent` procedure.

Listing 4-12 Timing an event using the `Microseconds` procedure

```
FUNCTION MyEventTimer: UnsignedWide;
VAR
   myStartTime:UnsignedWide;
   myEndTime: UnsignedWide;
BEGIN
   Microseconds(&myStartTime);
   DoMyEvent;
   Microseconds(&myEndTime);
   MyEventTimer := MyComputeElapsedTime(&myStartTime, &myEndTime);
END;
```

Because there is no compiler support for 64-bit integers, you must write an application-defined routine that calculates the elapsed time; you cannot obtain the elapsed time by subtracting the value in the `myStartTime` parameter from the value in the `myEndTime` parameter.

Date, Time, and Measurement Utilities Reference

This section describes the data structures and routines that are specific to the Date, Time, and Measurement Utilities. The section "Data Structures" shows the Pascal data structures for the date-time record, long date-time record, standard date-time value, long date-time value, and more. The section "Routines" describes the routines you can use to read, write, and manipulate date-time information.

Data Structures

This section describes the data structures that you use to exchange information with the Date, Time, and Measurement Utilities.

The Date-Time Record

The date-time record describes the date-time information as a date and time. The Date, Time, and Measurement Utilities use a date-time record to read and write date-time information to and from the clock chip. The `DateTimeRec` data type defines the date-time record.

Note

The date-time record can be used to hold date and time values only for a Gregorian calendar. The long date-time record (described on page 4-26) can be used for a Gregorian calendar as well as other calendar systems. u

```
TYPE DateTimeRec =
RECORD
   year:       Integer;    {year, ranging from 1904 to 2040}
   month:      Integer;    {month, 1= January and 12 = December}
   day:        Integer;    {day of the month, from 1 to 31}
   hour:       Integer;    {hour, from 0 to 23}
   minute:     Integer;    {minute, from 0 to 59}
   second:     Integer;    {second, from 0 to 59}
   dayOfWeek:  Integer;    {day of the week, 1 = Sunday, }
                           { 7 = Saturday}
END;
```

Field descriptions

`year` The year, ranging from 1904 to 2040. Note that to indicate the year 1984, this field would store the integer 1984, not just 84. This field accepts input of 0 or negative values, but these values produce unpredictable results in the `year`, `month`, and `day` fields when you use the `SecondsToDate` and `DateToSeconds` procedures. In addition, using `SecondsToDate` and `DateToSeconds` with year values greater than 2040 causes a wraparound to 1904 plus the number of years over 2040. For example, setting the year to 2045 returns a value of 1909, and the other fields in this record return unpredictable results.

`month` The month of the year, where 1 represents January, and 12 represents December. Values greater than 12 cause a wraparound to a future year and month. This field accepts input of 0 or negative values, but these values produce unpredictable results in the `year`, `month`, and `day` fields when you use the `SecondsToDate` and `DateToSeconds` procedures.

`day` The day of the month, ranging from 1 to 31. Values greater than the number of days in a given month cause a wraparound to a future month and day. This feature is useful for working with leap years. For example, the 366th day of January in 1992 (1992 was a leap year) evaluates as December 31, 1992, and the 367th day of that year evaluates as January 1, 1993.

This field accepts 0 or negative values, but when you use the `SecondsToDate` and `DateToSeconds` procedures, a value of 0 in this field returns the last day of the previous month. For example, a month value of 2 and a day value of 0 return 1 and 31, respectively.

Using `SecondsToDate` and `DateToSeconds` with a negative number in this field subtracts that number of days from the last day in the previous month. For example, a month value of 5 and a day value of –1 return 4 for the month and 29 for the day; a month value of 2 and a day value of –15 return 1 and 16, respectively.

`hour` The hour of the day, ranging from 0 to 23, where 0 represents midnight and 23 represents 11:00 P.M. Values greater than 23 cause a wraparound to a future day and hour. This field accepts input of negative values, but these values produce unpredictable results in the `month`, `day`, `hour`, and `minute` fields you use the `SecondsToDate` and `DateToSeconds` procedures.

`minute` The minute of the hour, ranging from 0 to 59. Values greater than 59 cause a wraparound to a future hour and minute. When you use the `SecondsToDate` and `DateToSeconds` procedures, a negative value in this field has the effect of subtracting that number from the beginning of the given hour. For example, an `hour` value of 1 and a `minute` value of –10 return 0 hours and 50 minutes. However, if the negative value causes the `hour` value to be less than 0, for example `hour` = 0, `minute` = –61, unpredictable results occur.

`second` The second of the minute, ranging from 0 to 59. Values greater than 59 cause a wraparound to a future minute and second. When you use the `SecondsToDate` and `DateToSeconds` procedures, a negative value in this field has the effect of subtracting that number from the beginning of the given minute. For example, a `minute` value of 1 and a `second` value of –10 returns 0 minutes and 50 seconds. However, if the negative value causes the `hour` value to be less than 0, for example hour = 0, minute = 0, and second = –61, unpredictable results occur.

dayOfWeek — The day of the week, where 1 indicates Sunday and 7 indicates Saturday. This field accepts 0, negative values, or values greater than 7. When you use the SecondsToDate and DateToSeconds procedures, you get correct values because this field is automatically calculated from the values in the year, month, and day fields.

Long Date-Time Value and Long Date-Time Conversion Record

The long date-time value specifies the date and time as seconds relative to midnight, January 1, 1904. But where the standard date-time value is an unsigned, 32-bit long integer, the long date-time value is a signed, 64-bit integer in SANE comp format. This format lets you use dates and times with a much longer span—roughly 500 billion years. You can use this value to represent dates and times prior to midnight, January 1, 1904. The LongDateTime data type defines the long date-time value.

```
TYPE LongDateTime = comp;
```

When storing a long date-time value in files, you can use a 5-byte or 6-byte format for a range of roughly 35,000 years. You should sign extend this value to restore it to a comp format.

The Date, Time, and Measurement Utilities provide the LongDateCvt record to help in setting up LongDateTime values.

```
TYPE LongDateCvt =
RECORD
   CASE Integer OF
   0:
      (c:      comp);      {number of seconds relative to }
                           { midnight, January 1, 1904}
   1:
      (lHigh:  LongInt;    {high long integer}
       lLow:   LongInt);   {low long integer}
END;
```

Field descriptions

c — The date and time, specified in seconds relative to midnight, January 1, 1904, as a signed, 64-bit integer in SANE comp format. The high-order bit of this field represents the sign of the 64-bit integer. Negative values allow you to indicate dates and times prior to midnight, January 1, 1904.

lHigh — The high-order 32 bits when converting from a standard date-time value. Set this field to 0.

`lLow` The low-order 32 bits when converting from a standard date-time value. Set this field to the standard date-time value representing the total number of seconds since midnight, January 1, 1904.

The Long Date-Time Record

In addition to the date-time record, system software provides the long date-time record, which extends the date-time record format by adding several more fields. This format lets you use dates and times with a much longer span (30,000 B.C. to 30,000 A.D.). In addition, the long date-time record allows conversions to different calendar systems, such as a lunar calendar.

The `LongDateRec` data type defines the format of the long date-time record.

```
TYPE LongDateRec =
RECORD
   CASE Integer OF
   0:
      (era:          Integer;    {era}
       year:         Integer;    {year, from 30,081 B.C. }
                                 { to 29,940 A.D.}
       month:        Integer;    {month}
       day:          Integer;    {day of the month}
       hour:         Integer;    {hour, from 0 to 23}
       minute:       Integer;    {minute, from 0 to 59}
       second:       Integer;    {second, from 0 to 59}
       dayOfWeek:    Integer;    {day of the week}
       dayOfYear:    Integer;    {day of the year}
       weekOfYear:   Integer;    {week of the year}
       pm:           Integer;    {morning/evening}
       res1:         Integer;    {reserved}
       res2:         Integer;    {reserved}
       res3:         Integer);   {reserved}
   1:
                                 {index by LongDateField}
      (list:         ARRAY[0..13] OF Integer);
   2:
      (eraAlt:       Integer;       {era}
      oldDate:       DateTimeRec);  {date-time record}
END;
```

Field descriptions

`era` The era, where 0 represents A.D., and –1 represents B.C.

`year` The year, ranging from 30,081 B.C. to 29,940 A.D. Values outside this range produce unpredictable results in all fields of the record. Note that to indicate the year 1984, this field would store the integer 1984, not just 84. This field accepts input of 0 or negative values, but these values return the positive result of the value plus one for the year. For example, a `year` value of 0 returns 1, and a `year` value of –1993 returns 1994. Other fields are unaffected.

`month` — The month of the year, where 1 represents January, and 12 represents December. When you use the `LongSecondsToDate` and `LongDateToSeconds` procedures, `month` values greater than 12 cause a wraparound to a future year and month. A value of 0 in this field returns the 12th month of the previous year. For example, a `month` value of 0 and a `year` value of 1993 return 12 and 1992, respectively. A negative value in this field has the effect of subtracting that number from the first month of the given year. For example, a `month` value of –2 and a `year` value of 1993 return 10 and 1992, respectively.

`day` — The day of the month, ranging from 1 to 31. When using the `LongSecondsToDate` and `LongDateToSeconds` procedures, `day` values greater than the number of days in a given month cause a wraparound to a future month and day. This feature is useful for working with leap years. For example, the 366th day of January in 1992 (1992 was a leap year) evaluates as December 31, 1992, and the 367th day of that year evaluates as January 1, 1993. A value of 0 in this field produces unpredictable results in the `month` and `day` fields. A negative value in this field has the effect of subtracting that number from the first day of the given month. For example, a `day` value of –10 and a `month` value of 10 return 9 and 20, respectively.

`hour` — The hour of the day, ranging from 0 to 23, where 0 represents midnight and 23 represents 11:00 P.M. When you use the `LongSecondsToDate` and `LongDateToSeconds` procedures, `hour` values greater than 23 cause a wraparound to a future day and hour. A negative value in this field produces unpredictable results. Note that this field is always maintained in 24-hour time. The `pm` field, if used, is redundant.

`minute` — The minute of the hour, ranging from 0 to 59. When you use the `LongSecondsToDate` and `LongDateToSeconds` procedures, `minute` values greater than 59 cause a wraparound to a future hour and minute. A negative value in this field has the effect of subtracting that number from the first minute of the given hour. For example, an `hour` value of 10 and a `minute` value of –10 return 9 and 50, respectively. However, if the negative value causes the `hour` value to become less than 0, for example `hour` = 0 and `minute` = –61, unpredictable results occur.

`second` — The second of the minute, ranging from 0 to 59. When you use the `LongSecondsToDate` and `LongDateToSeconds` procedures, `second` values greater than 59 cause a wraparound to a future minute and second. A negative value in this field has the effect of subtracting that number from the first second of the given minute. For example, an `minute` value of 10 and a `second` value of –10 return 9 and 50, respectively. However, if the negative value causes the `hour` value to become less than 0, for example `hour` = 0, `minute` = 0, and `second` = –61, unpredictable results occur.

`dayOfWeek` The day number of the week, where 1 indicates Sunday and 7 indicates Saturday. This field accepts 0, negative values, or values greater than 7. When you use the `LongSecondsToDate` and `LongDateToSeconds` procedures, you get correct values because this field is automatically calculated from the values in the `year`, `month`, and `day` fields. For calendars that have more than 7 day names and 12 month names (for example, the Jewish calendar sometimes has 13 months), you use the `'itl1'` resource, defined by the `Itl1ExtRec` data type. To get more information on the format of the `'itl1'` resource, see the appendix "International Resources" in *Inside Macintosh: Text*.

`dayOfYear` The day number of the year, ranging from 1 to 366. Values greater than the number of days in a given year cause a wraparound to a future year and day. This feature is useful for working with leap years. For example, in a Gregorian calendar the 366th day of January in 1992 (1992 was a leap year) evaluates as December 31, 1992, and the 367th day of that year evaluates as January 1, 1993.

`weekOfYear` The week number of the year, ranging from 1 to 52. Note that out-of-range values (such as 0, negative numbers, or numbers greater than 52) can be set for this field. However, you can use the `LongSecondsToDate` procedure to convert these out-of-range values to appropriate values.

`pm` The morning or evening half of the 24-hour day cycle, where 0 represents the morning (for example, A.M.), and 1 represents the evening (for example, P.M.). Note that out-of-range values can be set for this field. However, you can use the `LongSecondsToDate` procedure to convert these out-of-range values to appropriate values.

`res1` Reserved. Set this field to 0.

`res2` Reserved. Set this field to 0.

`res3` Reserved. Set this field to 0.

`list` An array of `LongDateField` values. The `field` parameter of the `ToggleDate` function uses the enumerated data type `LongDateField` to indicate the `LongDateRec` fields that the `ValidDate` function should check. The following values are available:

```
TYPE LongDateField =
     (eraField, yearField, monthField, dayField,
      hourField, minuteField, secondField,
      dayOfWeekField, dayOfYearField,
      weekOfYearField, pmField, res1Field,
      res2Field, res3Field);
```

`eraAlt` The era, where 0 represents A.D., and –1 represents B.C. Use this field and the `oldDate` field to convert from a date-time record.

`oldDate` The date-time record to convert. Use this field and the `eraAlt` field to convert from a date-time record.

The Geographic Location Record

The geographic location and time-zone information of a Macintosh computer are stored in extended parameter RAM. The `MachineLocation` data type defines the format for the geographic location record.

```
TYPE MachineLocation =  {geographic location record}
RECORD
   latitude:            Fract;         {latitude}
   longitude:           Fract;         {longitude}
   CASE Integer OF
   0:
      (dlsDelta:        SignedByte);   {daylight saving time}
   1:
      (gmtDelta:        LongInt);      {Greenwich mean time}
END;
```

Field descriptions

`latitude`
: The location's latitude, in fractions of a great circle. For example, Copenhagen, Denmark is at 55.43 degrees north latitude. When writing the latitude to extended parameter RAM with the `WriteLocation` procedure, you must convert this value to a `Fract` data type. (For example, a `Fract` value of 1.0 equals 90 degrees; –1.0 equals –90 degrees; and –2.0 equals –180 degrees.) For an example that shows this conversion process, see Listing 4-8 on page 4-19. For more information on the `Fract` data type, see the chapter "Mathematical and Logical Utilities" in this book.

`longitude`
: The location's longitude, in fractions of a great circle. For example, Copenhagen, Denmark is at 12.34 degrees east longitude. When writing the longitude to extended parameter RAM with the `WriteLocation` procedure, you must convert this value to a `Fract` data type. (For example, a `Fract` value of 1.0 equals 90 degrees; –1.0 equals –90 degrees; and –2.0 equals –180 degrees.) For an example that shows this conversion process, see Listing 4-8 on page 4-19. For more information on the `Fract` data type, see the chapter "Mathematical and Logical Utilities" in this book.

`dlsDelta`
: A signed byte value representing the hour offset for daylight saving time. This field is a 1-byte value contained in a long word. It should be preserved when writing `gmtDelta`. See Listing 4-10 on page 4-21 for an example that writes `gmtDelta` while preserving `dlsDelta`.

`gmtDelta`
: The Greenwich mean time (GMT). For example, Copenhagen, Denmark is at 1 hour west of GMT. This field is a 3-byte value contained in a long word. In addition, the top byte of this field should be masked off when writing because it is reserved. See Listing 4-9 on page 4-20 and Listing 4-10 on page 4-21 for code examples that get and set `gmtDelta` properly.

The `ReadLocation` and `WriteLocation` procedures use the geographic location record to read and store the geographic location and time zone information in extended parameter RAM. If the geographic location record has never been set, all fields contain 0.

The Toggle Parameter Block

The `ToggleDate` function exchanges information with your application using the toggle parameter block, defined by the `TogglePB` data type.

```
TYPE TogglePB =
RECORD
   togFlags:      LongInt;    {flags}
   amChars:       ResType;    {A.M. characters from 'itl0' }
                              { resource, but made uppercase}
   pmChars:       ResType;    {P.M. characters from 'itl0' }
                              { resource, but made uppercase}
   reserved:      ARRAY[0..3] OF LongInt; {reserved}
END;
```

Field descriptions

`togFlags` The high-order word of this field contains flags that specify special conditions for the `ToggleDate` function:

```
genCdevRangeBit   = 27;    {restrict date/time to }
                           { range used by }
                           { General Controls }
                           { control panel}
togDelta12HourBit = 28;    {if modifying hour }
                           { up/down, restrict to }
                           { 12-hour range}
togCharZCycleBit  = 29;    {modifier for }
                           { togChar12HourBit to }
                           { accept hours }
                           { 0…11 only}
togChar12HourBit     = 30; {if modifying hour by }
                           { char, accept hours }
                           { 1…12 only}
smallDateBit         = 31; {restrict valid }
                           { date/time to }
                           { range of Time global}
```

`genCdevRangeBit`
If this bit is set in addition to `smallDateBit`, then the date range is restricted to that used by the General Controls control panel—

January 1, 1920 to December 31, 2019 in the Gregorian calendar (the routine works correctly for other calendars as well). For dates outside this range but within the range specified by the system global variable `Time`—January 1, 1904 to February 6, 2040 in the Gregorian calendar—`ToggleDate` adds or subtracts 100 years to bring the dates into the range of the General Controls control panel if these bits are set. The `ToggleDate` function returns an error if the `smallDateBit` is set and the date is outside the range specified by the system global variable `Time`. This bit works with system software version 6.0.4 and later.

`togDelta12HourBit`
If this bit is set, modifying the hour up or down is limited to a 12-hour range. For example, increasing by one from 11 produces 0, increasing by one from 23 produces 12, and so on. This bit works with system software version 6.0.4 and later.

`togCharZCycleBit`
If this bit is set, the input character is treated as if it modifies an hour whose value is in the range 0–11. If this bit is not set, the input character is treated as if it modifies an hour whose value is in the range 12, 1–11. This bit works with system software version 6.0.4 and later.

`togChar12HourBit`
If this bit is set, modifying the hour by character is limited to the 12-hour range defined by `togCharZCycleBit`, mapped to the appropriate half of the 24-hour range, as determined by the `pm` field. This bit works with system software version 6.0.4 and later.

`smallDateBit`
If this bit is set, the valid date and time are restricted to the range of the system global variable `Time`—that is, between midnight on January 1, 1904 and 6:28:15 A.M. on February 6, 2040.

The low-order word of this field contains masks representing fields to be checked by the `ValidDate` function. Each mask corresponds to a value in the enumerated type `LongDateField`. You can set this field to check the `era` through `second` fields by using the predeclared constant `dateStdMask`. The following constants specify the `LongDateRec` fields for the `ValidDate` function to check.

```
CONST
eraMask                 = $0001;    {verify the era}
yearMask                = $0002;    {verify the year}
monthMask               = $0004;    {verify the month}
dayMask                 = $0008;    {verify the day}
hourMask                = $0010;    {verify the hour}
minuteMask              = $0020;    {verify the }
```

```
                                               { minute}
               secondMask          = $0040;    {verify the }
                                               { second}
               dateStdMask         = $007F;    {verify the era }
                                               { through second}
               dayOfWeekMask       = $0080;    {verify the day }
                                               { of the week}
               dayOfYearMask       = $0100;    {verify the day }
                                               { of the year}
               weekOfYearMask      = $0200;    {verify the week }
                                               { of the year}
               pmMask              = $0400;    {verify the }
                                               { evening (P.M.)}
```

`amChars` The trailing string to display for morning (for example, A.M.). This string is read from the numeric-format resource (resource type `'itl0'`) of the current script system.

`pmChars` The trailing to display for evening (for example, P.M.). This string is read from the numeric-format resource (resource type `'itl0'`) of the current script system.

`reserved` Reserved. Set each of the three elements of this field to 0.

The Unsigned Wide Record

The Microseconds procedure uses the unsigned wide record to return the number of microseconds elapsed since system startup time. The `UnsignedWide` data type defines the format for the unsigned wide record.

```
UnsignedWide =    {Microseconds procedure return type}
   PACKED RECORD
      hi:      LongInt;          {high-order 32 bits}
      lo:      LongInt;          {low-order 32 bits}
   END;
```

Field descriptions

`hi` The high-order 32 bits

`lo` The low-order 32 bits

Routines

The Date, Time, and Measurement Utilities provide routines you can use to read and write current date-time information, convert between internal date and time formats (for example, you can access date-time information as a number of seconds elapsed since midnight, January 1, 1904 or as a date and time), manipulate date-time information, read and write location information, and determine the current measurement system.

Some of the routines provided by the Date, Time, and Measurement Utilities were previously associated with the Script Manager or the International Utilities Package. In addition, some routines have been renamed to reflect their functions more clearly. You can access the renamed routines using more than one spelling of the routine's name, depending on the interface files supported by your development environment. For example, the `IsMetric` function is also available as the `IUMetric` function. Table 4-4 provides a summary of these changes.

Table 4-4 Renamed and relocated routines

Current name	Previous name	Former location
`DateToSeconds`	`Date2Secs`	(Unchanged)
`IsMetric`	`IUMetric`	International Utilities Package
`LongDateToSeconds`	`LongDate2Secs`	Script Manager
`LongSecondsToDate`	`LongSecs2Date`	Script Manager
`ReadLocation`	`ReadLocation`	Script Manager
`SecondsToDate`	`Secs2Date`	(Unchanged)
`ToggleDate`	`ToggleDate`	Script Manager
`ValidDate`	`ValidDate`	Script Manager
`WriteLocation`	`WriteLocation`	Script Manager

Getting the Current Date and Time

At system startup time, system software uses the `ReadDateTime` function to copy the current date-time information from the clock chip into low memory. You can access this date-time information as the number of seconds elapsed since midnight of January 1, 1904 or as a date and time. To obtain the current date-time information expressed as the number of seconds elapsed since midnight of January 1, 1904, use the `GetDateTime` procedure. To obtain the current date-time information expressed as a date and time, use the `GetTime` procedure.

IMPORTANT

If an application disables interrupts for longer than a second, the date-time information returned by the `GetDateTime` and `GetTime` procedures might not be exact. The `GetDateTime` and `GetTime` procedures are intended to provide fairly accurate time information, but not scientifically precise data. s

ReadDateTime

System software uses at system startup time the `ReadDateTime` function to copy the date-time information from the clock chip into low memory. Your application should never need to use this function.

```
FUNCTION ReadDateTime (VAR time: LongInt): OSErr;
```

`time` On return, the current time expressed as the number of seconds elapsed since midnight, January 1, 1904.

DESCRIPTION

The `ReadDateTime` function copies the current date-time information from the clock chip into low memory. It then returns in the `time` parameter a copy of the date-time information, expressed as the number of seconds elapsed since midnight, January 1, 1904.

The low-memory copy of the date and time information is accessible through the global variable `Time`.

If the clock chip cannot be read, `ReadDateTime` returns the `clkRdErr` result code. The operation might fail if the clock chip is damaged. Otherwise, the function returns the `noErr` result code.

ASSEMBLY-LANGUAGE INFORMATION

You must set up register A0 with a pointer to a long integer in which you wish to store the current date-time information. On exit, register A0 contains the same pointer to the now-changed long integer, and register D0 contains the result code.

The registers on entry and exit for this routine are

Registers on entry

A0	Pointer to long word

Registers on exit

A0	Pointer to current time
D0	Result code

RESULT CODES

`noErr`	0	No error
`clkRdErr`	–85	Unable to read clock

GetDateTime

You can use the `GetDateTime` procedure to obtain the current date-time information, expressed as the number of seconds elapsed since midnight, January 1, 1904.

```
PROCEDURE GetDateTime (VAR secs: LongInt);
```

`secs` On return, the number of seconds elapsed since midnight, January 1, 1904.

DESCRIPTION

The `GetDateTime` procedure returns in the `secs` parameter the number of seconds elapsed since midnight, January 1, 1904.

The low-memory copy of the date and time information (expressed as the number of seconds elapsed since midnight, January 1, 1904) is also accessible through the global variable `Time`.

SEE ALSO

For an example that uses the `GetDateTime` procedure to get the current date and time, see Listing 4-1 on page 4-10.

GetTime

You can use the `GetTime` procedure to obtain the current date-time information, expressed as a date and time.

```
PROCEDURE GetTime (VAR d: DateTimeRec);
```

`d` On return, the fields of the date-time record contain the current date and time.

DESCRIPTION

The `GetTime` procedure returns in the `d` parameter the current date and time. The `GetTime` procedure first calls the `GetDateTime` procedure to obtain the number of seconds elapsed since midnight, January 1, 1904. It then calls the `SecondsToDate` procedure to convert the number of seconds (returned by the `GetDateTime` procedure) into a date and time.

As an alternative to using the `GetTime` procedure, you can pass the value of the global variable `Time` to the `SecondsToDate` procedure; a `SecondsToDate(Time)` procedure call is identical to a `GetTime(d)` procedure call.

SEE ALSO

For more information about the `SecondsToDate` procedure, see page 4-38. The `GetDateTime` procedure is described on page 4-35. For sample code that uses the `GetTime` procedure to get the current date and time, see Listing 4-2 on page 4-10. The date-time record is described in detail beginning on page 4-23.

Setting the Current Date and Time

You can modify the date-time information stored in the clock chip by using the `SetDateTime` function or the `SetTime` procedure. The two routines differ in the type of arguments they require. The `SetDateTime` function requires that the new date-time information be expressed as the number of seconds elapsed since midnight of January 1, 1904 (using a value of type `LongInt`). The `SetTime` procedure requires that the new date-time information be expressed as a date and time (using a value of type `DateTimeRec`).

IMPORTANT

Users can change the current date and time stored in both the system global variable `Time` and in the clock chip by using the General Controls control panel, Date & Time control panel, or the Alarm Clock desk accessory. In general, your application should not directly change the current date-time information. If your application does need to modify the current date-time information, it should instruct the user how to change the date and time. s

SetDateTime

You can use the `SetDateTime` function to modify the date-time information stored in the clock chip. The `SetDateTime` function requires that the new date-time information be passed to the function as the number of seconds elapsed since midnight, January 1, 1904.

```
FUNCTION SetDateTime (time: LongInt): OSErr;
```

`time` The number of seconds elapsed since midnight, January 1, 1904; this value is written to the clock chip.

DESCRIPTION

The `SetDateTime` function writes the number of seconds, specified by the `time` parameter, to the clock chip. The `SetDateTime` function also updates the low-memory copy of the date-time information.

The `SetDateTime` function attempts to verify the value written by reading it back in and comparing it to the value in the low-memory copy. If a problem occurs, the `SetDateTime` function returns either the `clkRdErr` result code, because the clock chip could not be read, or the clkWrErr result code, because the time written to the clock chip could not be verified. Otherwise, the function returns the noErr result code.

ASSEMBLY-LANGUAGE INFORMATION

You must set up register D0 with the number of seconds to which you wish to change the clock chip. When the SetDateTime function returns, register D0 contains the result code.

The registers on entry and exit for this routine are

Registers on entry

D0 Seconds elapsed since midnight, January 1, 1904

Registers on exit

D0 Result code

RESULT CODES

noErr	0	No error
clkRdErr	–85	Unable to read clock
clkWrErr	–86	Time written did not verify

SEE ALSO

For sample code that uses the SetDateTime function to write date-time information (represented as a number of seconds) to the clock-chip, see Listing 4-3 on page 4-11.

SetTime

You can use the SetTime procedure to modify the date-time information in the clock chip. The SetTime requires that the new date-time information be passed to the function as a date and time.

```
PROCEDURE SetTime (d: DateTimeRec);
```

d The date and time to which to set the clock chip.

DESCRIPTION

The SetTime procedure writes the date and time specified by the d parameter to the clock chip. The SetTime procedure first converts the date and time to the number of seconds elapsed since midnight, January 1, 1904 (by calling the DateToSeconds procedure). It then writes these seconds to the clock chip and to the system global variable Time (by calling the SetDateTime function).

As an alternative to using the `SetTime` procedure, you can use the `DateToSeconds` and `SetDateTime` routines.

Note

The `SetTime` procedure does not return a result code. If you need to know whether an attempt to change the date and time information in the clock chip is successful, you must use the `SetDateTime` function. u

SEE ALSO

See page 4-23 for a description of the fields of a date-time record. For more information on the `DateToSeconds` procedure, see page 4-39. The `SetDateTime` function is described on page 4-36. For sample code that uses the `SetTime` procedure to write date-time information (represented as a date and time) to the clock-chip, see Listing 4-4 on page 4-11.

Converting Between Date-Time Formats

The Date, Time, and Measurement Utilities provide two procedure, `SecondsToDate` and `DateToSeconds`, that you can use to convert between date-time formats. You can convert a number of seconds to a date and time and a date and time to a number of seconds.

If you use a standard date-time value (used to access a number of seconds) or a date-time record (used to access a date and time) to access date-time information, you can use the `SecondsToDate` and `DateToSeconds` procedures to convert between these date-time formats. Use the `SecondsToDate` procedure to convert a number of seconds to a date and time, and use the `DateToSeconds` procedure to convert a date and time to a number of seconds.

Note

The system software uses the `SecondsToDate` and `DateToSeconds` procedures provided by the current script system. u

SecondsToDate

You can use the `SecondsToDate` procedure to convert a number of seconds elapsed since midnight, January 1, 1904 to a date and time.

```
PROCEDURE SecondsToDate (s: LongInt; VAR d: DateTimeRec);
```

`s` The number of seconds elapsed since midnight, January 1, 1904.

`d` On return, the fields of the date-time record that contain the date and time corresponding to the value indicated in the `s` parameter.

DESCRIPTION

The `SecondsToDate` procedure converts the number of seconds, specified in the `s` parameter, to a date and time. The date and time values are returned in the `d` parameter.

The `SecondsToDate` procedure is also available as the `Secs2Date` procedure.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for this routine are

Registers on entry

D0	Seconds since midnight, January 1, 1904
A0	Pointer to a date-time record

Registers on exit

A0	Pointer to a date-time record

SEE ALSO

For a complete description of the date-time record, see page 4-23.

DateToSeconds

You can use the `DateToSeconds` procedure to convert a date and time to a number of seconds elapsed since midnight, January 1, 1904.

```
PROCEDURE DateToSeconds (d: DateTimeRec; VAR s: LongInt);
```

`d` The date-time record containing the date and time to convert.

`s` On return, the number of seconds elapsed between midnight, January 1, 1904, and the time specified in the `d` parameter.

DESCRIPTION

The `DateToSeconds` procedure converts the date and time specified in the `d` parameter to the number of seconds elapsed since midnight, January 1, 1904. The number of seconds are returned in the `s` parameter. For example, specifying a date and time of 5:50 A.M. on June 13, 1990 results in 41627 being returned in the `s` parameter.

The `DateToSeconds` procedure is also available as the `Date2Secs` procedure.

ASSEMBLY-LANGUAGE INFORMATION

You must set up register A0 with a pointer to the date and time record containing the date and time you wish to convert. When `DateToSeconds` returns, register D0 contains a long integer representing the converted date and time.

The registers on entry and exit for this routine are

Registers on entry

A0 Pointer to date-time record

Registers on exit

D0 Corresponding seconds since midnight, January 1, 1904

SEE ALSO

For a complete description of the date-time record, see page 4-23.

Converting Between Long Date-Time Format

The Date, Time, and Measurement Utilities provide two procedures, `LongSecondsToDate` and `LongDateToSeconds`, that you can use to convert between long date-time formats. You can convert a number of seconds to a date and time and a date and time to a number of seconds.

If you use a long date-time value (used to access a number of seconds) or a long date-time record (used to access a date and time) to access date-time information, you can use the `LongSecondsToDate` and `LongDateToSeconds` procedures to convert between these date-time formats. Use the `LongSecondsToDate` procedure to convert a number of seconds to a date and time, and use the `LongDateToSeconds` procedure to convert a date and time to a number of seconds.

Note

The system software uses the `LongSecondsToDate` and `LongDateToSeconds` procedures provided by the current script system. u

LongSecondsToDate

You can use the `LongSecondsToDate` procedure to convert the number of seconds elapsed since midnight, January 1, 1904 to a date and time.

```
PROCEDURE LongSecondsToDate (lSecs: LongDateTime;
                             VAR lDate: LongDateRec);
```

`lSecs` The number of seconds elapsed since midnight, January 1, 1904.

`lDate` On return, the fields of the long date-time record that contain the date and time corresponding to the value indicated in the `lSecs` parameter.

DESCRIPTION

The `LongSecondsToDate` procedure converts the representation of the date-time information from a number of seconds, specified in the `lSecs` parameter, to a date and time. The date and time are returned in the `lDate` parameter as values in the date-time record. For example, specifying the number of seconds 41627 results in the date and time 5:50 A.M. on June 13, 1990 being returned in the `lDate` parameter.

The `LongSecondsToDate` procedure is also available as the `LongSecs2Date` procedure.

SEE ALSO

To learn more about the long date-time value, see the section page 4-25. For more information on the long date-time record, see page 4-26.

LongDateToSeconds

You can use the `LongDateToSeconds` procedure to convert a date and time to the number of seconds elapsed since midnight, January 1, 1904.

```
PROCEDURE LongDateToSeconds (lDate: LongDateRec;
                             VAR lSecs: LongDateTime);
```

`lDate` The long date-time record containing the date and time to convert.

`lSecs` On return, the number of seconds elapsed since midnight, January 1, 1904, and the time specified in the `lDate` parameter.

DESCRIPTION

The `LongDateToSeconds` procedure converts the representation of the date-time information from a date and time, specified in the `lDate` parameter, to the number of seconds elapsed since midnight, January 1, 1904. The number of seconds are returned as a long date-time value in the `lSecs` parameter. For example, specifying the date and time 5:50 A.M. on June 13, 1990 results in 41627 being returned in the `lSecs` parameter.

The `LongDateToSeconds` procedure is also available as the `LongDate2Secs` procedure.

SEE ALSO

To learn more about the long date-time value, see page 4-25. For more information on the long date-time record, see page 4-26.

Modifying and Verifying Long Date-Time Records

You can modify and verify the values in a long date-time record by using the `ToggleDate` function and the `ValidDate` function, respectively.

The `ToggleDate` function accepts a pointer to a toggle parameter block as a parameter. Information about the fields in the toggle parameter block appears in the following format:

Parameter block

→	`input1`	`LongInt`	Input parameter comment.
←	`output1`	`LongInt`	Output parameter comment.

The arrow on the far left indicates whether the field is an input or output parameter. You must supply values for all input parameters. The routine returns values in the output parameters. The next column shows the field name as defined in the MPW interface files, followed by the data type of that field. This matches the MPW interface name of the data type as shown in the parameter block. The fourth column contains a comment about or brief definition of the field.

ToggleDate

You can use the `ToggleDate` function to modify a date and time, by modifying one specific component of a date and time (day, hour, minute, seconds, day of week, and so on). For example, you can use the `ToggleDate` function to increase a date and time by one minute, decrease a date and time by one minute, or explicitly add or subtract a number of seconds to or from a date and time.

```
FUNCTION ToggleDate (VAR lSecs: LongDateTime;
                     field: LongDateField; delta: DateDelta;
                     ch: Integer; params: TogglePB)
                     : ToggleResults;
```

`lSecs` The date-time information to modify, expressed as the number of seconds elapsed since midnight, January 1, 1904.

`field` The name of the field in the date-time record you want modify. Use one of the `LongDateField` enumeration constants for the value of this parameter.

`delta` A signed byte specifying the action you want to perform on the value specified in the `field` parameter. Set `delta` to 1, to increase the value in the field by 1. Set `delta` to -1, to decrease the value of the field by 1. Set `delta` to 0. If you want to set the value of the field explicitly; pass the new value through the `ch` field, described next.

`ch`	If the value in the `delta` field is 0, the value of the field in the date-time record (specified by the `field` parameter) is set to the value in the `ch` parameter. If the value in the `delta` field is not equal to 0, the value in the `ch` parameter is ignored.
`params`	The settings of the toggle parameter block settings. Note that you are responsible for setting this field.

Parameter block

→	`togFlags`	`LongInt`	The fields to be checked by the `ValidDate` function.
→	`amChars`	`ResType`	A.M. characters from `'itl0'` resource.
→	`pmChars`	`ResType`	P.M. characters from `'itl0'` resource.
→	`reserved`	`ARRAY [0…3] OF LongInt`	Reserved; set each element to 0.

DESCRIPTION

The `ToggleDate` function first converts the number of seconds, specified in the `lSecs` parameter, to a date and time—making each component of the date and time (day, minute, seconds, day of week, and so on) available through a long date-time record. The `ToggleDate` function then modifies the value of the field, specified by the `field` parameter. If the value in the `delta` field is greater than 0, the value of the field (specified in the `field` parameter) increases by 1; if the value in the `delta` field is less than 0, the value of the field decreases by 1; and if the value of `delta` is 0, the value of the field is explicitly set to the value specified in the `ch` field.

After the `ToggleDate` function modifies the field, it calls the `ValidDate` function. The `ValidDate` function checks the long date-time record for correctness, using the values of the `togFlags` field in the toggle parameter block that the `ToggleDate` function passes to it. If any of the record fields are invalid, the `ValidDate` function returns a `LongDateField` value corresponding to the field in error. Otherwise, it returns the result code for `validDateFields`. Note that `ValidDate` reports only the least significant erroneous field.

After the `ToggleDate` function checks the validity of the modified field, it converts the modified date and time back into a number of seconds (the number of seconds elapsed since midnight, January 1, 1904) and returns these seconds in the `lSecs` parameter.

The following constants specify the `LongDateRec` fields for the `ValidDate` function to check:

```
CONST
   eraMask              = $0001;    {verify the era}
   yearMask             = $0002;    {verify the year}
   monthMask            = $0004;    {verify the month}
   dayMask              = $0008;    {verify the day}
   hourMask             = $0010;    {verify the hour}
   minuteMask           = $0020;    {verify the minute}
   secondMask           = $0040;    {verify the second}
```

```
dateStdMask          = $007F;    {verify the era through second}
dayOfWeekMask        = $0080;    {verify the day of the week}
dayOfYearMask        = $0100;    {verify the day of the year}
weekOfYearMask       = $0200;    {verify the week of the year}
pmMask               = $0400;    {verify the evening (P.M.)}
```

SPECIAL CONSIDERATIONS

Although `ToggleDate` does not move or purge memory, you should not call it at interrupt time.

RESULT CODES

The `ToggleDate` function returns its own set of result codes. The `ToggleResults` data type defines the result code of the `ToggleDate` function:

```
TYPE ToggleResults = Integer; {ToggleDate function return type}
```

The following list gives the result codes defined for this function:

`toggleUndefined`	0	Undefined error
`toggleOK`	1	No error
`toggleBadField`	2	Invalid field number
`toggleBadDelta`	3	Invalid delta value
`toggleBadChar`	4	Invalid character
`toggleUnknown`	5	Unknown error
`toggleBadNum`	6	Tried to use character as number
`toggleOutOfRange`	7	Out of range (synonym for `toggleErr3`)
`toggleErr3`	7	Reserved
`toggleErr4`	8	Reserved
`toggleErr5`	9	Reserved

SEE ALSO

To learn more about the `LongDateTime` data type, see page 4-25. For more information on the `LongDateRec` structure, see page 4-26. The toggle parameter block record is described on page 4-30.

For more information about the `GetIntlResource` function, see the chapter “Script Manager” in *Inside Macintosh: Text*. For details on the `UppercaseText` procedure, see the chapter “Text Utilities” in *Inside Macintosh: Text*. The `ValidDate` function is described next.

ValidDate

You can use the `ValidDate` function to verify specific date and time values in a long date-time record.

```
FUNCTION ValidDate (VAR vDate: LongDateRec; flags: LongInt;
                    VAR newSecs: LongDateTime): Integer;
```

`vDate` The long date-time record whose fields you want to verify.

`flags` The fields that you want to verify in the long date-time record.

`newSecs` The date-time information, passed by the `ToggleDate` function, that you want to verify.

DESCRIPTION

The `ValidDate` function verifies the fields, specified by the `flags` parameter, in the long date-time record specified by the `vDate` parameter. If any of the specified fields contain invalid values, the `ValidDate` function returns a `LongDateField` value indicating the field in error. Otherwise, it returns the constant `validDateFields`. Note that `ValidDate` reports only the least significant erroneous field.

The following constants specify the `LongDateRec` fields for the `ValidDate` function to check:

```
CONST
   eraMask              = $0001;    {verify the era}
   yearMask             = $0002;    {verify the year}
   monthMask            = $0004;    {verify the month}
   dayMask              = $0008;    {verify the day}
   hourMask             = $0010;    {verify the hour}
   minuteMask           = $0020;    {verify the minute}
   secondMask           = $0040;    {verify the second}
   dateStdMask          = $007F;    {verify the era through }
                                    { second}
   dayOfWeekMask        = $0080;    {verify the day of the week}
   dayOfYearMask        = $0100;    {verify the day of the year}
   weekOfYearMask       = $0200;    {verify the week of the year}
   pmMask               = $0400;    {verify the evening (P.M.)}
```

SPECIAL CONSIDERATIONS

Although `ValidDate` does not move or purge memory, you should not call it at interrupt time.

SEE ALSO

To learn more about the `LongDateTime` data type, see page 4-25. For more information on the long date-time record, see page 4-26. The `ToggleDate` function is described on page 4-42. The enumerated type `LongDateField` is described on page 4-29.

Reading and Writing Location Data

You can read and set geographic location and time-zone information using the `Readlocation` and `WriteLocation` procedures.

ReadLocation

You can use the `ReadLocation` procedure to get information about a geographic location or time zone.

```
PROCEDURE ReadLocation (VAR loc: MachineLocation);
```

`loc` On return, the fields of the geographic location record containing the geographic location and the time-zone information.

DESCRIPTION

The `ReadLocation` procedure reads the stored geographic location and time zone of the Macintosh computer from extended parameter RAM and returns it in the `loc` parameter.

You can get values for the latitude, longitude, daylight savings time (DST), or Greenwich mean time (GMT). If the geographic location record has never been set, all fields contain 0.

The latitude and longitude are stored as `Fract` values, giving accuracy to within one foot. For example, a `Fract` value of 1.0 equals 90 degrees; –1.0 equals –90 degrees; and –2.0 equals –180 degrees.

To convert these values to a degrees format, you need to convert the `Fract` values first to the `Fixed` data type, then to the `LongInt` data type. You can use the Mathematical and Logical Utilities routines `Fract2Fix` and `Fix2Long` to accomplish this task.

The DST value is a signed byte value that you can use to specify the offset for the `hour` field—whether to add one hour, subtract one hour, or make no change at all.

The GMT value is in seconds east of GMT. For example, San Francisco is at –28,800 seconds (8 hours * 3,600 seconds per hour) east of GMT. The `gmtDelta` field is a 3-byte value contained in a long word, so you must take care to get it properly.

SPECIAL CONSIDERATIONS

Although the `ReadLocation` procedure does not move or purge memory, you should not call it at interrupt time.

SEE ALSO

For more information on the geographic location record, see page 4-29. For an example of how to use the `ReadLocation` procedure to get latitude and longitude, see Listing 4-8 on page 4-19. Listing 4-9 on page 4-20 shows an application-defined procedure for obtaining the value of `gmtDelta`.

For more information on the `Fract` data type and the conversion routines `Long2Fix`, `Fix2Fract`, `Fract2Fix`, and `Fix2Long`, see the chapter "Mathematical and Logical Utilities" in this book.

WriteLocation

You can use the `WriteLocation` procedure to change the geographic location or time-zone information stored in extended parameter RAM.

```
PROCEDURE WriteLocation (loc: MachineLocation);
```

`loc` The geographic location and time-zone information to write to the extended parameter RAM.

DESCRIPTION

The `WriteLocation` procedure takes the geographic location and time-zone information, specified in the `loc` parameter, and writes it to the extended parameter RAM.

The latitude and longitude are stored in the geographic location record as `Fract` values, giving accuracy to within 1 foot. For example, a `Fract` value of 1.0 equals 90 degrees; –1.0 equals –90 degrees; and –2.0 equals –180 degrees.

To store latitude and longitude values, you need to convert them first to the `Fixed` data type, then to the `Fract` data type. You can use the Operating System Utilities routines `Long2Fix` and `Fix2Fract` to accomplish this task. Listing 4-8 on page 4-19 shows a procedure that converts San Francisco's latitude and longitude to `Fract` values, then writes the `Fract` values to extended parameter RAM using the `WriteLocation` procedure.

The daylight savings time value is a signed byte value that you can use to specify the offset for the `hour` field—whether to add one hour, subtract one hour, or make no change at all.

The Greenwich mean time value is in seconds east of GMT. For example, San Francisco is at –28,800 seconds (8 hours * 3,600 seconds per hour) east of GMT. The `gmtDelta` field is a 3-byte value contained in a long word, so you must take care to set it properly. When writing gmtDelta, you should mask off the top byte because it is reserved. In addition, you should preserve the value of dlsDelta. Listing 4-10 on page 4-21 shows a procedure that writes gmtDelta, with the top byte masked off, while preserving the value of dlsDelta.

SPECIAL CONSIDERATIONS

Although WriteLocation does not move or purge memory, you should not call it at interrupt time.

SEE ALSO

For more information on the geographic location record, see page 4-29. For more information on the Fract data type and the conversion routines Long2Fix, Fix2Fract, Fract2Fix, and Fix2Long, see the chapter "Mathematical and Logical Utilities" in this book.

Determining the Measurement System

You can determine the type of measurement system that is used by the current script system by the using the IsMetric function.

IsMetric

You can use the IsMetric function to determine whether the current script system is using the metric system (also called the International System of Units) or the English system of measurement (also called the British imperial system). The IsMetric function is also available as the IUMetric function.

```
FUNCTION IsMetric: BOOLEAN;
```

DESCRIPTION

The IsMetric function examines the metricSys field of the numeric-format resource (resource type 'itl0') to determine if the current script is using the metric system. A value of 255 in the metricSys field indicates that the metric system (centimeters, kilometers, milligrams, degrees Celsius, and so on) is being used. In this case, the IsMetric function returns a value of TRUE. A value of 0 in the metricSys field indicates that the English system of measurement (inches, miles, ounces, degrees Fahrenheit, and so on) is used. In that case, the IsMetric function returns a value of FALSE.

If you want to use units of measurement different from that of the current script, you need to override the value of the metricSys field in the current numeric-format resource (resource type 'itl0'). You can do this by using your own version of the numeric-format resource instead of the current script system's default international resource.

SPECIAL CONSIDERATIONS

The IsMetric function may move or purge blocks in the heap; calling it may cause problems if you've dereferenced a handle. You should not call this function from within interrupt code, such as in a completion routine or a VBL task.

SEE ALSO

For a complete description of the international numeric-format resource (resource type 'itl0') and how to use it, see the appendix "International Resources" in *Inside Macintosh: Text*.

For information on how to replace a script system's default international resources, see the chapter "Script Manager" in *Inside Macintosh: Text*.

Measuring Time

You can measure the number of elapsed microseconds since system startup, using the Microseconds procedure.

Microseconds

You can use the Microseconds procedure to determine the number of microseconds that have elapsed since system startup time.

```
PROCEDURE Microseconds (VAR microTickCount: UnsignedWide);
```

microsecondCount
: The number of microseconds elapsed since system startup.

DESCRIPTION

The Microseconds procedure returns, in the microsecondCount parameter, the number of microseconds that has elapsed since system startup time.

SEE ALSO

For information about the return type for this procedure—the UnsignedWide record—see page 4-32. For an example of how to use the Microseconds procedure, see Listing 4-11 on page 4-21.

Summary of the Date, Time, and Measurement Utilities

Pascal Summary

Constants

```
CONST
   {date equates for ToggleDate control bits}
   validDateFields      = -1;       {date fields are valid}
   genCdevRangeBit      = 27;       {restrict date/time to range used by }
                                    { General Controls control panel}
   togDelta12HourBit    = 28;       {if toggling hour up/down, restrict to }
                                    { 12-hour range}
   togCharZCycleBit     = 29;       {modifier for togChar12HourBit to }
                                    { accept hours 0..11 only}
   togChar12HourBit     = 30;       {if toggling hour by char, accept }
                                    { hours 1..12 only}
   smallDateBit         = 31;       {restrict valid date/time to range }
                                    { of Time global}

   {long date-time record field masks}
   eraMask              = $0001;    {era}
   yearMask             = $0002;    {year}
   monthMask            = $0004;    {month}
   dayMask              = $0008;    {day}
   hourMask             = $0010;    {hour}
   minuteMask           = $0020;    {minute}
   secondMask           = $0040;    {second}
   dayOfWeekMask        = $0080;    {day of the week}
   dayOfYearMask        = $0100;    {day of the year}
   weekOfYearMask       = $0200;    {week of the year}
   pmMask               = $0400;    {evening (P.M.)}

   {default value for togFlags field in the toggle parameter block }
   { and default value for the flags parameter passed to the Verify function}
   dateStdMask          = $007F;    {default value for checking era }
                                    { through second fields}
```

Data Types

```
TYPE

   DateTimeRec =              {date-time record}
   RECORD
      year:       Integer;    {year}
      month:      Integer;    {month}
      day:        Integer;    {day of the month}
      hour:       Integer;    {hour}
      minute:     Integer;    {minute}
      second:     Integer;    {second}
      dayOfWeek:  Integer;    {day of the week}
   END;

   LongDateField =   {long date field enumeration}
                     (eraField, yearField, monthField, dayField,
                     hourField, minuteField, secondField,dayOfWeekField,
                     dayOfYearField,weekOfYearField, pmField, res1Field,
                     res2Field, res3Field);
   LongDateTime = comp;       {date and time in 64-bit SANE comp format}

   LongDateCvt =              {long date-time conversion record}
   RECORD
      CASE Integer OF
         0:
            (c:      comp);   {copy field into a variable of type }
                              { LongDateTime}
         1:
            (lHigh:  LongInt; {high-order 32 bits}
             lLow:   LongInt);{low-order 32 bits}
   END;

   LongDateRec =              {long date-time record}
   RECORD
      CASE Integer OF
         0:
            (era:          Integer;       {era}
             year:         Integer;       {year}
             month:        Integer;       {month}
             day:          Integer;       {day of the month}
             hour:         Integer;       {hour}
             minute:       Integer;       {minute}
             second:       Integer;       {second}
```

```
          dayOfWeek:    Integer;       {day of the week}
          dayOfYear:    Integer;       {day of the year}
          weekOfYear:   Integer;       {week of the year}
          pm:           Integer;       {half of day--0 for morning, }
                                       { 1 for evening}
          res1:         Integer;       {reserved}
          res2:         Integer;       {reserved}
          res3:         Integer);      {reserved}
      1:                               {index by LongDateField}
         (list:         ARRAY[0..13] OF Integer);
      2:
         (eraAlt:       Integer;       {era}
          oldDate:      DateTimeRec);  {date-time record}
END;

TogglePB =                       {toggle parameter block}
RECORD
   togFlags:      LongInt;    {flags}
   amChars:       ResType;    {from 'itl0' resource, but made uppercase}
   pmChars:       ResType;    {from 'itl0' resource, but made uppercase}
                              {reserved}
   reserved:      ARRAY[0..3] OF LongInt;

END;

ToggleResults = Integer;         {ToggleDate function return type}

DateDelta = SignedByte;          {ToggleDate function delta field type}


MachineLocation =                   {geographic location record}
RECORD
   latitude:            Fract;      {latitude}
   longitude:           Fract;      {longitude}
   CASE Integer OF
      0:
         (dlsDelta:     SignedByte);{daylight savings time}
      1:
         (gmtDelta:     LongInt);   {Greenwich mean time}
END;
```

```
UnsignedWide =          {Microseconds procedure return type}
PACKED RECORD
hi:   longInt;       {high-order 32 bits}
lo:   longInt;       {low-order 32 bits}
END;
```

Routines

Getting the Current Date and Time

```
FUNCTION ReadDateTime       (VAR time: LongInt) : OSErr;
PROCEDURE GetDateTime       (VAR secs: LongInt);
PROCEDURE GetTime           (VAR d: DateTimeRec);
```

Setting the Current Date and Time

```
FUNCTION SetDateTime        (time: LongInt) : OSErr;
PROCEDURE SetTime           (d: DateTimeRec);
```

Converting Between Date-Time Formats

```
{each procedure has two spellings, see Table 4-4 for the alternate spelling}
PROCEDURE SecondsToDate     (secs: LongInt; VAR d: DateTimeRec);
PROCEDURE DateToSeconds     (d: DateTimeRec; VAR secs: LongInt);
```

Converting Between Long Date-Time Formats

```
{each procedure has two spellings, see Table 4-4 for the alternate spelling}
PROCEDURE LongSecondsToDate (VAR lSecs: LongDateTime;
                             VAR lDate: LongDateRec);
PROCEDURE LongDateToSeconds (lDate: LongDateRec; VAR lSecs: LongDateTime);
```

Modifying and Verifying Long Date-Time Records

```
FUNCTION ToggleDate         (VAR lSecs: LongDateTime; field: LongDateField;
                             delta: DateDelta; ch: Integer;
                             params: TogglePB): ToggleResults;
FUNCTION ValidDate          (vDate: LongDateRec; flags: LongInt;
                             VAR newSecs: LongDateTime): Integer;
```

Reading and Writing Location Data

```
PROCEDURE ReadLocation      (VAR loc: MachineLocation);
PROCEDURE WriteLocation     (VAR loc: MachineLocation);
```

Determining the Measurement System

```
{this function has two spellings, see Table 4-4 for the alternate spelling}
FUNCTION IsMetric:          Boolean;
```

Measuring Time

```
PROCEDURE Microseconds      (VAR microTickCount UnsignedWide);
```

C Summary

Constants

```
enum
{
   /*date equates for ToggleDate control bits*/
   validDateFields      = -1,       /*date fields are valid*/
   genCdevRangeBit      = 27,       /*restrict date/time to range used by */
                                    /* General Controls control panel*/
   togDelta12HourBit    = 28,       /*if toggling hour up/down, restrict */
                                    /* to 12-hour range*/
   togCharZCycleBit     = 29,       /*modifier for TogChar12HourBit to */
                                    /* accept hours 0..11 only*/
   togChar12HourBit     = 30,       /*if toggling hour by char, accept */
                                    /* hours 1..12 only*/
   smallDateBit         = 31,       /*restrict valid date/time to range */
                                    /* of Time global*/

   /*long date-time record field masks*/
   eraMask              = 0x0001,   /*era*/
   yearMask             = 0x0002,   /*year*/
   monthMask            = 0x0004,   /*day*/
   dayMask              = 0x0008,   /*month*/
   hourMask             = 0x0010,   /*hour*/
   minuteMask           = 0x0020,   /*minute*/
   secondMask           = 0x0040,   /*second*/
   dayOfWeekMask        = 0x0080,   /*day of the week*/
   dayOfYearMask        = 0x0100,   /*day of the year*/
   weekOfYearMask       = 0x0200,   /*week of the year*/
   pmMask               = 0x0400    /*evening (P.M.)*/
};
```

```
enum
{
   /*default value for togFlags field in the toggle parameter block and */
   /* default value for the flags parameter passed to the Verify function*/
   dateStdMask          = 0x007F,   /*default value for checking era */
                                    /* through second fields*/
};
```

Data Types

```
struct DateTimeRec          /*date-time record*/
{
   short       year;        /*year*/
   short       month;       /*month*/
   short       day;         /*day of the month*/
   short       hour;        /*hour*/
   short       minute;      /*minute*/
   short       second;      /*second*/
   short       dayOfWeek;   /*day of the week*/
};
typedef struct DateTimeRec DateTimeRec;

enum                        /*long date field enumeration*/
{
   eraField, yearField, monthField, dayField, hourField, minuteField,
   secondField,dayOfWeekField, dayOfYearField, weekOfYearField, pmField,
   res1Field, res2Field, res3Field
};
typedef unsigned char LongDateField;

typedef comp LongDateTime;    /*date and time in 64-bit SANE comp format*/

union LongDateCvt             /*long date-time conversion record*/
{
   comp     c;                /*copy field into a LongDateTime variable*/
   struct
   {
      long  lHigh;            /*high-order 32 bits*/
      long  lLow;             /*low-order 32 bits*/
   } hl;
};
typedef union LongDateCvt LongDateCvt;
```

```
union LongDateRec              /*long date-time record*/
{
   struct
   {
      short era;               /*era*/
      short year;              /*year*/
      short month;             /*month*/
      short day;               /*day of the month*/
      short hour;              /*hour*/
      short minute;            /*minute*/
      short second;            /*second*/
      short dayOfWeek;         /*day of the week*/
      short dayOfYear;         /*day of the year*/
      short weekOfYear;        /*week of the year*/
      short pm;                /*half of day--0 for morning, 1 for evening*/
      short res1;              /*reserved*/
      short res2;              /*reserved*/
      short res3;              /*reserved*/
   } ld;
   short list[14];             /*index by LongDateField*/
   struct
   {
      short       eraAlt;      /*era*/
      DateTimeRec oldDate;     /*date-time record*/
   } od;
};
typedef union LongDateRec LongDateRec;

struct TogglePB                /*toggle parameter block*/
{
   long     togFlags;          /*flags*/
   ResType  amChars;           /*from 'itl0' resource, but made uppercase*/
   ResType  pmChars;           /*from 'itl0' resource, but made uppercase*/
   long     reserved[4];       /*reserved*/
};
typedef struct TogglePB TogglePB;

typedef short ToggleResults;  /*ToggleDate function return type*/

typedef char DateDelta;       /*ToggleDate function delta field type*/

struct MachineLocation         /*geographic location record*/
{
   Fract    latitude;          /*latitude*/
```

```
   Fract    longitude;        /*longitude*/
   union
   {
      char  dlsDelta;         /*daylight saving time*/
      long  gmtDelta;         /*Greenwich mean time*/
   } gmtFlags;
};
typedef struct MachineLocation MachineLocation;

struct UnsignedWide           /*Microseconds procedure return type*/
{
   unsigned long     hi;      /*high-order 32 bits*/
   unsigned long     lo;      /*high-order 32 bits*/
};
typedef struct UnsignedWide UnsignedWide;
```

Routines

Getting the Current Date and Time

```
pascal OSErr ReadDateTime   (unsigned long *time);
pascal void GetDateTime     (unsigned long *secs);
pascal void GetTime         (DateTimeRec *d);
```

Setting the Current Date and Time

```
pascal OSErr SetDateTime    (unsigned long time);
pascal void SetTime         (const DateTimeRec *d);
```

Converting Between Date-Time Formats

```
{each procedure has two spellings, see Table 4-4 for the alternate spelling}
pascal void SecondsToDate   (unsigned long secs, DateTimeRec *d);
pascal void DateToSeconds   (const DateTimeRec *d, unsigned long *secs);
```

Converting Between Long Date-Time Formats

```
{each procedure has two spellings, see Table 4-4 for the alternate spelling}
pascal void LongSecondsToDate
                            (LongDateTime *lSecs, LongDateRec *lDate);
pascal void LongDateToSeconds
                            (const LongDateRec *lDate, LongDateTime *lSecs);
```

Modifying and Verifying Long Date-Time Records

```
pascal ToggleResults ToggleDate
                            (LongDateTime *lSecs, LongDateField field,
                             DateDelta delta, short ch,
                             const TogglePB *params);
pascal short ValidDate      (const LongDateRec vDate, long flags,
                              LongDateTime *newSecs);
```

Reading and Writing Location Data

```
pascal void ReadLocation    (MachineLocation *loc);
pascal void WriteLocation   (MachineLocation *loc);
```

Determining the Measurement System

```
{this functiosn has two spellings, see Table 4-4 for the alternate spelling}
pascal Boolean IsMetric     (void);
```

Measuring Time

```
pascal void Microseconds    (UnsignedWide *microTickCount);
```

Assembly-Language Summary

Data Structures

Date-Time Record

0	dtYear	word	year
2	dtMonth	word	month
4	dtDay	word	day of the month
6	dtHour	word	hour
8	dtMinute	word	minute
10	dtSecond	word	second
12	dtDayOfWeek	word	day of the week

Long Date Field Enumeration

0	eraField	byte	era
1	yearField	byte	year
2	monthField	byte	month
3	dayField	byte	day of the month
4	hourField	byte	hour
5	minuteField	byte	minute
6	secondField	byte	second
7	dayOfWeekField	byte	day of the week
8	dayOfYearField	byte	day of the year
9	weekOfYearField	byte	week of the year
10	pmField	byte	pm
11	res1Field	byte	reserved
12	res2Field	byte	reserved
13	res3Field	byte	reserved

Long Date-Time Value

0	highLong	long	high-order 32 bits
4	lowLong	long	low-order 32 bits

Long Date-Time Record

0	era	word	era
2	year	word	year
4	month	word	month
6	day	word	day of the month
8	hour	word	hour
10	minute	word	minute
12	second	word	second
14	dayOfWeek	word	day of the week
16	dayOfYear	word	day of the year
18	weekOfYear	word	week of the year
20	pm	word	half of day, morning or evening
22	ldReserved	6 bytes	reserved

Geographic Location Record

0	latitude	long	latitude
4	longitude	long	longitude
8	dlsDelta	byte	daylight savings time
9	gmtDelta	3 bytes	Greenwich mean time

Toggle Parameter Block

0	togFlags	long	flags
2	amChars	word	ResType from 'itl0' made uppercase
4	pmChars	word	ResType from 'itl0' made uppercase
6	reserved	word	reserved

Unsigned Wide Record

0	hi	long	high-order 32 bits
4	lo	long	low-order 32 bits

Global Variables

Time	The number of seconds since midnight, January 1, 1904

Result Codes

toggleErr5	9	Reserved
toggleErr4	8	Reserved
toggleErr3	7	Reserved
toggleOutOfRange	7	Out of range (synonym for toggleErr3)
toggleBadNum	6	Tried to use character as number
toggleUnknown	5	Unknown error
toggleBadChar	4	Invalid character
toggleBadDelta	3	Invalid delta value
toggleBadField	2	Invalid field number
toggleOK	1	No error
toggleUndefined	0	Undefined error
noErr	0	No error
clkRdErr	–85	Unable to read clock
clkWrErr	–86	Time written did not verify

Control Panel Extensions

Contents

This chapter describes how you can create a control panel extension to add a panel to an existing control panel. Some of the control panels provided with the Macintosh system software allow you to install additional panels to control settings for your own devices. You can also install additional panels to allow the user to manipulate other system-wide settings or configuration data not directly associated with any hardware.

You need to read this chapter if you are developing hardware or software that provides system-wide services and that has one or more settings that a user might want to alter. However, you need to read this chapter only if some existing control panel is extensible in the way described in the next section, "About Control Panel Extensions." Currently, only certain versions of the Sound control panel and the Video control panel allow you to add panels by creating control panel extensions. In all other cases, you'll need to create a control panel to handle any necessary user interaction. For a complete description of how to create a control panel, see the chapter "Control Panels" in *Inside Macintosh: More Macintosh Toolbox*. (Also see the chapter "Control Panels" if you are the manufacturer of a video card and need to create an extension to the Monitors control panel.)

To use this chapter, you should already be familiar with creating dialog boxes and handling user actions in them. See the chapters "Dialog Manager" and "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for more information about these topics. Because control panel extensions are components, you also need to be familiar with the Component Manager, described in *Inside Macintosh: More Macintosh Toolbox*.

Note

The programming interface to control panel extensions described in this chapter is virtually identical to the programming interface to sequence grabber panel components, described in the chapter "Sequence Grabber Panel Components" in *Inside Macintosh: QuickTime Components.* If you are programming in C, you might find it useful to consult the source code samples, which are in C in that chapter. u

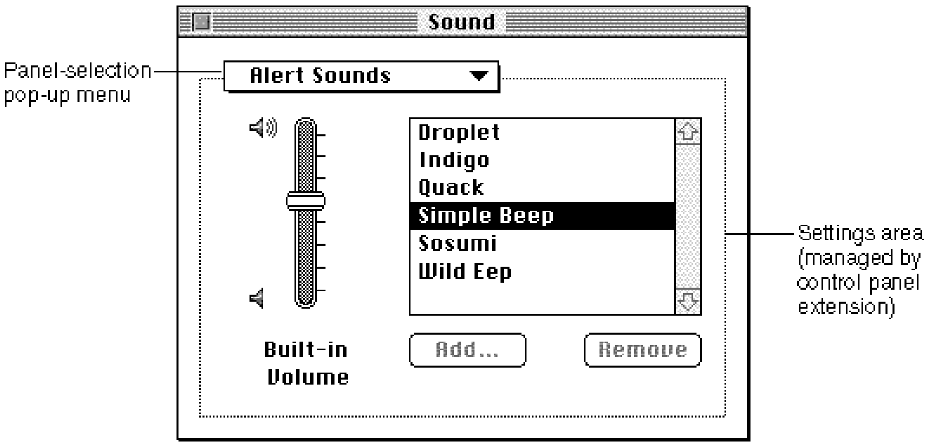
About Control Panel Extensions

A **control panel** manages the settings of a system-wide feature, such as the amount of memory allocated to a disk cache, the speed at which the cursor moves relative to movement of the mouse, the background pattern used on the desktop, or the picture displayed by a screen saver. On the screen, a control panel appears as a modeless dialog box with controls that let users specify basic settings and preferences for the feature. A control panel such as the General Controls or Color control panel usually defines the contents of its display area and manages the settings of its own controls; however, a control panel such as the Sound or Video control panel may use one or more control panel extensions to manage parts of its display area. The rest of this chapter discusses control panels that use control panel extensions and describes how to write a control panel extension. For information on control panels that do not use control panel extensions, see the chapter "Control Panels" in *Inside Macintosh: More Macintosh Toolbox*.

A **control panel extension** works in conjunction with and at the request of a control panel to manage a certain part of the control panel's display area. The area managed by a control panel extension is called a **panel**. A panel contains controls and other items related to the features managed by the control panel extension. These items are the same items used in dialog and alert boxes. The control panel extension is responsible for handling events in its panel and for responding to requests from its associated control panel. A control panel that uses control panel extensions typically includes a pop-up menu, from which the user chooses which panel to view. The control panel displays the current panel's items within a dotted-line border extending from its pop-up menu.

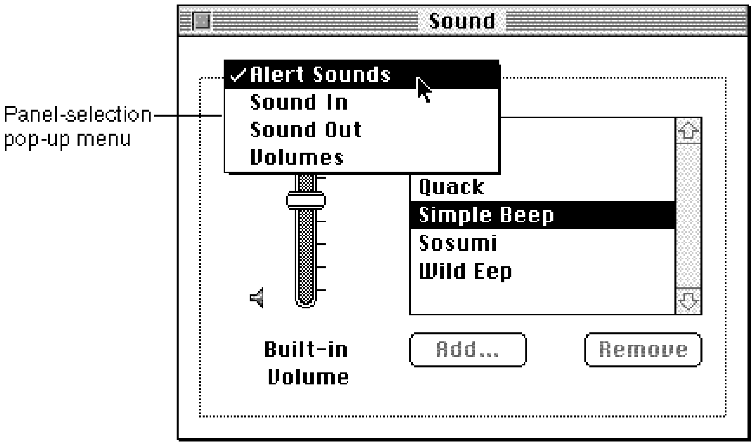
Figure 5-1 shows the Sound control panel introduced with version 3.0 of the Sound Manager. The Sound control panel manages the pop-up menu in its display area. When the user chooses a menu item from the pop-up menu, the Sound control panel uses a control panel extension to display the panel corresponding to the user's choice. The control panel extension is responsible for managing the area within its panel.

Figure 5-1 A control panel with a panel



As shown in Figure 5-1, control panels that use control panel extensions typically include a pop-up menu from which the user can choose one or more items. Each item typically corresponds to a feature managed by a control panel extension. For example, Figure 5-2 shows the menu items in the pop-up menu of the Sound control panel. This pop-up menu can have the items Alert Sounds, Sound In, Sound Out, or Volumes as well as items corresponding to other control panel extensions. Apple supplies the control panel extensions for Alert Sounds, Sound In, Sound Out, and Volumes.

Figure 5-2 Panel-selection pop-up menu in a control panel



As shown in Figure 5-2, when the user chooses the Alert Sounds pop-up menu item, the Sound control panel calls the Alert Sounds control panel extension to display a panel and manage the items associated with the extension. The Alert Sounds control panel extension is responsible for the items within its panel: the volume slider, the scrollable list of sounds, and the two buttons.

The user interface for a panel consists of the display area defined by the owning control panel and includes the items defined and managed by your panel. Each control panel that supports control panel extensions defines the bounding area in which panels can place items. For example, the panel inserted into the Sound control panel is given a default rectangle size of 185 pixels in height, and 302 pixels in width. All of the items for this panel must be placed at least 13 pixels from the dialog's border.

Control panel extensions are implemented as components. A control panel uses the Component Manager to request services from the appropriate control panel extension as needed. For example, when the user opens a control panel, the Finder sends the control panel an initialization request. In response to this request, the control panel uses the Component Manager to determine which control panel extensions are available and includes the name of each available extension in its pop-up menu.

The control panel then uses the Component Manager to open the control panel extension associated with the current pop-up menu item and set up the panel. (For example, if the Sound control panel determines that its panel area should display information for Alert Sounds panel, the Sound control panel opens the Alert Sounds control panel extension.) As directed, the control panel extension returns information about its controls and other items in its panel area and sets initial values for these items. The control panel continues to use the Component Manager to communicate with the control panel extension, requesting it to respond to user events within the panel area. When the user closes the control panel, the control panel uses the Component Manager to close the current control panel extension before the control panel terminates.

This chapter describes the general structure of a control panel extension. For information on providing a control panel extension for a specific control panel, see the documentation describing that control panel. For example, for information on the Video control panel, see the chapter “Sequence Grabber Panel Components” in *Inside Macintosh: QuickTime Components.*

Writing a Control Panel Extension

A control panel extension is a component that works with a control panel to manage a panel—a certain part of an existing control panel’s display area. Because a control panel extension is a component, it must be able to respond to standard request codes sent by the Component Manager. In addition, a control panel extension must

- return information about the items in its panel
- handle user actions and other events in its panel
- get and set the values of its items

This section describes how to write a control panel extension. You need to read this section if you want to create a new panel for an existing control panel.

Creating a Component Resource for a Control Panel Extension

A control panel extension is stored as a component resource. It contains a number of resources, including icons, strings, pictures, and the standard component resource (a resource of type `'thng'`) required of any Component Manager component. In addition, a control panel extension must contain code to handle required request codes passed to it by the Component Manager as well as panel-specific request codes. A control panel extension also usually contains an item list resource (`'DITL'`) that defines the items for the panel.

Note

For complete details on components and their structure, see the chapter “Component Manager” in *Inside Macintosh: More Macintosh Toolbox*. This section provides specific information about control panel extensions. u

The component resource binds together all the relevant resources contained in a component; its structure is defined by the `ComponentResource` data type.

```
TYPE ComponentResource =
   RECORD
      cd:              ComponentDescription;
      component:       ResourceSpec;
      componentName:   ResourceSpec;
```

```
      componentInfo:    ResourceSpec;
      componentIcon:    ResourceSpec;
   END;
```

The cd field contains a component description record that specifies the component type, subtype, manufacturer, and flags. The component field specifies the resource type and resource ID of the component's executable code. By convention, this resource should be of the same type as the componentType field of the component description record referenced through the cd field. (You can, however, specify some other resource type if you wish.) The resource ID can be any integer greater than or equal to 128. See the next section, "Dispatching to Control Panel Extension-Defined Routines," for further information about this code resource. The ResourceSpec data type has this structure:

```
TYPE ResourceSpec =
   RECORD
      resourceType:     ResType;
      resourceID:       Integer;
   END;
```

The componentName field of the ResourceSpec data type specifies the resource type and resource ID of the resource that contains the component's name. Usually the name is contained in a resource of type 'STR '. This string should be as short as possible.

The componentInfo field specifies the resource type and resource ID of the resource that contains a description of the component. Usually the description is contained in a resource of type 'STR '. This information is not currently used by control panels, but some development tools may use it.

The componentIcon field specifies the resource type and resource ID of the resource that contains an icon for the component. Usually the icon is contained in a resource of type 'ICON'. This icon is not currently used by control panels, but some development tools may use it.

As previously described, the cd field of the ComponentResource structure is a component description record, which includes additional information about the component. A component description record is defined by the ComponentDescription data structure.

```
TYPE ComponentDescription =
   RECORD
      componentType:          LongInt;
      componentSubType:       LongInt;
      componentManufacturer:  LongInt;
      componentFlags:         LongInt;
      componentFlagsMask:     LongInt;
   END;
```

For control panel extensions, the `componentType` field must be set to a value associated with an existing control panel. Currently, you can specify one of two available component types for control panel extensions:

```
CONST
   SoundPanelType              = 'sndP';   {sound panel}
   VideoPanelType              = 'vidP';   {video panel}
```

In addition, the `componentSubType` field must be set to a value that indicates the type of control panel services your panel provides. For example, the Apple-supplied control panel extensions for the Sound control panel have these subtypes:

```
CONST
   kAlertSoundsPanel           = 'alrt';   {alert sounds panel}
   kInputsPanel                = 'mics';   {input devices panel}
   kOutputsPanel               = 'spek';   {output devices panel}
   kVolumesSubType             = 'vols';   {volumes panel}
```

If you add panels to the Sound control panel, you should assign some other subtype.

Note

Apple reserves for its own uses all types and subtypes composed solely of lowercase letters. u

You can assign any value you like to the `componentManufacturer` field; typically, you put the signature of your control panel extension in this field.

The `componentFlags` field of the component description for a control panel extension contains bit flags that encode information about the extension. Currently, you can use this field to specify whether the control panel should open your extension's resource file.

```
CONST
   channelFlagDontOpenResFile = 2;     {do not open resource file}
```

The `channelFlagDontOpenResFile` bit indicates to the owning control panel whether or not to open the component's resource file. When bit 2 is cleared (set to 0), the control panel opens the component's resource file for you. In general, this is the most convenient way to gain access to your extension's resources. However, if the component is linked with an application and does not have its own resource file, you might not want the control panel to try to open the resource file. In that case, set this bit to 1.

You should set the `componentFlagsMask` field to 0.

Your control panel extension is contained in a resource file. The creator of the file can be any type you wish, but the type of the file must be `'thng'`. If the extension contains a `'BNDL'` resource, then the file's bundle bit must be set. Control panel extensions should be located in the Control Panels folder (or Extensions folder if the component needs automatic registration).

Listing 5-1 shows the Rez listing of a component resource that describes a control panel extension.

Listing 5-1 A component resource for a control panel extension

```
resource 'thng' (kExamplePanelID, kExampleName, purgeable) {
   kExamplePanelComponentType,   /*component type*/
   kExamplePanelSubType,         /*component subtype*/
   kExampleManufacturer,         /*component manufacturer*/
   cmpWantsRegisterMessage,      /*control flags*/
   0,                            /*control flags mask*/
                                 /*code res type, res ID*/
   kExamplePanelCodeType, kExamplePanelCodeID,
   'STR ', kExamplePanelNameID,  /*name res type, res ID*/
   'STR ', kExamplePanelInfoID,  /*info res type, res ID*/
   'ICON', kExamplePanelIconID   /*icon res type, res ID*/
};
```

Dispatching to Control Panel Extension-Defined Routines

As explained in the previous section, the code stored in the control panel extension component should be contained in a resource whose resource type matches the type stored in the `componentType` field of the component description record. The Component Manager expects that the entry point in this resource is a function having this format:

```
FUNCTION MyPanelDispatch (VAR params: ComponentParameters;
                          storage: Handle): ComponentResult;
```

Whenever the Component Manager receives a request for your control panel extension, it calls your component's entry point and passes any parameters, along with information about the current connection, in a component parameters record. The Component Manager also passes a handle to the global storage (if any) associated with that instance of your component.

When your component receives a request, it should examine the parameters to determine the nature of the request, perform the appropriate processing, set an error code if necessary, and return an appropriate function result to the Component Manager.

The component parameters record is defined by a data structure of type `ComponentParameters`. The `what` field of this record contains a value that specifies the type of request. Your component's entry point should interpret the request code and possibly dispatch to some other subroutine. Your extension must be able to handle the required request codes, defined by these constants:

```
CONST
   kComponentOpenSelect          = -1;
   kComponentCloseSelect         = -2;
   kComponentCanDoSelect         = -3;
   kComponentVersionSelect       = -4;
```

Note
For complete details on required component request codes, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*. u

In addition, your extension must be able to respond to panel-specific request codes. Currently, you need to be able to handle these request codes:

```
CONST
   kPanelGetDitlSelect        = 0;  {get panel's item list}
   kPanelGetTitleSelect       = 1;  {get panel's name}
   kPanelInstallSelect        = 2;  {restore item settings}
   kPanelEventSelect          = 3;  {handle event in panel}
   kPanelItemSelect           = 4;  {handle click in a panel item}
   kPanelRemoveSelect         = 5;  {panel is about to be removed}
   kPanelValidateInputSelect  = 6;  {validate panel settings}
   kPanelGetSettingsSelect    = 7;  {get panel settings}
   kPanelSetSettingsSelect    = 8;  {set panel settings}
```

You should respond to these request codes by performing the requested action. To service the request, your component may need to access additional information provided in the `params` field of the component parameters record. The `params` field is an array that contains the parameters specified by the control panel that called your component. You can directly extract the parameters from this array, or you can use the `CallComponentFunction` or `CallComponentFunctionWithStorage` function to extract the parameters from this array and pass these parameters to a subroutine of your component.

Listing 5-2 illustrates how to define the entry-point routine for a control panel extension.

Listing 5-2 Handling Component Manager request codes

```
FUNCTION MyPanelDispatch (VAR params: ComponentParameters; storage: Handle)
                         : ComponentResult;
CONST
   kPanelVersion = 1;
   kExamplePanelDITLID = 128;
   kDefaultButton = 1;
   kExampleOtherButton = 2;
   kExampleBeepButton = 3;
   kExampleRadioButton1 = 4;
   kExampleRadioButton1 = 5;
TYPE
   PanelGlobalsRec =          {global storage for this component instance}
      RECORD
         itemOffset:    Integer;
```

```
         mySelf:        ComponentInstance;
      END;
   PanelGlobalsPtr = ^PanelGlobalsRec;
   PanelGlobalsHandle = ^PanelGlobalsPtr;
VAR
   myGlobals:     PanelGlobalsHandle;
   selector:      Integer;
BEGIN
   CASE params.what OF
      kComponentOpenSelect:                  {component is opening}
         BEGIN
            myGlobals :=
               PanelGlobalsHandle(NewHandleClear(SizeOf(PanelGlobalsRec)));
            IF myGlobals <> NIL THEN
            BEGIN
               myGlobals^^.mySelf := ComponentInstance(params.params[0]);
               SetComponentInstanceStorage(myGlobals^^.mySelf,
                                           Handle(myGlobals));
               MyPanelDispatch := noErr;
            END
            ELSE
               MyPanelDispatch := MemError;
         END;

      kComponentCloseSelect:                 {component is closing; clean up}
         BEGIN
            IF storage <> NIL THEN
               DisposeHandle(storage);
            MyPanelDispatch := noErr;
         END;

      kComponentCanDoSelect:                 {indicate whether component }
                                             { supports this request code}
         BEGIN
            selector := Integer((Ptr(params.params)^));
            IF (((kComponentVersionSelect <= selector)
                  AND (selector <= kComponentOpenSelect))
               OR ((kPanelGetDitlSelect <= selector)
                  AND (selector <= kPanelSetSettingsSelect))) THEN
               MyPanelDispatch := 1 {valid request}
            ELSE
               MyPanelDispatch := 0;{invalid request}
         END;
```

```
kComponentVersionSelect:{return version number}
   MyPanelDispatch := kPanelVersion;

kPanelGetDitlSelect:    {get panel's item list}
   MyPanelDispatch := CallComponentFunctionWithStorage
                        (storage, params,
                         ComponentFunction(@MyPanelGetDITL));

kPanelInstallSelect:    {restore items' settings if necessary}
   MyPanelDispatch := CallComponentFunctionWithStorage
                        (storage, params,
                         ComponentFunction(@MyPanelInstall));

kPanelEventSelect:      {handle event in panel}
   MyPanelDispatch := CallComponentFunctionWithStorage
                        (storage, params,
                         ComponentFunction(@MyPanelEvent));

kPanelItemSelect:       {handle hit in one of panel's items}
   MyPanelDispatch := CallComponentFunctionWithStorage
                        (storage, params,
                         ComponentFunction(@MyPanelItem));

kPanelRemoveSelect:  {panel is about to be removed, respond as needed}
   MyPanelDispatch := CallComponentFunctionWithStorage
                        (storage, params,
                         ComponentFunction(@MyPanelRemove));
kPanelValidateInputSelect:{validate panel settings}
   MyPanelDispatch :=
               CallComponentFunctionWithStoMyPanelValidateInputrage
                           (storage, params,
                            ComponentFunction(@MyPanelValidateInput));

kPanelGetTitleSelect:   {get panel's name}
   MyPanelDispatch := CallComponentFunctionWithStorage
                        (storage, params,
                         ComponentFunction(@MyPanelGetTitle));

kPanelGetSettingsSelect:   {get panel settings}
   MyPanelDispatch := CallComponentFunctionWithStorage
                        (storage, params,
                        ComponentFunction(@MyPanelGetSettings));
```

```
      kPanelSetSettingsSelect:{set panel settings}
         MyPanelDispatch := CallComponentFunctionWithStorage
                              (storage, params,
                              ComponentFunction(@MyPanelSetSettings));

      OTHERWISE               {unrecognized request code}
         MyPanelDispatch := badComponentSelector;
   END; {of CASE}
END;
```

The `MyPanelDispatch` function defined in Listing 5-2 simply inspects the `what` field of the component parameters record to determine which request code to handle. For panel-specific request codes, it dispatches to the appropriate function in the control panel extension. See the following sections for more details on handling panel-specific request codes.

Your extension can be dynamically loaded or unloaded at any time. When the owning control panel first discovers the extension, it loads it into a subheap of some existing heap. In all likelihood, your extension is loaded into either the system heap or temporary memory. In some cases, however, your extension might be loaded into an application's heap. Your extension is guaranteed 32 KB of available heap space. You should do all allocation in that heap using normal Memory Manager routines.

If you need to access resources that are stored in your control panel extension, you can use the `OpenComponentResFile` and `CloseComponentResFile` functions (which are provided by the Component Manager), or you can allow the control panel to open your resource fork for you automatically by setting the appropriate component flag. The `OpenComponentResFile` routine requires the `ComponentInstance` parameter supplied to your routine. You should not call the Resource Manager routines `OpenResFile` or `CloseResFile`.

s **WARNING**

Do not leave any resource files open when your control panel extension is closed. Their maps will be left in the subheap when the subheap is freed, causing the Resource Manager to crash. s

The following sections illustrate how to write control panel extension functions that respond to panel-specific request codes.

Installing and Removing Panel Items

After opening your control panel extension, the control panel calls your control panel extension with a get-item list request followed by an install request. When your component receives a get-item list request, it should return the item list that defines the items in its panel. When your component receives an install request, it should set the default values of any items in the panel or set up any user items in the panel. For example, your component can restore previous settings as set by the user or create lists at this time. When your component receives a remove request, it should perform any processing that is necessary before the panel is removed from the display area of the control panel.

A control panel that uses your control panel extension calls your component with the get-item list request (followed by an install request) before displaying the panel to the user. If your component returns a result code of `noErr` in response to both of these request codes, the control panel displays your panel to the user.

The relevant fields in the component parameters record when your component receives a get-item list request are:

Field	Description
`what`	This field is set to `kPanelGetDitlSelect`.
`params`	The first entry in this array contains a handle to a block of memory. Your component should resize the handle as necessary and then use this memory to return an item list of the items supported by your control panel extension.

In response to a get-item list request, set your component's function result to `noErr` if your component successfully placed the item list in memory; otherwise, set it to a nonzero value.

Listing 5-3 shows an example of a control panel extension-defined routine that handles the get-item list request.

Listing 5-3 Responding to the get-item list request

```
FUNCTION MyPanelGetDITL(globals: PanelGlobalsHandle;
                        ditlHandle: Handle): ComponentResult;
BEGIN
   MyPanelGetDITL := resNotFound; {set default return value}
   ditlHandle := Get1Resource('DITL', kExamplePanelDITLID);
   IF (ditlHandle <> NIL)
   BEGIN
      DetachResource(ditlHandle);
      MyPanelGetDITL := noErr;
   END;
END;
```

The relevant fields in the component parameters record when your component receives an install request are:

Field	Description
`what`	This field is set to `kPanelInstallSelect`.
`params`	The first entry in this array contains the dialog pointer of the owning control panel. The dialog box can be a color dialog box on systems that support color windows. The second entry contains the item offset to your panel's first item.

In response to an install request, set your component's function result to `noErr` if your component successfully handled the request; otherwise, set it to a nonzero value.

Listing 5-4 shows an example of a control panel extension-defined routine that handles the install request.

Listing 5-4 Responding to the install request

```
FUNCTION MyPanelInstall(globals: PanelGlobalsHandle;
                        cpDialogPtr: DialogPtr;
                        itemOffset: Integer): ComponentResult;
BEGIN
   {restore previous settings of panel items as set by user}
   MyPanelInstall := MyRestoreSettings(globals, itemOffset,
                                       cpDialogPtr);
END;
```

The `MyPanelInstall` function shown in Listing 5-4 calls one of its own routines (`MyRestoreSettings`) to set the panel's items to the last settings chosen by the user. In response to the install request, you can also create other elements needed by your panel, such as lists.

The relevant fields in the component parameters record when your component receives a remove request are:

Field	Description
`what`	This field is set to `kPanelRemoveSelect`.
`params`	The first entry in this array contains the dialog pointer of the owning control panel. The dialog box can be a color dialog box on systems that support color windows. The second entry contains the item offset to your panel's first item.

In response to a remove request, dispose of any additional dialog data you created (for example, if you created a list, call `LDispose`), but do not dispose of your component's global storage. Also, set your component's function result to `noErr` if your component successfully handled the request; otherwise, set it to a nonzero value.

Handling Panel Items

Your control panel extension typically receives an item-select request (indicated by the kPanelItemSelect request code) when the user clicks in one of your panel's items. When your component receives an item-select request, it should perform the appropriate action for the selected item.

Note that when a click in one of your panel's items occurs, the owning control panel first sends your component an event-select request, giving your component a chance to filter the event, if necessary. A control panel sends your component an item-select request only if your component returns FALSE in the handled parameter in response to an event-select request. Typically, your component only returns FALSE in response to an event-select request if the event is a mouse event. The event-select request is discussed in detail in the next section, "Handling Events in a Panel" beginning on page 5-17.

The relevant fields in the component parameters record when your component receives an item-select request are:

Field	**Description**
what	This field is set to kPanelItemSelect.
params	The first entry in this array contains the dialog pointer of the owning control panel. The dialog box can be a color dialog box on systems that support color windows. The second entry contains the item number of the item selected by the user. Note that to map the item number to an item in your panel, you must offset the item number by the number of items in the owning control panel.

You must set your component's function result to noErr in response to an item-select request; otherwise, the owning control panel closes the panel.

Listing 5-5 shows an example of a control panel extension-defined routine that handles an item-select request.

Listing 5-5 Responding to an item-select request

```
FUNCTION MyPanelItemSelect(globals: PanelGlobalsHandle;
                           cpDialogPtr: DialogPtr;
                           itemHit: Integer): ComponentResult;
BEGIN
   MyPanelItemSelect := noErr; {set return value}
   {adjust item number to take into account control panel's items}
   itemHit := itemHit - (globals^^).itemsOffset;
   CASE itemHit OF
      kExampleBeepButton:  {user clicked beep button}
         SysBeep(40);
      kExampleOtherButton: {user clicked this button}
         MyPanelItemSelect := MyDoButton(cpDialogPtr, itemHit);
      kExampleRadioButton1:{user clicked this radio button}
```

```
         MyPanelItemSelect := MySetRadioButton(cpDialogPtr,
                                               itemHit);
      kExampleRadioButton2:{user clicked this radio button}
         MyPanelItemSelect := MySetRadioButton(cpDialogPtr,
                                               itemHit);
      kDefaultButton:     {user clicked the default button}
         MyPanelItemSelect :=
                        MyDoDefaultButtonAction(cpDialogPtr,
                                                 itemHit);
   END; {of CASE}
END;
```

Handling Events in a Panel

A control panel sends an event-select request (indicated by the `kPanelEventSelect` request code) to your extension whenever an event occurs in your panel. The event-select request is intended to provide your extension with the ability to respond just like an event filter function specified in calls to the `ModalDialog` procedure or other Dialog Manager routines. A control panel sends your extension the event-select request to give it an opportunity to intercept events in its panel and handle events before, or instead of the owning control panel. For example, you can change a keystroke into a click on an item, use idle time during null events, or track the movement of the cursor through mouse events.

The relevant fields in the component parameters record when your component receives an event-select request are:

Field	Description
`what`	This field is set to `kPanelEventSelect`.
`params`	The first entry in this array contains the dialog pointer of the owning control panel. The second entry contains the item offset to your panel's first item. Note that to map the item number to an item in your panel, you must offset the item number by the number of items in the owning control panel. The third entry contains an event record describing the event. If your extension handles the event, it should return in the fourth entry the item number of the associated panel item. On exit, your extension should indicate in the fifth entry whether it has handled the event by returning `TRUE` (handled the event) or `FALSE` (did not handle the event).

When your extension receives an event-select request, it indicates (through the fifth entry in `params`) whether it handled the event or not. Typically, your extension responds to an event-select request in this manner:

- maps the Return or Enter key to the default button, performs the action corresponding to the default button, and returns `TRUE` and the item number of the default button through entries in `params`

- maps the Esc (Escape) key or Command-period combination to the Cancel button (if any), performs the action corresponding to the Cancel button, and returns TRUE and the item number through entries in params
- updates the panel if needed (typically updating only those items that need updating apart from the standard updating performed by the Dialog Manager, such as user-defined panel items or lists) and returns TRUE and the item number of the default button through entries in params
- activates certain panel items (such as lists) as necessary and returns TRUE
- maps keyboard equivalents (if any) to corresponding item numbers, performs the corresponding action for that item number, and returns TRUE
- tracks movement of the cursor as needed (typically tracking the cursor only in those items, such as user-defined items or lists, that the Dialog Manager doesn't handle) and returns TRUE

In general, for all other events, your extension should return FALSE (in the fifth entry of params) and allow the owning control panel to handle the event. However, note that if your extension returns FALSE, the owning control panel calls your extension with the item-select request code. See the previous section, "Handling Panel Items" on page 5-16 for information on handling clicks in your panel's items.

Listing 5-6 shows an example of a control panel extension-defined routine that handles the event-select request.

Listing 5-6 Responding to an event-select request

```
FUNCTION MyPanelEvent (globals: Handle; dialog: DialogPtr;
                       itemOffset: Integer;
                       theEvent: eventRecord;
                       VAR itemHit: Integer;
                       VAR handled: Boolean): ComponentResult;
VAR
   itemType:      Integer;
   itemHandle:    Handle;
   itemRect:      Rect;
   finalTicks:    LongInt;
BEGIN
   MyPanelEvent := noErr;
   CASE theEvent.what OF
      keyDown, autoKey:
      BEGIN
         CASE ((char)(theEvent->message & charCodeMask))
            kEnterKey, kReturnKey:
            BEGIN    {respond as if user clicked Default button}
               itemHit := kDefaultButton + itemOffset;
```

```
                GetDialogItem(dialog, itemHit, itemType,
                              itemHandle, itemRect);
                HiliteControl(ControlHandle(itemHandle),inButton);
                Delay(kVisualDelay, finalTicks);
                HiliteControl(ControlHandle(itemHandle),0);
                MyPanelEvent :=
                         MyDoDefaultButtonAction(dialog, itemHit);
             END;
          OTHERWISE
       {let control panel/Dialog Mgr handle other keyboard events}
             handled := FALSE;
       END; {of CASE keyDown, autoKey}
       updateEvt:
          DoUpdatePanel(globals, dialog);
       OTHERWISE
       {let owning control panel & Dialog Mgr handle other events}
          handled := FALSE;
   END; {of CASE}
END;
```

Handling Title Requests

A control panel may send your control panel extension a title request to determine the name it should display for the panel in the control panel's pop-up menu. Note that a control panel usually uses the name of your component as the name to display.

The relevant fields in the component parameters record when your component receives a title request are:

Field	Description
`what`	This field is set to `kPanelGetTitleSelect`.
`params`	The first entry in this array contains a value that identifies a specific instance of your component. In the second entry of this array, your component should return the name you want displayed in the pop-up menu associated with your panel.

Note

Current versions of the Sound and Video control panels do not send the `kPanelGetTitleSelect` request code. u

Managing Control Panel Settings

A control panel may send the `kPanelValidateInputSelect`, `kPanelGetSettingsSelect`, or `kPanelSetSettingsSelect` request codes to your extension to request it to validate the settings of its items, or return or set the current settings of its items. If a control panel sends this request code, your extension should respond appropriately.

Note
Current versions of the Sound and Video control panels do not send the `kPanelValidateInputSelect`, `kPanelGetSettingsSelect`, or `kPanelSetSettingsSelect` request code. u

Control Panel Extensions Reference

This section describes the extension-defined routines that you can write to handle the panel-specific request codes that your control panel extension receives. See "Writing a Control Panel Extension" beginning on page 5-6 for information on creating a component that contains these extension-defined routines.

Control Panel Extension-Defined Routines

This section describes the routines you'll need to define in order to write a control panel extension. You need to write routines that respond to panel-specific request codes. The panel-specific request codes request your control panel extension to perform various actions. These actions include:

- returning an item list describing the panel's items and setting up the initial values of these items
- receiving and handling events in the panel
- getting and setting a panel's settings

Your control panel extension-defined routines should always return result codes of type `ComponentResult`. If a routine succeeds, it should return `noErr`.

See "Dispatching to Control Panel Extension-Defined Routines" beginning on page 5-9 for a description of how you call these routines from within a control panel extension.

Managing Panel Components

A control panel extension should respond to the `kPanelGetDitlSelect`, `kPanelInstallSelect`, `kPanelGetTitleSelect`, and `kPanelRemoveSelect` request codes. You typically define subroutines that the main program of your control panel extension calls (using `CallComponentFunctionWithStorage`) to handle these requests. You can choose any name for these subroutines, but by convention they're called `MyPanelGetDITL`, `MyPanelInstall`, `MyPanelGetTitle`, and `MyPanelRemove`.

When the appropriate control panel prepares to add a control panel extension's items to a control panel, it obtains a list of those items by calling the extension and specifying the kPanelGetDitlSelect request code. The control panel extension typically responds by calling a subroutine (for example, MyPanelGetDITL) to handle the request. Once the control panel has installed the items, it calls the extension and specifies the kPanelInstallSelect request code to give the extension the opportunity to set any default values in the panel. The extension's MyPanelInstall function responds to this request code.

Before the control panel removes the panel from its display, it calls the extension and specifies the kPanelRemoveSelect request code. The extension's MyPanelRemove function responds to this request code. The kPanelGetTitleSelect request code is currently optional for control panel extensions. If your extension responds to this request code, it should return the name that the control panel should display for the panel in the control panel's pop-up menu. The extension's MyPanelGetTitle function responds to this request code.

MyPanelGetDITL

A control panel extension must respond to the kPanelGetDitlSelect request code. A control panel sends this request code to an extension to obtain a list of the panel's items. A control panel extension typically responds to the kPanelGetDitlSelect request code by calling an extension-defined subroutine (for example, MyPanelGetDITL) to handle the request.

```
FUNCTION MyPanelGetDITL (globals: Handle; VAR ditl: Handle)
                        : ComponentResult;
```

globals — A handle to the control panel extension's global data.

ditl — On entry, a handle to a block of memory in your application heap. On exit, a handle to an item list.

DESCRIPTION

Your MyPanelGetDITL function should return, through the ditl parameter, an item list of the items supported by your extension. The control panel then places those items into the control panel and, after installing the panel, displays the panel to the user. When the control panel creates the panel, it places the items at the locations specified in the item list.

On entry to your MyPanelGetDITL function, the ditl parameter contains a handle to a block of memory in your application heap. You should resize the handle as necessary to hold the item list you return to the control panel. (If you use a Resource Manager routine such as Get1Resource, the Resource Manager automatically resizes the handle for you.)

In general, the owning control panel disposes of the handle you pass it once it's finished constructing the panel. As a result, you must make sure that the handle you pass to the control panel is not a resource handle. If you obtain your item list by reading it into memory from a resource, you should call the Resource Manager's DetachResource procedure to convert that resource handle into one that is suitable for use with the `MyPanelGetDITL` function.

The `componentFlags` field of the component description record for a control panel extension contains a bit flag, `channelFlagDontOpenResFile`, that indicates whether the control panel should open your extension's resource file before calling your extension.

Set the `channelFlagDontOpenResFile` component flag to 0 if you want the control panel to open your extension's resource file before calling your extension. Set the `channelFlagDontOpenResFile` component flag to 1 to specify that the control panel should not open your extension's resource file before calling your extension.

RESULT CODES

Your `MyPanelGetDITL` function should return `noErr` if successful, or an appropriate result code otherwise.

SEE ALSO

For an example of the `MyPanelGetDITL` function, see Listing 5-3 on page 5-14.

MyPanelInstall

A control panel extension must respond to the `kPanelInstallSelect` request code. A control panel sends this request code to an extension immediately after sending the `kPanelGetDitlSelect` request code (which initially adds your panels's items to the control panel) and just before displaying the panel to the user. A control panel extension typically responds to the `kPanelInstallSelect` request code by calling an extension-defined subroutine (for example, `MyPanelInstall`) to handle the request.

```
FUNCTION MyPanelInstall (globals: Handle; dialog: DialogPtr;
                         itemOffset: Integer): ComponentResult;
```

`globals` A handle to the control panel extension's global data.

`dialog` A pointer to the dialog record of the owning control panel. The owning control panel displays your panel's items in the dialog box referenced through this parameter.

`itemOffset`
An offset to the panel's first item.

DESCRIPTION

Your `MyPanelInstall` function should perform any processing that must occur after the panel is created but before it is displayed to the user. For example, your

`MyPanelInstall` function can set or restore default values of various items in the panel. You can also use this opportunity to create user items (such as lists) in the panel.

The `itemOffset` parameter specifies the offset from 1 to the first item in your panel. The items installed by your control panel extension are contained in a larger dialog box containing other items; as a result, if you call the `GetDialogItem` procedure to obtain a handle to an item, you need to increment the `itemNo` parameter passed to `GetDialogItem` by the value of `itemOffset`.

In most cases, you'll need to save the value passed in the `itemOffset` parameter in your extension's global storage for later use. For example, you usually need this value to determine which panel item the user selected when your extension responds to the `kPanelItemSelect` request code.

The value passed to your `MyPanelInstall` function in the `itemOffset` parameter may be different each time `MyPanelInstall` is called. You should not assume it is always the same value.

RESULT CODES

Your `MyPanelInstall` function should return `noErr` if successful, or an appropriate result code otherwise.

SEE ALSO

For an example of the `MyPanelInstall` function, see Listing 5-4 on page 5-15.

MyPanelGetTitle

A control panel extension should respond to the `kPanelGetTitleSelect` request code but is not required to do so. A control panel sends this request code to your extension to get the name of your panel extension. A control panel extension typically responds to the `kPanelGetTitleSelect` request code by calling an extension-defined subroutine (for example, `MyPanelGetTitle`) to handle the request.

```
FUNCTION MyPanelGetTitle (self: ComponentInstance; title: Str255)
                          : ComponentResult;
```

`self` A component instance identifying the specific instance of your control panel extension.

`title` On exit, the name of your control panel extension as you want it to appear in the panel-selection pop-up menu of the control panel.

DESCRIPTION

Your MyPanelGetTitle function should return, through the title parameter, a string that is the desired title of your control panel extension. This name appears as a menu item in the pop-up menu that lets the user select which panel to view.

SPECIAL CONSIDERATIONS

Currently, all control panels use the component name as the title of the control panel extension. The MyPanelGetTitle function is intended to allow your extension to assign a title different from the component name. Future control panels are likely to call your MyPanelGetTitle function.

RESULT CODES

Your MyPanelGetTitle function should return noErr if successful, or an appropriate result code otherwise.

MyPanelRemove

A control panel extension must respond to the kPanelRemoveSelect request code. A control panel sends this request code to an extension just before removing the panel from the enclosing dialog box. A control panel extension typically responds to the kPanelRemoveSelect request code by calling an extension-defined subroutine (for example, MyPanelRemove) to handle the request.

```
FUNCTION MyPanelRemove (globals: Handle; dialog: DialogPtr;
                        itemOffset: Integer): ComponentResult;
```

globals — A handle to the control panel extension's global data.

dialog — A pointer to the dialog record of the owning control panel.

itemOffset — An offset to the panel's first item.

DESCRIPTION

Your MyPanelRemove function should perform any processing that must occur before your panel is removed from the enclosing dialog box. For example, your MyPanelRemove function can save the current values of any items in the dialog box. You can also use this opportunity to dispose of any user items (such as lists) in the dialog box. If the control panel opened your component's resource file, that file is still open at the time MyPanelRemove is called.

The itemOffset parameter specifies the offset from 1 to the first item in your control panel. The dialog items installed by your control panel extension are contained in a larger dialog box containing other items; as a result, if you call the GetDialogItem procedure to obtain a handle to a dialog item, you need to increment the `itemNo` parameter passed to `GetDialogItem` by the value of `itemOffset`.

The value passed to your `MyPanelRemove` function in the `itemOffset` parameter may be different each time `MyPanelRemove` is called. You should not assume it is always the same value.

RESULT CODES

Your `MyPanelRemove` function should return `noErr` if successful, or an appropriate result code otherwise.

Handling Panel Events

A control panel extension should respond to the `kPanelItemSelect` and `kPanelEventSelect` request codes. You typically define subroutines that the main program of your control panel extension calls (using the `CallComponentFunctionWithStorage` function) to handle these requests. You can choose any name for these subroutines, but by convention they're called `MyPanelItem` and `MyPanelEvent`. These two routines should respond to mouse clicks and other events in the items of the panel.

MyPanelItem

A control panel extension must respond to the `kPanelItemSelect` request code. In general, a control panel sends this request code to your extension whenever the user clicks an item in your panel. A control panel extension typically responds to the `kPanelItemSelect` request code by calling an extension-defined subroutine (for example, `MyPanelItem`) to handle the request.

```
FUNCTION MyPanelItem (globals: Handle; dialog: DialogPtr;
                      itemOffset: Integer; itemNum: Integer)
                      : ComponentResult;
```

`globals` A handle to the control panel extension's global data.

`dialog` A pointer to the dialog record of the owning control panel. The owning control panel displays your panel's items in the dialog box (of the control panel) referenced through this parameter.

`itemOffset`
An offset to the panel's first item.

`itemNum` The item number of the item selected by the user. This item number is an index into the list of items in the dialog box. To map this value to the item list you passed to the control panel (in the `MyPanelGetDITL` function), you need to compensate for the offset reported in the `itemOffset` parameter.

DESCRIPTION

Your `MyPanelItem` function should handle mouse clicks on specific items in your panel. The owning control panel calls your control panel extension with the `kPanelItemSelect` whenever your component returns `FALSE` in response to an event-select request. Your `MyPanelItem` function is therefore typically invoked each time the user clicks on some item in your panel. Your function should respond appropriately, according to the item that was clicked.

As just described, note that when a click in one of your panel's items occurs, the owning control panel first sends your component an event-select request, giving your component a chance to filter the event, if necessary. In this case, if your component returns `FALSE` in the `handled` parameter, then the control panel sends your component the item-select request code; if your component returns `TRUE` in the `handled` parameter, the control panel does not send your component the subsequent item-select request code.

RESULT CODES

Your `MyPanelItem` function should return `noErr` if successful, or an appropriate result code otherwise.

SEE ALSO

For an example of the `MyPanelItem` function, see Listing 5-5 on page 5-16. For information on responding to events, see the description of the `MyPanelEvent` function in the next section.

MyPanelEvent

A control panel extension must respond to the `kPanelEventSelect` request code. A control panel sends this request code to your extension whenever an event occurs in your panel. A control panel extension typically responds to the `kPanelEventSelect` request code by calling an extension-defined subroutine (for example, `MyPanelEvent`) to handle the request.

```
FUNCTION MyPanelEvent (globals: Handle; dialog: DialogPtr;
                      itemOffset: Integer;
                      theEvent: eventRecord;
                      VAR itemHit: Integer;
                      VAR handled: Boolean): ComponentResult;
```

`globals` A handle to the control panel extension's global data.

`dialog` A pointer to the dialog record of the owning control panel. The owning control panel displays your items in the dialog box (of the control panel) referenced through this parameter.

`itemOffset`
An offset to the panel's first item.

`theEvent` An event record describing the event being reported to your control panel extension.

`itemHit` On entry, the item number of an item. This number is valid only for mouse events (on input, do not interpret this parameter for any other type of event). On exit, if the `MyPanelEvent` function has handled the event, it should return the item number of the associated item in this parameter.

`handled` On entry, the value `FALSE` for mouse events; the value `TRUE` for all other events. On exit, the `MyPanelEvent` function should return a Boolean value that indicates whether it has handled the event (`TRUE`) or has not handled the event (`FALSE`).

DESCRIPTION

Your `MyPanelEvent` function is called whenever an event occurs in your panel. The parameter `theEvent` contains a complete description of the event. A control panel handles events in its own items and also gives your component a chance to handle events in its own panel.

The `MyPanelEvent` function is intended to operate just like an event filter function specified in calls to the `ModalDialog` procedure or other Dialog Manager routines. The main difference between `MyPanelEvent` and other event filter functions is that `MyPanelEvent` does not return a Boolean value as its function result. Instead, it indicates whether it handled the event in the `handled` parameter.

If the specified event is a mouse event, you might prefer your extension's `MyPanelItem` function to handle the event. In that case, you should return `FALSE` in the `handled` parameter. Otherwise, you should attempt to handle the event.

If your `MyPanelEvent` function does handle the event, it should update the `itemHit` parameter to reflect the affected item and return `TRUE` in the `handled` parameter. If you set `handled` to `FALSE`, the owning control panel sends your panel an item-select request.

RESULT CODES

Your `MyPanelEvent` function should return `noErr` if successful, or an appropriate result code otherwise.

SEE ALSO

For an example `MyPanelEvent` function, see Listing 5-6 on page 5-18. See the description of `MyPanelItem` on page 5-25 for information on handling clicks in dialog items. For a description of the fields of the event record, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

Managing Panel Settings

A control panel extension should respond to the `kPanelValidateInputSelect`, `kPanelGetSettingsSelect`, and `kPanelSetSettingsSelect` request codes. You typically define subroutines that the main program of your control panel extension calls (using the routine `CallComponentFunctionWithStorage`) to handle these requests. You can choose any name for these subroutines, but by convention they're called `MyPanelValidateInput`, `MyPanelGetSettings`, and `MyPanelSetSettings`. These routines should manage item settings in a panel.

Note

Current versions of the Sound and Video control panels do not send the `kPanelValidateInputSelect`, `kPanelGetSettingsSelect`, or `kPanelSetSettingsSelect` request code. u

MyPanelValidateInput

A control panel extension must respond to the `kPanelValidateInputSelect` request code. A control panel sends this request code to your extension whenever the user clicks a control panel's close box. A control panel extension typically responds to the `kPanelValidateInputSelect` request code by calling an extension-defined subroutine (for example, `MyPanelValidateInput`) to handle the request.

```
FUNCTION MyPanelValidateInput (globals: Handle; VAR ok: Boolean)
                                  : ComponentResult;
```

`globals` A handle to the control panel extension's global data.

`ok` On return, a Boolean value that indicates whether the panel's current values are valid (`TRUE`) or invalid (`FALSE`).

DESCRIPTION

Your `MyPanelValidateInput` function should perform any processing necessary to validate the current settings in the panel. For example, if your panel contains any editable text items, you might need to ensure that the text they contain makes sense. The control panel calls this function when the user clicks the control panel's close box.

If the current settings of the panel items are acceptable, set the `ok` parameter to `TRUE` before returning from `MyPanelValidateInput`. If the current settings are not valid, set `ok` to `FALSE`. When you set `ok` to `FALSE`, the control panel ignores any of the user's subsequent clicks in the panel's OK button.

RESULT CODES

Your `MyPanelValidateInput` function should return `noErr` if successful, or an appropriate result code otherwise.

MyPanelGetSettings

A control panel extension must respond to the `kPanelGetSettingsSelect` request code. A control panel sends this request code to your extension to get the panel's current settings. A control panel extension typically responds to the `kPanelGetSettingsSelect` request code by calling an extension-defined subroutine (for example, `MyPanelGetSettings`) to handle the request.

```
FUNCTION MyPanelGetSettings (globals: Handle; VAR ud: UserData;
                             flags: LongInt): ComponentResult;
```

`globals` A handle to the control panel extension's global data.

`ud` A handle to the control panel's configuration data.

`flags` Reserved. This parameter is always 0.

DESCRIPTION

Your `MyPanelGetSettings` function should return, through the `ud` parameter, a copy of the panel's current settings. This copy is maintained privately by the control panel. The control panel may subsequently restore your panel's settings by passing those settings to your `MyPanelSetSettings` function.

Your control panel extension is responsible for allocating storage for the configuration data to which `ud` is a handle. You might do that when the Component Manager passes your extension the `kComponentOpenSelect` parameter. Your extension should not dispose of that storage until it closes (that is, when the Component Manager passes it the `kComponentCloseSelect` parameter).

You can arrange the panel configuration data in any way you like. The data needs to contain whatever information is necessary for your `MyPanelSetSetting` function to set all relevant panel items to specified values. For example, the standard Apple sound panels save information such as the component type of the default sound output device, the current volumes levels, the current alert beep, and so forth. You might want to begin the configuration data with a version number so that you can easily change the format of the rest of the data, if necessary.

The information you return to the control panel may get stored as part of the owner's configuration information and might therefore persist across system restarts. As a result, you should not store values that might change without the control panel's knowledge (such as component ID numbers, file reference numbers, and similar volatile information).

RESULT CODES

Your `MyPanelGetSettings` function should return `noErr` if successful, or an appropriate result code otherwise.

MyPanelSetSettings

A control panel extension must respond to the kPanelSetSettingsSelect request code. A control panel sends this request code to your extension to request that your extension set the panel's current settings to the specified values. A control panel extension typically responds to the kPanelSetSettingsSelect request code by calling an extension-defined subroutine (for example, MyPanelSetSettings) to handle the request.

```
FUNCTION MyPanelSetSettings (globals: Handle; ud: UserData;
                             flags: LongInt): ComponentResult;
```

globals A handle to the control panel extension's global data.

ud A handle to the control panel's configuration data.

flags Reserved. This parameter is always 0.

DESCRIPTION

Your MyPanelSetSettings function should parse the block of configuration data passed in the ud parameter and set the values of the items in the panel based on that data. The control panel calls this function just before your panel is displayed to the user and whenever a user cancels changes to your panel. You can assume that the data passed in the ud parameter was created by a previous call to your extension's MyPanelGetSetting function.

It's possible that your extension might not able to set the value of one or more panel items to the values specified in the configuration data. (For example, the hardware environment might have changed since the configuration data was last stored by the control panel.) When this happens, you should try to match the specified panel settings as closely as possible. If you cannot match perfectly, you should return some nonzero result code.

RESULT CODES

Your MyPanelSetSettings function should return noErr if successful, or an appropriate result code otherwise.

Summary of Control Panel Extensions

Pascal Summary

Constants

```
CONST
   {component types}
   SoundPanelType                = 'sndP';   {sound panel}
   VideoPanelType                = 'vidP';   {video panel}

   {component subtypes}
   kAlertSoundsPanel             = 'alrt';   {alert sounds panel}
   kInputsPanel                  = 'mics';   {input devices panel}
   kOutputsPanel                 = 'spek';   {output devices panel}
   kVolumesSubType               = 'vols';   {volumes panel}

   {component flags}
   channelFlagDontOpenResFile    = 2;        {do not open resource file}

   {Component Manager request codes for routines}
   kPanelGetDitlSelect           = 0;        {get panel's item list}
   kPanelGetTitleSelect          = 1;        {get panel's name}
   kPanelInstallSelect           = 2;        {restore item settings}
   kPanelEventSelect             = 3;        {handle event in panel}
   kPanelItemSelect              = 4;        {handle click in a panel item}
   kPanelRemoveSelect            = 5;        {panel is about to be removed}
   kPanelValidateInputSelect     = 6;        {validate panel settings}
   kPanelGetSettingsSelect       = 7;        {get panel settings}
   kPanelSetSettingsSelect       = 8;        {set panel settings}
```

Control Panel Extension-Defined Routines

Managing Panel Components

```
FUNCTION MyPanelGetDITL      (globals: Handle; VAR ditl: Handle)
                              : ComponentResult;
```

```
FUNCTION MyPanelInstall     (globals: Handle; dialog: DialogPtr;
                             itemOffset: Integer): ComponentResult;
FUNCTION MyPanelGetTitle    (self: ComponentInstance; title: Str255)
                             : ComponentResult;
FUNCTION MyPanelRemove      (globals: Handle; dialog: DialogPtr;
                             itemOffset: Integer): ComponentResult;
```

Handling Panel Events

```
FUNCTION MyPanelItem        (globals: Handle; dialog: DialogPtr;
                             itemOffset: Integer; itemNum: Integer)
                             : ComponentResult;
FUNCTION MyPanelEvent       (globals: Handle; dialog: DialogPtr;
                             itemOffset: Integer; theEvent: eventRecord;
                             VAR itemHit: Integer; VAR handled: Boolean)
                             : ComponentResult;
```

Managing Panel Settings

```
FUNCTION MyPanelValidateInput
                            (globals: Handle; VAR ok: Boolean)
                             : ComponentResult;
FUNCTION MyPanelGetSettings (globals: Handle; VAR ud: UserData;
                             flags: LongInt): ComponentResult;
FUNCTION MyPanelSetSettings (globals: Handle; ud: UserData;
                             flags: LongInt): ComponentResult;
```

C Summary

Constants

```
/*component types*/
#define SoundPanelType          'sndP'   /*sound panel*/
#define VideoPanelType          'vidP'   /*video panel*/

/*component subtypes*/
#define kAlertSoundsPanel       'alrt'   /*alert sounds panel*/
#define kInputsPanel            'mics'   /*input devices panel*/
#define kOutputsPanel           'spek'   /*output devices panel*/
#define kVolumesSubType         'vols'   /*volumes panel*/
```

```
/*component flags*/
enum {
   channelFlagDontOpenResFile    = 2      /*do not open resource file*/
};

/*Component Manager request codes for routines*/
enum {
   kPanelGetDitlSelect           = 0,     /*get panel's item list*/
   kPanelGetTitleSelect,                  /*get panel's name*/
   kPanelInstallSelect,                   /*restore item settings*/
   kPanelEventSelect,                     /*handle event in panel*/
   kPanelItemSelect,                      /*handle click in a panel item*/
   kPanelRemoveSelect,                    /*panel is about to be removed*/
   kPanelValidateInputSelect,             /*validate panel settings*/
   kPanelGetSettingsSelect,               /*get panel settings*/
   kPanelSetSettingsSelect                /*set panel settings*/
};
```

Control Panel Extension-Defined Routines

Managing Panel Components

```
pascal ComponentResult MyPanelGetDITL
                            (Handle globals, Handle *ditl);
pascal ComponentResult MyPanelInstall
                            (Handle globals, DialogPtr dialog,
                             short itemOffset);
pascal ComponentResult MyPanelGetTitle
                            (ComponentInstance self, StringPtr title);
pascal ComponentResult MyPanelRemove
                            (Handle globals, DialogPtr dialog,
                             short itemOffset);
```

Handling Panel Events

```
pascal ComponentResult MyPanelItem
                            (Handle globals, DialogPtr dialog,
                             short itemOffset, short itemNum);
pascal ComponentResult MyPanelEvent
                            (Handle globals, DialogPtr dialog,
                             short itemOffset, eventRecord *theEvent,
                             short *itemHit, Boolean *handled);
```

Managing Panel Settings

```
pascal ComponentResult MyPanelValidateInput
                            (Handle globals, Boolean *ok);
pascal ComponentResult MyPanelGetSettings
                            (Handle globals, UserData *ud, long flags);
pascal ComponentResult MyPanelSetSettings
                            (Handle globals, UserData *ud, long flags);
```

Queue Utilities

Contents

This chapter describes how your application can directly add elements to and remove them from an operating-system queue. The Macintosh Operating System stores some of the information it uses in data structures called queues. The Queue Utilities allow you to manipulate those queues directly by adding and removing elements.

Ordinarily, you do not need to use the Queue Utilities. The Operating System itself is responsible for managing the various operating-system queues that it creates internally, and you should manipulate those queues only indirectly. For example, to add an element to the notification queue maintained by the Notification Manager, you should call the `NMInstall` function. To remove an element from that queue, you should call the `NMRemove` function. But if you discover some unusual need for adding or removing such elements directly, you can use the Queue Utilities routines. In addition, you can use the Queue Utilities routines for directly manipulating queues that you create.

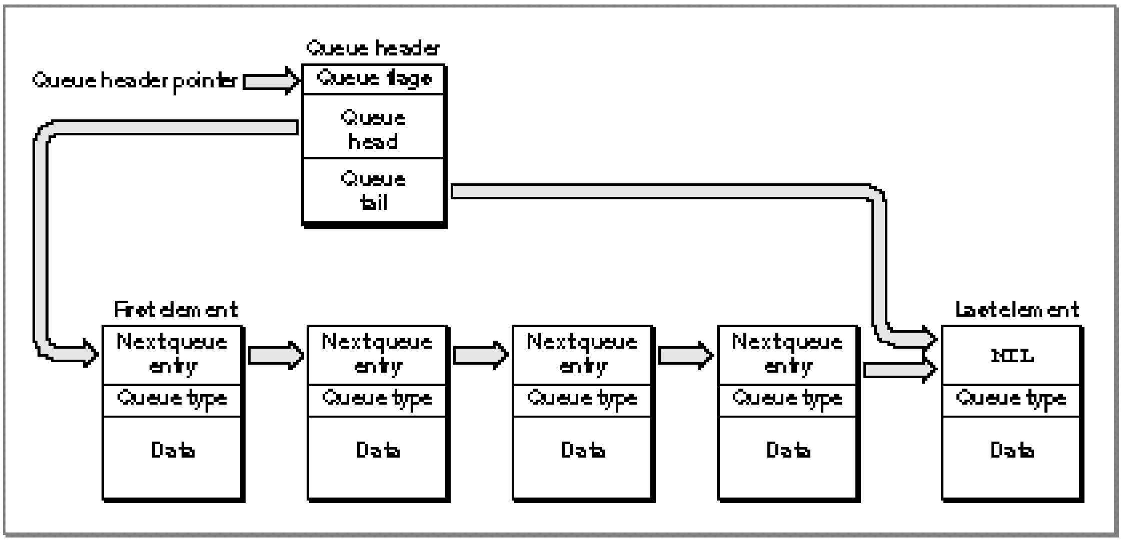
This chapter describes the general structure of operating-system queues and then

- lists the routines your application should use to manipulate an operating-system queue indirectly
- shows how your application can use the Queue Utilities for directly manipulating queues that you create.

About Queues

The Macintosh Operating System uses operating-system queues to keep track of a wide variety of items, including VBL tasks, notifications, I/O requests, events, mounted volumes, and disk drives (or other block-formatted devices). A **queue** is a list of identically structured entries linked together by pointers. A single entry in a queue is called a **queue element.** Figure 6-1 illustrates the general structure of an operating-system queue.

Figure 6-1 An operating-system queue



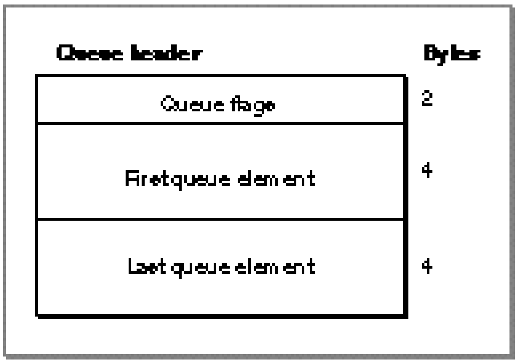
As you can see, the addresses of the first and last elements in the queue are stored in a **queue header.** The queue header also contains some queue flags, which contain information about the queue.

Each queue element contains the address of the next element in the queue (or the value `NIL` if there is no next element), an indication of the type of queue to which the next element belongs, and some data. The exact format and size of the data differs among the various queue types. In some cases, the data in the queue element contains the address of a routine to be executed. Table 6-1 on page 6-7 lists the different types of operating-system queues used by the Macintosh Operating System.

The Queue Header

The queue header is the head of a list of identically structured entries linked together by pointers. Figure 6-2 shows the format of a queue header.

Figure 6-2 The format of a queue header



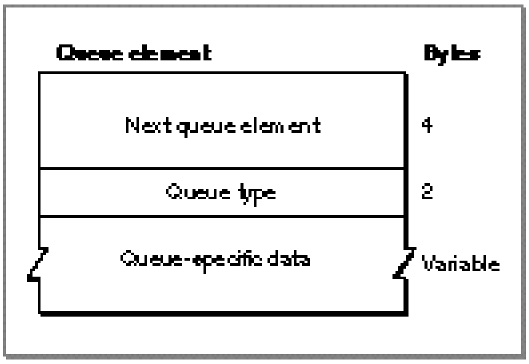
A queue header is a record defined by a data structure of type `QHdr`, which contains three fields: flags, a pointer to the first element in the queue (`qHead`), and a pointer to the last element in the queue (`qTail`). The flags field contains information specific to each queue. Ordinarily, these flags are for use by the system software only, and your application should not need to read or manipulate these flags. The `qHead` field is a pointer to the first element in a queue, and the `qTail` field is a pointer to the last element in a queue. If the queue has no elements, both of these fields are set to `NIL.` Thus, if you have access to a variable `myQueueHdr` of type `QHdrPtr`, you can access the corresponding first queue element of a non-empty queue with myQueueHdr^.qHead^ and access the last element with myQueueHdr^.qTail^.

Each queue element itself is a record of type `QElem`, which is described in the next section.

The Queue Element

The exact format of a queue element is not the same for all types of operating-system queues; thus, a queue element is defined by a variant record that is a data structure of type QElem. Figure 6-3 shows the format of a queue element.

Figure 6-3 The format of a queue element



Each queue element contains two fixed fields: a pointer to the next element in the queue (qLink), a value describing the queue type (qType), and a variable data field specific to each queue type.

The qLink field contains a pointer to the next element in the queue. All queue elements are linked through these pointers. Each pointer points to the qLink field in the next queue element, and the last queue element contains a NIL pointer. The data type of the pointer to the next queue element is always QElemPtr.

The qType field contains an integer that usually designates the queue type; for example, ORD(evType) for the event queue. Table 6-1 contains a list of all the supported operating-system queue types.

Table 6-1 Operating-system queue types

Constant	Queue type	Description
`vType`	Vertical retrace queue	A list of tasks to be executed during VBL interrupts
`ioQType`	File I/O queue (or driver I/O queue)	A list of parameter blocks for all asynchronous routines awaiting execution
`drvQType`	Drive queue	A list of all disk drives connected to the computer
`evType`	Event queue	A list of pending events
`fsQType`	Volume control block queue	A list of volume control blocks for each mounted volume
`sIQType`	Slot interrupt queue	A list of slot interrupts
`dtQType`	Deferred task queue	A list of deferred tasks
`nmQType`	Notification queue	A list of notification requests
`slpQType`	Sleep queue	A list of routines to be notified before a Macintosh Portable or a PowerBook is put into the sleep state

Often, you need to set the `qType` field of a queue element to an appropriate value before installing the queue element. However, some operating-system queues use this field for different purposes. For example, the Time Manager uses an operating-system queue to track Time Manager tasks. In the high bit of this field, the revised Time Manager places a flag to indicate whether a task timer is active. The Time Manager (along with other parts of the Operating System that use this field for their own purposes) shields you from the implementation-level details of operating a queue. Indeed, there is no way for you to access a Time Manager queue directly, and the `QElem` data type does not support access of Time Manager task records from Time Manager queue elements.

The third field contains data that is specific to the type of operating-system queue to which the queue element belongs. For example, a queue element in a vertical retrace queue, maintained by the Vertical Retrace Manager, includes information about the task procedure to be called, the number of interrupts, and the task phase. A queue element in a notification queue, maintained by the Notification Manager, includes information about the alert box, the sound response, the item to be marked in the Application menu, a response procedure, and some reserved values. Figure 6-4 shows the format of these two different types of queue elements.

Figure 6-4 Formats of a vertical retrace queue element and a notification queue element

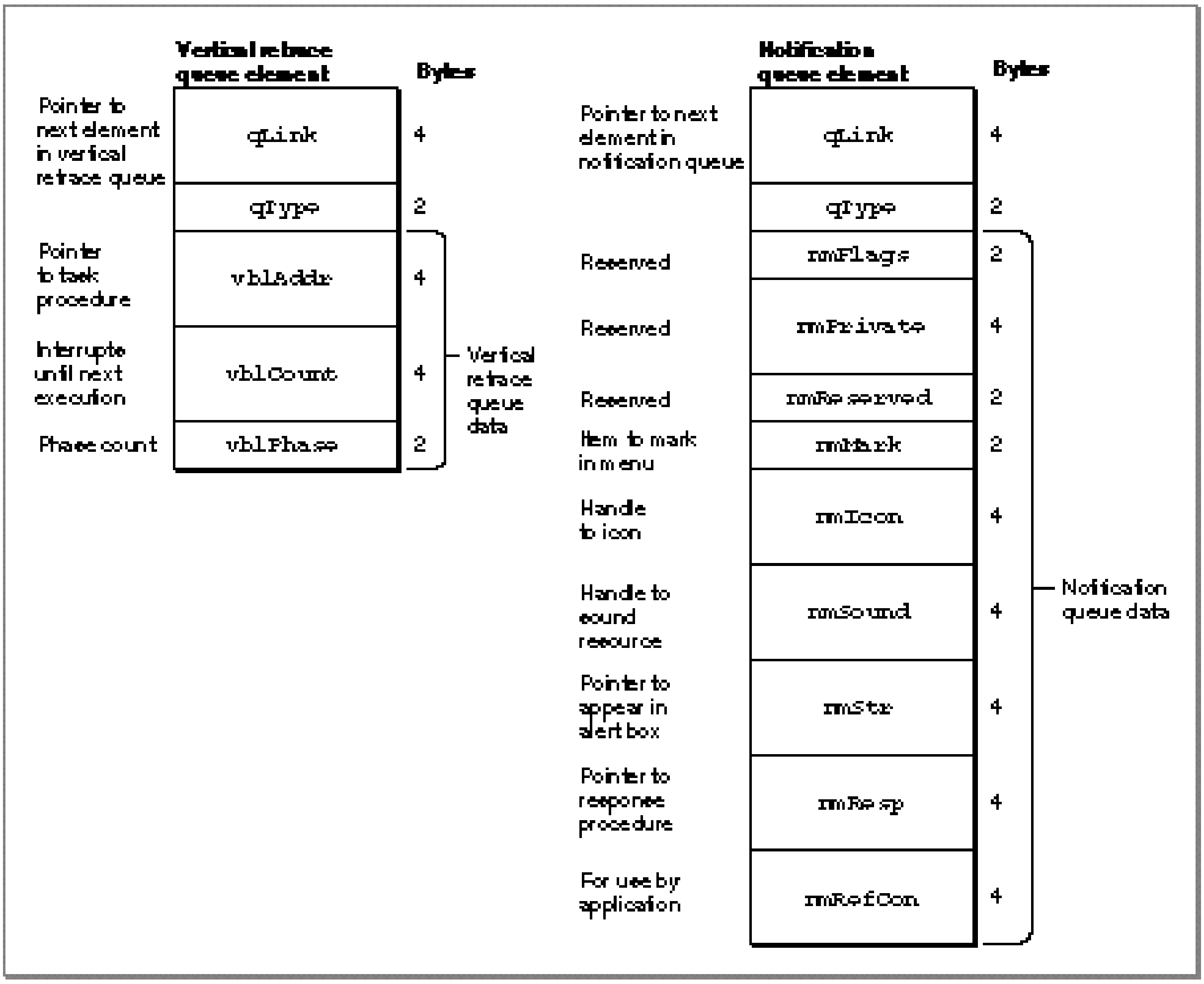


Figure 6-4 illustrates how the format and size of an operating-system queue element can vary because of the variable data field. For example, an element of type vType (a vertical retrace queue element) uses 10 bytes for VBL-specific data, whereas an element of type nmType (a notification queue element) uses 30 bytes for notification-specific data. All operating-system queue elements use at least 6 bytes: 4 bytes to store a pointer to the next element in the queue and 2 bytes to store a value indicating the queue type.

Using the Queue Utilities

The Queue Utilities provide routines for directly adding elements to a queue and removing them from a queue. The Enqueue procedure lets you add elements to the end of a queue, and the Dequeue function lets you remove elements from a queue.

You should manipulate an operating-system queue used by the Macintosh Operating System indirectly, by calling special-purpose routines. For example, to install a deferred task into a deferred task queue, your application should use the `DTInstall` function instead of the `Enqueue` procedure. However, if you create your own queues, you can use the `Enqueue` procedure and the `Dequeue` function to manipulate these queues directly. This section describes how to

- search for an element in an operating-system queue
- add an element to an operating-system queue
- remove an element from an operating-system queue

Searching for an Element in an Operating-System Queue

You can search an operating-system queue for a specific element or elements. For example, Listing 6-1 shows a simplified way to search a drive queue for all the drives connected to the computer. The application-defined function, `MySearchDriveQueue`, walks through the drive queue searches for all connected drives. If it finds any, it calls the application-defined function `DoDisplayDriveInfo` to display information about the connected drive.

Listing 6-1 Searching for drives in the drive queue

```
FUNCTION MySearchDriveQueue: Boolean;
VAR
   driveQHdr:     QHdrPtr;
   result:        Boolean;
BEGIN
   result := FALSE;                          {assume no drivers in the queue}
   driveQHdr := GetDrvQHdr;                  {get the drive queue header}
   driveQPtr := DrvQElPtr(driveQHdr^.qHead);
   WHILE (driveQPtr <> NIL) DO               {while drive queue is not empty}
   BEGIN
      result := TRUE;                        {found a drive}
      DoDisplayDriveInfo(driveQPtr);         {display drive information}
                                             {go to next drive in the queue}
      driveQPtr := DrvQElPtr(driveQPtr^.qLink);
   END; {of while}
   MySearchDriveQueue := result;             {return result of search}
END;
```

Adding Elements to an Operating-System Queue

You should avoid direct manipulation of an operating-system queue used by the Macintosh Operating System. Your application should, when possible, use the installation routines in Table 6-2 to add new elements to an operating-system queue.

Table 6-2 Installation routines for operating-system queue elements

Queue element	Installation routine	Additional information
Slot-based VBL task	`SlotVInstall`	The chapter "Vertical Retrace Manager" in *Inside Macintosh: Processes*
System-based VBL task	`VInstall`	The chapter "Vertical Retrace Manager" in *Inside Macintosh: Processes*
Parameter block for an asynchronous routine awaiting execution	*	The chapter "File Manager" in *Inside Macintosh: Files*
Disk drive	`AddDrive`	The chapter "File Manager" in *Inside Macintosh: Files*
Event	`PPostEvent` and `PostEvent`	The chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*
Volume control block	*	The chapter "File Manager" in *Inside Macintosh: Files*
Deferred task	`DTInstall`	The chapter "Deferred Task Manager" in *Inside Macintosh: Processes*
Slot interrupt	`SIntInstall`	The chapter "Slot Manager" in *Inside Macintosh: Devices*
Notification request	`NMInstall`	The chapter "Notification Manager" in *Inside Macintosh: Processes*
Sleep	`SleepQInstall`	The chapter "Power Manager" in *Inside Macintosh: Devices*

* No comparative installation routine available.

IMPORTANT

It is not recommended that you directly add elements to an operating-system queue used by the Macintosh Operating System. If at all possible, your application should use the installation routines provided by the various managers. s

If you have created a queue for your own use, you can use the `Enqueue` procedure to add a new element to your queue. For example, Listing 6-2 presents the application-defined procedure `DoAddBankCustomer`, which uses the `Enqueue` procedure for directly installing a customer into a bank-teller queue.

Listing 6-2 Using the `Enqueue` procedure to add a bank customer to a teller queue

```
PROCEDURE DoAddBankCustomer(myQueueHdrPtr: QHdrPtr,
                            Var bankCustomer: MyCustomerRecord);
BEGIN
   WITH bankCustomer^ DO               {get bank customer data}
   BEGIN
      qType := kTellerQType;           {queue type for the bank-teller queue}
      account := MyGetNextAccount;     {get account number}
      action := MyGetBankAction;       {get action to perform}
      amount := MyGetAmount;           {get the amount}
   END;
   Enqueue(QElemPtr(bankCustomer), myQueueHdrPtr);    {add customer to queue}
END;
```

Note that you are responsible for allocating memory for a queue element before you insert into a queue and for deallocating that memory when you remove the queue element.

Removing Elements From an Operating-System Queue

This section describes how your application can remove elements from an operating-system queue. Whenever possible, your application should use the removal routines listed in Table 6-3 to remove elements indirectly from an operating-system queue used by the Macintosh Operating System.

Table 6-3 Removal routines for operating-system elements

Queue element	Removal routine	Additional information
Slot-based VBL task	`SlotVRemove`	The chapter "Vertical Retrace Manager" in *Inside Macintosh: Processes*
System-based VBL task	`VRemove`	The chapter "Vertical Retrace Manager" in *Inside Macintosh: Processes*
Parameter block for an asynchronous routine awaiting execution	*	The chapter "File Manager" in *Inside Macintosh: Files*
Disk drive	*	The chapter "File Manager" in *Inside Macintosh: Files*
Event	`WaitNextEvent`	The chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*
Volume control block	*	The chapter "File Manager" in *Inside Macintosh: Files*
Deferred task	*	The chapter "Deferred Task Manager" in *Inside Macintosh: Processes*
Slot interrupt	`SIntRemove`	The chapter "Slot Manager" in *Inside Macintosh: Devices*
Notification request	`NMRemove`	The chapter "Notification Manager" in *Inside Macintosh: Processes*
Sleep	`SleepQRemove`	The chapter "Power Manager" in *Inside Macintosh: Devices*

* No comparative removal routine available.

IMPORTANT

It is not recommended that you directly remove queue elements from an operating-system queue used by the Macintosh Operating System. If at all possible, your application should use the removal routines provided by the various managers. s

If you have created a queue for your own use, you can use the `Dequeue` function to remove elements from that queue.

Listing 6-3 shows the application-defined function `DoRemoveBankCustomer`, which uses the `Dequeue` procedure for directly removing the first customer from a bank-teller queue. The `DoRemoveBankCustomer` function returns `TRUE` if it removes the customer.

Listing 6-3 Using `Dequeue` to remove the first customer in the bank-teller queue

```
FUNCTION DoRemoveBankCustomer (VAR myQueueHdr: QHdr): BOOLEAN;
VAR
   bankCustomerPtr: MyCustomerRecordPtr;
   customerRemoved: Boolean;
```

```
BEGIN
   customerRemoved := FALSE;
   bankCustomerPtr := MyCustomerRecordPtr(myQueueHdr.qHead);
   IF bankCustomerPtr <> NIL THEN      {Check for non-empty queue}
   BEGIN
      Dequeue(QElemPtr(bankCustomerPtr),&myQueueHdr) {remove customer}
      customerRemoved := TRUE;
   END; {of queue not empty}
   DoRemoveCustomer := customerRemoved;
END;
```

Queue Utilities Reference

This section describes the data structures of operating-system queues and two Queue Utilities routines for directly adding elements to and removing them from queues that you create.

Data Structures

Each operating-system queue created and maintained by the Macintosh Operating System consists of a queue header and a linked list of queue elements. This section describes the structure of queue headers and queue elements.

Queue Headers

A queue header is a block of data that contains information about a queue. The `QHdr` data type defines the structure of a queue header.

```
TYPE QHdr =
RECORD
   qFlags:     Integer;       {information on queue}
   qHead:      QElemPtr;      {pointer to first queue entry}
   qTail:      QElemPtr;      {pointer to last queue entry}
END;
```

Field descriptions

`qFlags` Queue flags. This field contains information that is different for each queue type. Ordinarily, these flags are reserved for use by system software.

`qHead` A pointer to the first element in the queue. If a queue has no elements, this field is set to `NIL`.

qTail — A pointer to the last element in the queue. If a queue has no elements, this field is set to NIL.

Queue Elements

A queue element is a single entry in a queue. The exact structure of an element in an operating-system queue depends on the type of the queue. The different queue types that are accessible to your application are defined by the QTypes data type.

```
TYPE QTypes =
   (dummyType,        {reserved}
   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}
   sIQType,           {slot interrupt queue type}
   dtQType,           {deferred task queue type}
   {nmType,}          {notification queue type}
   {slpQType}         {sleep queue type}
   );
```

Each of these enumerated queue types determines a different type of queue element. The QElem data type defines the available queue elements.

```
TYPE QElem =
RECORD
   CASE QTypes OF
      vType:      (vblQElem:  VBLTask);
      ioQType:    (ioQElem:   ParamBlockRec);
      drvQType:   (drvQElem:  DrvQEl);
      evType:     (evQElem:   EvQEl);
      fsQType:    (vcbQElem:  VCB);
      dtQType:    (dtQElem:   DeferredTask);
      {siQType:   (siQElem:   SlotIntQElement);}
      {nmType:    (nmQElem:   NMRec);}
      {slpQType:  (slpQElem:  SleepQRec);}
END;
QElemPtr = ^QElem;
```

Data type	Additional information
VBLTask	The chapter "Vertical Retrace Manager" in *Inside Macintosh: Processes*
ParamBlockRec	The chapter "File Manager" in *Inside Macintosh: Files*
DrvQEl	The chapter "File Manager" in *Inside Macintosh: Files*
EvQEl	The chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*
VCB	The chapter "File Manager" in *Inside Macintosh: Files*
DeferredTask	The chapter "Deferred Task Manager" in *Inside Macintosh: Processes*
SlotIntQElement	The chapter "Slot Manager" in *Inside Macintosh: Devices*
NMRec	The chapter "Notification Manager" in *Inside Macintosh: Processes*
SleepQRec	The chapter "Power Manager" in *Inside Macintosh: Devices*

Routines

The Queue Utilities provide two routines: `Enqueue` and `Dequeue`. The `Enqueue` procedure allows you to add queue elements directly to an operating-system queue, and the `Dequeue` function allows you to remove the element. Ordinarily, these routines are used only by system software. If possible, you should manipulate an operating-system queue indirectly, by calling special-purpose routines. For example, to install a task record into a slot-based vertical retrace queue, your application should use the `SlotVInstall` function (provided by the Vertical Retrace Manager) instead of the `Enqueue` procedure. In addition, you can use the Queue Utilities routines for directly manipulating queues that you create.

Enqueue

You can use the `Enqueue` procedure to add elements directly to an operating-system queue or a queue that you create.

```
PROCEDURE Enqueue (qElement: QElemPtr; qHeader: QHdrPtr);
```

`qElement` A pointer to the queue element to add to a queue.

`qHeader` A pointer to a queue header.

DESCRIPTION

The `Enqueue` procedure adds the queue element specified by `qElement` parameter to the end of the queue specified by the `qHeader` parameter. The specified queue header is updated to reflect the new queue element.

SPECIAL CONSIDERATIONS

Because interrupt routines are likely to manipulate operating-system queues, interrupts are disabled for a short time while the specified queue is updated. You can call the `Enqueue` procedure at interrupt time. Whenever possible, use the installation routines listed in Table 6-2 on page 6-10 instead of the `Enqueue` procedure.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for the `Enqueue` procedure are

Registers on entry

A0	Pointer to the queue element to be added
A1	Pointer to the queue header

Registers on exit

A1	Pointer to the queue header

SEE ALSO

For a description of the `QElem` record, see page 6-14; for a description of the `QHdr` record, see page 6-13.

Dequeue

You can use the `Dequeue` function to remove a queue element directly from an operating-system queue or from a queue that you have created.

```
FUNCTION Dequeue (qElement: QElemPtr; qHeader: QHdrPtr): OSErr;
```

`qElement` A pointer to a queue element to remove from a queue.

`qHeader` A pointer to a queue header.

DESCRIPTION

The `Dequeue` function attempts to find the queue element specified by the `qElement` parameter in the queue specified by the `qHeader` parameter. If `Dequeue` finds the element, it removes the element from the queue, adjusts the other elements in the queue accordingly, and returns `noErr`. Otherwise, it returns `qErr`, indicating that it could not find the element in the queue. The `Dequeue` function does not deallocate the memory occupied by the queue element.

SPECIAL CONSIDERATIONS

The `Dequeue` function disables interrupts as it searches through the queue for the element to be removed. The time during which interrupts are disabled depends on the length of the queue and the position of the entry in the queue. The `Dequeue` function can be called at interrupt time. Whenever possible, use the removal routines listed in Table 6-3 on page 6-12 instead the `Dequeue` function.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for the `Dequeue` function are

Registers on entry

A0	Pointer to the queue element to be removed
A1	Pointer to the queue header

Registers on exit

A1	Pointer to the queue header
D0	Result code

RESULT CODES

`noErr`	0	No error
`qErr`	–1	Entry is not in specified queue

SEE ALSO

For a description the `QElem` record, see page 6-14; for a description of the `QHdr` record, see page 6-13.

Summary of the Queue Utilities

Pascal Summary

Constants

```
CONST    {queue types}
   vType    = 1;          {vertical retrace queue type}
   ioQType  = 2;          {file I/O or driver I/O queue type}
   drvQType = 3;          {drive queue type}
   evType   = 4;          {event queue type}
   fsQType  = 5;          {volume-control-block queue type}
   sIQType  = 6;          {slot interrupt queue type}
   dtQType  = 7;          {deferred task queue type}
   nmType   = 8;          {notification queue type}
   slpQType = 16;         {sleep queue type}
```

Data Types

```
TYPE QHdr =    {queue header record}
   RECORD
      qFlags:     Integer;       {information on queue}
      qHead:      QElemPtr;      {pointer to the first queue element}
      qTail:      QElemPtr;      {pointer to the last queue element}
   END;
   QHdrPtr = ^QHdr;

   QTypes = (  {queue types}
      dummyType,                 {reserved}
      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}
      sIQType,                   {slot interrupt queue type}
      dtQType,                   {deferred task queue type}
```

```
      {nmType,}                  {notification queue type}
      {slpQType}                 {sleep queue type}
   );

   QElem =     {queue element record}
   RECORD
      CASE QTypes OF
         dtQType:    (dtQElem:   DeferredTask);    {deferred task }
                                                   { queue element}
         vType:      (vblQElem:  VBLTask);         {vertical retrace }
                                                   { queue element}
         ioQType:    (ioQElem:   ParamBlockRec);   {file I/O queue element}
         drvQType:   (drvQElem:  DrvQEl);          {drive queue element}
         evType:     (evQElem:   EvQEl);           {event queue element}
         fsQType:    (vcbQElem:  VCB);             {volume-control-block }
                                                   { queue element}
         {sIQType:   (siQElem:   SlotIntQElement;} {slot interupt }
                                                   { queue element}
         {nmType:    (nmQElem:   NMRec);}          {notification }
                                                   { queue element}
         {slpQType:  (slpQElem:  SleepQRec);}      {sleep queue element}
   END;
   QElemPtr = ^QElem;
```

Routines

```
PROCEDURE Enqueue            (qElement: QElemPtr; qHeader: QHdrPtr);
FUNCTION Dequeue             (qElement: QElemPtr; qHeader: QHdrPtr): OSErr;
```

C Summary

Constants

```
enum {      /*queue types*/
   vType    = 1,              /*vertical retrace queue type*/
   ioQType  = 2,              /*file I/O or driver I/O queue type*/
   drvQType = 3,              /*drive queue type*/
   evType   = 4,              /*event queue type*/
   fsQType  = 5,              /*volume-control-block queue type*/
```

```
   sIQType  = 6,              /*slot interrupt queue type*/
   dtQType  = 7,              /*deferred task queue type*/
};

enum {      /*value for the notification queue type*/
   nmType   = 8               /*notification queue type*/
};

enum {      /*value for the sleep queue type*/
   slpQType = 16              /*sleep queue type*/
};
```

Data Types

```
struct QHdr {  /*queue header record*/
   short          qFlags;     /*information on queue*/
   QElemPtr       qHead;      /*pointer to the first queue element*/
   QElemPtr       qTail;      /*pointer to the last queue element*/
};
typedef struct QHdr QHdr;
typedef QHdr *QHdrPtr;

typedef unsigned short QTypes;   /*queue types*/

struct QElem { /*queue element record*/
   struct QElem   *qLink;     /*pointer to the next queue element*/
   short          qType;      /*type of queue element*/
   short          qData[1];   /*variable array of data; type of data and */
                              /* length depend on the queue type, */
                              /* specified in the qType field*/
};
typedef struct QElem QElem;
typedef QElem *QElemPtr;
```

Routines

```
pascal void Enqueue         (QElemPtr qElement, QHdrPtr qHeader);
pascal OSErr Dequeue        (QElemPtr qElement, QHdrPtr qHeader);
```

Assembly-Language Summary

QHdr Data Structure

0	qFlags	word	information on queue
2	qHead	long	pointer to first queue entry
6	qTail	long	pointer to last queue entry

QElem Data Structure

0	qLink	long	pointer to the next queue element
4	qType	word	type of queue element
6	qData	word	variable array of data; type of data and length depend on the queue type, specified in the qType field

Result Codes

noErr	0	No error
qErr	–1	Entry is not in specified queue

Parameter RAM Utilities

Contents

This chapter describes how your application can access and modify the information used by the system software at system startup time. Various user settings, such as the volume setting for the built-in speaker, need to be present at the next system startup. This startup information is stored in battery-powered parameter RAM, located in the computer's real-time clock chip. The Parameter RAM Utilities available in the Macintosh Operating System allow you to manipulate startup information stored in parameter RAM.

Because you can use Toolbox routines to indirectly access most of the useful information stored in parameter RAM, you should not need to use the utility routines described in this chapter. However, if you should discover some important need to directly manipulate the startup information in parameter RAM, you can use the Parameter RAM Utilities routines.

To use this chapter, you should already understand how to read and change the values of low-memory global variables. See the chapter "Memory Manager" in *Inside Macintosh: Memory* for a discussion on how to read and write system global variables.

This chapter

- introduces the kinds of information stored in parameter RAM
- describes some of the values stored in parameter RAM

About Parameter RAM

Most user settings that need to be present at system startup are stored in **parameter RAM**. Parameter RAM takes up 256 bytes of battery-powered RAM: 20 bytes are documented in this chapter, and 236 bytes are reserved by the system software. The 236 bytes of parameter RAM are also known as **extended parameter RAM**. The parameter RAM is located in the computer's real-time clock chip, together with the date and time setting. No matter what system disk is used at system startup, parameter RAM ensures that certain settings remain the same on a given computer from one session to another.

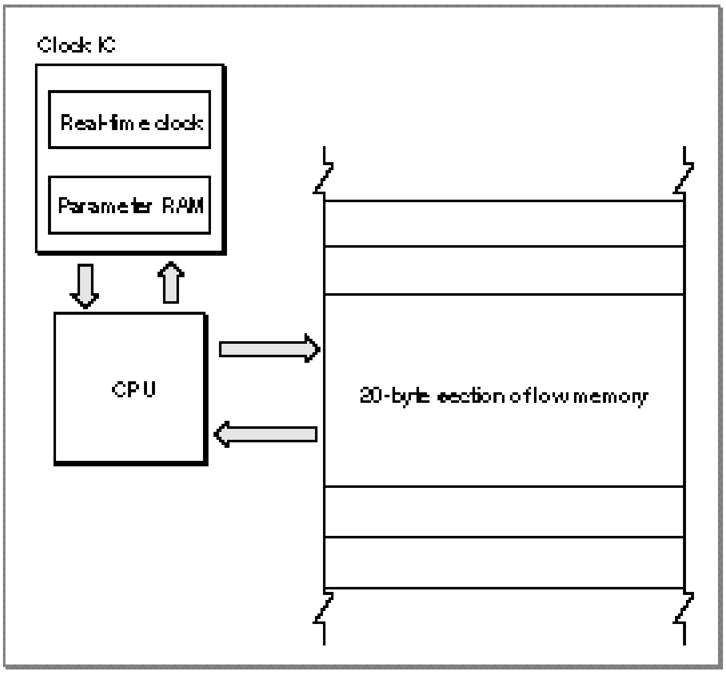
Much of the information stored in parameter RAM is used exclusively by the system software. For example, system software uses 2 bits of parameter RAM to keep track of how many times menu items should blink after being selected. Other values stored in parameter RAM are useful to applications. For example, parameter RAM stores the suggested time interval that your application should use when determining whether two mouse clicks constitute a double-click. You can access this double-click time indirectly by using the Toolbox Event Manager's `GetDblTime` function. Whenever possible, you should use Toolbox routines to access parameter RAM values.

s **WARNING**

The operating-system routines described in this chapter let you directly manipulate values in parameter RAM; however, because the organization of parameter RAM is subject to change, you should rarely use them. Instead, use the appropriate Toolbox routines to *indirectly* manipulate values in parameter RAM. s

The 20 bytes of parameter RAM that are commonly accessible by applications are copied into low memory at system startup. Figure 7-1 illustrates the interaction between parameter RAM and low memory. Parameter RAM is read into low memory at system startup, and any modifications to this low-memory copy of parameter RAM are written back to the clock chip.

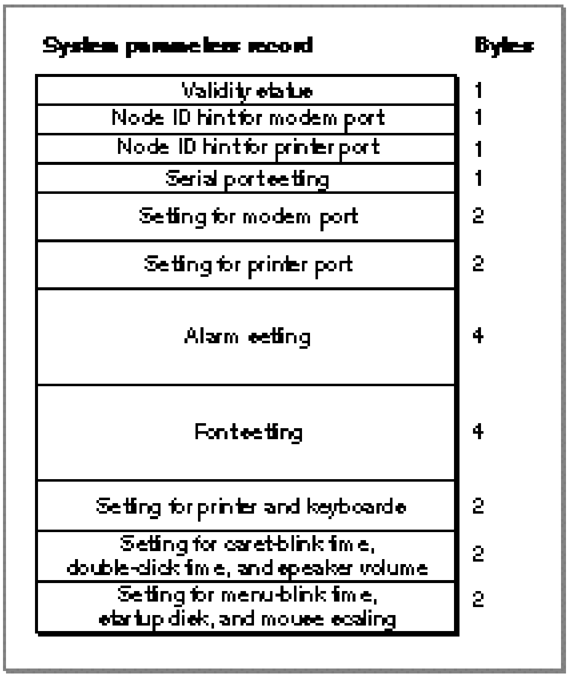
Figure 7-1 Interaction between parameter RAM and low memory



The 20 accessible bytes of parameter RAM are described by the **system parameters record**, which is defined by a data structure of type `SysParmType`.

Figure 7-2 shows the general structure of the system parameters record, which contains 11 fields.

Figure 7-2 The format of the system parameter record



A system parameters record contains 11 fields. See page 7-9 for the exact structure of each field.

The first field of the system parameters record contains information about the validity status of the clock chip. Whenever a write to the clock chip is successful, the value $A8 is stored in this field. The status is examined when the clock chip is read at system startup.

The second and third fields contain information about the node ID for the modem port and printer port.

The fourth field tells which device or devices may use each of the serial ports.

The fifth field contains the baud rate, data bits, stop bits, and parity for the modem port. Bits 0–9 define the baud rate; bits 10 and 11 define the number of data bits; bits 12 and 13 define the parity; and bits 14 and 15 define the number of stop bits.

The sixth field contains the baud rate, data bits, stop bits, and parity for the printer port. As with the modem port, bits 0–9 define the baud rate; bits 10 and 11 define the number of data bits; bits 12 and 13 define the parity; and bits 14 and 15 define the number of stop bits.

The seventh field contains the time at which the alarm clock should sound. The time is defined in terms of seconds since midnight, January 1, 1904.

The eighth field contains the default application font number minus 1.

The ninth field contains the settings for the printer and for the keyboard. Bit 0 designates whether the currently chosen printer (if any) is connected to the printer port (0) or the modem port (1). Bits 1–7 are reserved for future use. Bits 8–11 of this field contain the **auto-key rate**, the rate at which a character key repeats when it's held down; this value is stored in 2-tick units. Bits 12–15 contain the **auto-key threshold**, the length of time a key must be held down before it begins to repeat; this value is stored in 4-tick units.

The tenth field contains miscellaneous user settings. Bits 0–3 contain the caret-blink time, and bits 4–7 contain the double-click time; both values are stored in four-tick units. The **caret-blink time** is the interval between blinks of a caret that marks the insertion point in text. 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 8–10 contain the speaker volume setting, which ranges from silent (0) to loud (7).

The last field contains more miscellaneous user settings. Bits 2 and 3 contain a value from 0 to 3 designating the **menu-blink time**, which is how many times a menu item blinks when the user chooses it. Because system software automatically calls both standard and nonstandard menu definition procedures the appropriate number of times, you should not need to worry about that value in parameter RAM. Bit 4 indicates whether the preferred system startup disk is in an internal (0) or external (1) drive. If there is any problem using the disk in the specified drive, the other drive is used. Bit 6 designates whether mouse scaling is on (1) or off (0). If mouse scaling is on, cursor movement doubles if the user moves the mouse more than a certain number of pixels between vertical retrace interrupts.

The global variable `SysParam` contains the address of the start of the system parameters record. Other global variables allow you to access individual fields of the system parameters record directly. These global variables all begin with the letters `SP` and point directly into the system parameters record stored in low memory. Other global variables referencing memory locations outside of the system parameters record are used to store copies of individual fields of the system parameters record.

s **WARNING**

The default values for parameter RAM vary depending on the version of the system software. Therefore, do not rely on any one default value being the same for all machines. s

Though default values can vary, most of the U.S. system software "shares" default values. The default values for parameter RAM, for U.S. system software, are shown in Table 7-1.

Table 7-1 Default values for parameter RAM (for U.S. system software)

Description	Default value
Validity status	$A8
Node ID hint for modem port	0
Node ID hint for printer port	0
Serial port use	0 (both ports)
Modem port configuration	9600 baud, 8 data bits, no parity, 2 stop bits
Printer port configuration	9600 baud, 8 data bits, no parity, 2 stop bits
Alarm setting	0 (midnight, January 1, 1904)
Application font minus 1	2 (indicating Geneva)
Auto-key threshold	6 (24 ticks)
Auto-key rate	3 (6 ticks)
Printer connection	0 (printer port)
Caret-blink time	8 (32 ticks)
Double-click time	8 (32 ticks)
Speaker volume	3 (medium)
Menu-blink time	3
Preferred system start-up disk	0 (internal drive)
Mouse scaling	1 (on)

In System 7, a user can clear the current settings in the parameter RAM and restore the default values by holding down the ⌘-Option-P-R keys at system startup. When system software detects this key combination, it resets parameter RAM to the default values and then restarts the computer again. Clearing the current settings in the parameter RAM also causes system software to change other settings not stored in parameter RAM to default values. These settings include the desktop pattern and the color depth of the default monitor.

Using the Parameter RAM Utilities

The Parameter RAM Utilities provide two functions—`GetSysPPtr` and `WriteParam`—that allow you to directly manipulate parameter RAM. The `GetSysPPtr` function lets you access the low-memory copy of the parameter RAM, and the `WriteParam` function lets you write the modified low-memory copy back to parameter RAM. A third function, `InitUtil`, is used by the system software only. At system startup, this function reads the values from parameter RAM into low memory.

You may find it necessary to read the values in parameter RAM or even change them. You read from and write to parameter RAM using the `GetSysPPtr` and `WriteParam` functions.

Many of the values held in parameter RAM are also copied at system startup into other low-memory locations. Therefore, to change a value in parameter RAM, you must change all low-memory copies representing the value before you call `WriteParam` to write the values back to the clock chip. For example, the global variable `SPVolCtl` points to the location within the system parameters record that stores the speaker volume, and the global variable `SdVolume` references a copy of this information stored elsewhere in low memory. You could change one without changing the other, although ordinarily you change both simultaneously.

s **WARNING**

It is not recommended that you directly manipulate parameter RAM. Your application should, if at all possible, use the routines provided by the Toolbox to read the information stored in parameter RAM. s

The global variable `SysParam` points to the beginning of the system parameters record stored in low memory. You can access the system parameters record directly by using this global variable, or you can use the `GetSysPPtr` routine to return a pointer to the system parameters record. Thus, you can access the low-memory system parameters record like this:

```
WITH GetSysPPtr^ DO
BEGIN
   ...   {access the system parameters record directly here}
END;
```

IMPORTANT

Though system software automatically copies parameter RAM into low memory at startup, it does not automatically do the reverse. Therefore, after you make a change to the information in the low-memory system parameters record, you must use the `WriteParam` function to copy values from that record back to the clock chip to make the change permanent. s

At startup, system software calls the `InitUtil` function (which you should never need to call yourself) to copy the values stored in parameter RAM into low memory. (It then copies those values into other appropriate global variables.) When you make changes to the low-memory copy of parameter RAM, you must call the `WriteParam` function to record your changes in the clock chip.

Parameter RAM Utilities Reference

This section describes the data structure and routines that are specific to the Parameter RAM Utilities. The section “Data Structures” shows the Pascal data structure for the system parameters record. The section “Routines” describes the routines that are used to access and manipulate the startup information stored in parameter RAM.

Data Structures

This section describes the systems parameter record, which contains the current settings for startup information stored in parameter RAM. For information about parameter RAM default values, see Table 7-1 on page 7-7.

The System Parameters Record

The `SysParmType` data type describes a system parameters record.

```
TYPE SysParmType =
PACKED RECORD
   valid:      Byte;    {validity status}
   aTalkA:     Byte;    {node ID hint for modem port}
   aTalkB:     Byte;    {node ID hint for printer port}
   config:     Byte;    {use types for serial ports}
   portA:      Integer; {modem port configuration}
   portB:      Integer; {printer port configuration}
   alarm:      LongInt; {alarm setting}
   font:       Integer; {application font number minus 1}
   kbdPrint:   Integer; {printer connection, auto-key settings}
   volClik:    Integer; {caret blink, double click, speaker vol.}
   misc:       Integer; {menu blink, startup disk, mouse scaling }
END;
SysPPtr = ^SysParmType;
```

Field descriptions

`valid`	Contains information about the validity status of the clock chip. Whenever a write to the clock chip is successful, the value $A8 is stored in this field. The status is examined when the clock chip is read at system startup.
`aTalkA`	Contains the node ID hint for the modem port.
`aTalkB`	Contains the node ID hint for the printer port.
`config`	Indicates which device or devices may use each of the serial ports.
`portA`	Contains the baud rate, data bits, parity, and stop bits for the modem port. Bits 0–9 define the baud rate; bits 10 and 11 define the number of data bits; bits 12 and 13 define the parity; and bits 14 and 15 define the number of stop bits.
`portB`	Contains the baud rate, data bits, parity, and stop bits for the printer port. Bits 0–9 define the baud rate; bits 10 and 11 define the number of data bits; bits 12 and 13 define the parity; and bits 14 and 15 define the number of stop bits.
`alarm`	Contains the time at which the alarm clock should sound. The time is defined in terms of seconds since midnight, January 1, 1904.
`font`	Adding 1 to this field produces the font number of the default application font.

`kbdPrint`	Contains the settings for the printer and for the keyboard. Bit 0 designates whether the currently chosen printer (if any) is connected to the printer port (0) or the modem port (1). Bits 1–7 are reserved for future use. Bits 8–11 of this field contain the auto-key rate, whose value is stored in 2-tick units. Bits 12–15 contain the auto-key threshold, whose value is stored in 4-tick units.
`volClik`	Contains miscellaneous user settings, including the caret-blink time, double-click time, and the speaker volume setting.
`misc`	Contains more miscellaneous user settings. Bits 2 and 3 contain a value from 0 to 3 designating the menu-blink time. Because system software automatically calls both standard and nonstandard menu definition procedures many times, you should not need to worry about that value in parameter RAM. Bit 4 indicates whether the preferred startup disk is in an internal (0) or external (1) drive. If there is any problem with using the disk in the specified drive, the other drive is used. Bit 6 designates whether mouse scaling is on (1) or off (0).

Routines

The Parameter RAM Utilities provide two functions for use by your application and one function for use by system software. At startup, system software uses the `InitUtil` function to read parameter RAM values into low memory. You can access the values through a system parameters record of type `SysParmType` described in the previous section. To obtain a pointer to the low-memory system parameters record, call the `GetSysPPtr` function. To copy the values in the system parameters record back into the clock chip, call the `WriteParam` function.

s **WARNING**

The organization of parameter RAM is subject to change. Therefore, you should not manipulate parameter RAM values directly using the operating-system routines described in this chapter; instead, use the appropriate Toolbox routines. s

InitUtil

System software uses the `InitUtil` function at startup time to copy values from parameter RAM and date and time information into low memory. Your application should never need to use this function.

```
FUNCTION InitUtil: OSErr;
```

DESCRIPTION

The InitUtil function copies the contents of parameter RAM into 20 bytes of low memory and calls the Date, Time, and Measurement Utilities' ReadDateTime function to copy the date and time from the clock chip into a separate low-memory location.

If the validity status in parameter RAM is not $A8 when InitUtil is called, InitUtil returns a non-zero result code. In this case, the default values are read into the low-memory copy of parameter RAM; these values are then written to the clock chip.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for the InitUtil function are

Registers on exit

D0 Result code

RESULT CODES

noErr	0	No error
prInitErr	–88	Validity status not $A8

SEE ALSO

For more information about the ReadDateTime function, see the chapter "Date, Time, and Measurement Utilities" in this book.

GetSysPPtr

You can use the GetSysPPtr function to obtain a pointer to the low-memory copy of parameter RAM.

```
FUNCTION GetSysPPtr: SysPPtr;
```

DESCRIPTION

The GetSysPPtr function returns a pointer to the low-memory copy of parameter RAM. The copied parameter RAM values are accessible through the system parameters record.

You can examine the values stored in the various fields of this record, or you can change them and call the WriteParam function to copy your changes back into parameter RAM.

SPECIAL CONSIDERATIONS

Because of the organization of parameter RAM is subject to change, you should not use the `GetSysPPtr` function to change the values in parameter RAM. Instead use the appropriate Toolbox routines to modify values in parameter RAM.

ASSEMBLY-LANGUAGE INFORMATION

The global variable `SysParam` contains the address of the start of the system parameters record. Other global variables allow you to access individual fields of the system parameters record directly. These global variables all begin with the letters `SP` and point directly into the system parameters record stored in low memory. Other global variables referencing memory locations outside of the system parameters record are used to store copies of individual fields of the system parameters record.

SEE ALSO

For information about the system parameters record, see page 7-9. For a list of global variables associated with the system parameters record, see "Global Variables" on page 7-16. The `WriteParam` function is described next.

WriteParam

You can use the `WriteParam` function to write the modified values in the system parameters record to parameter RAM.

```
FUNCTION WriteParam: OSErr;
```

DESCRIPTION

The `WriteParam` function writes the modified values in the system parameters record to parameter RAM. Your application should call this function only after making changes to the system parameters record (returned by the `GetSysPPtr` function described in the previous section).

The `WriteParam` function also attempts to verify the values written by reading them back in and comparing them to the values in the low-memory copy.

SPECIAL CONSIDERATIONS

Because the organization of parameter RAM is subject to change, you should not use the `WriteParam` function to change the values in parameter RAM. Instead use the appropriate Toolbox routines to modify values in parameter RAM.

Note

If you accidentally use `WriteParam` to write incorrect values into parameter RAM, the user can clear the current settings in the parameter RAM and restore the default values by holding down the ⌘-Option-P-R keys at system startup. u

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for the `WriteParam` functions are

Registers on entry

A0	SysParam
D0	MinusOne

Registers on exit

D0	Result code

For historical reasons, you must set up register A0 with the global variable `SysParam` and register D0 with the global variable `MinusOne`. When `WriteParam` returns, register D0 contains the result code.

RESULT CODES

`noErr`	0	No error
`prWrErr`	–87	Parameter RAM written did not verify

SEE ALSO

For a description of the system parameters record, see page 7-9.

Summary of the Parameter RAM Utilities

Pascal Summary

Data Types

```
TYPE SysParmType =
   PACKED RECORD
      valid:      Byte;    {validity status}
      aTalkA:     Byte;    {node ID hint for modem port}
      aTalkB:     Byte;    {node ID hint for printer port}
      config:     Byte;    {use types for serial ports}
      portA:      Integer; {modem port configuration}
      portB:      Integer; {printer port configuration}
      alarm:      LongInt; {alarm setting}
      font:       Integer; {application font number minus 1}
      kbdPrint:   Integer; {printer connection, auto-key settings}
      volClik:    Integer; {caret blink, double click, speaker volume}
      misc:       Integer; {menu blink, startup disk, mouse scaling}
   END;

   SysPPtr        = ^SysParmType;
```

Routines

```
FUNCTION InitUtil           : OSErr;
FUNCTION GetSysPPtr         : SysPPtr;
FUNCTION WriteParam         : OSErr;
```

C Summary

Data Types

```
struct SysParmType {
      char      valid;          /*validity status*/
      char      aTalkA;         /*node ID hint for modem port*/
      char      aTalkB;         /*node ID hint for printer port*/
      char      config;         /*use types for serial ports*/
      short     portA;          /*modem port configuration*/
      short     portB;          /*printer port configuration*/
      long      alarm;          /*alarm setting*/
      short     font;           /*application font number minus 1*/
      short     kbdPrint;       /*printer connection, auto-key settings*/
      short     volClik;        /*caret blink, double click, speaker volume*/
      short     misc;           /*menu blink, startup disk, mouse scaling*/
   };

typedef struct SysParmType SysParmType;
typedef SysParmType *SysPPtr;
```

Routines

```
pascal OSErr InitUtil       (void);
SysPPtr GetSysPPtr          (void);
pascal OSErr WriteParam     (void);
```

Assembly-Language Summary

Data Structures

SysParmType Data Structure

0	valid	1 byte	validity status
1	aTalkA	1 byte	node ID hint for modem port
2	aTalkB	1 byte	node ID hint for printer port
3	config	1 byte	use types for serial ports
4	portA	word	modem port configuration
6	portB	word	printer port configuration
8	alarm	long	alarm setting
12	font	word	application font number minus 1
14	kbdPrint	word	printer connection, auto-key settings
16	volClik	word	caret blink, double click, speaker volume
18	misc	word	menu blink, system startup disk, mouse scaling

Global Variables

GetParam	System parameter scratch
SPAlarm	The alarm setting
SPATalkA	The node ID hint for modem port
SPATalkB	The node ID hint for printer port
SPClikCaret	The double-click and caret-blink times
SPConfig	The use types for serial ports
SPFont	The application font number minus 1
SPKbd	The auto-key threshold and rate
SPMisc1	Miscellaneous
SPMisc2	The setting for mouse scaling, the system startup disk, and menu-blink time
SPPortA	The modem port configuration
SPPortB	The printer port configuration
SPPrint	The printer connection
SPValid	The validity status of parameter RAM
SPVolCtl	The speaker volume
SysParam	The low-memory copy of parameter RAM

Result Codes

noErr	0	No error
prWrErr	–87	Parameter RAM written did not verify
prInitErr	–88	Validity status is not $A8

CHAPTER 8

Trap Manager

Contents

This chapter describes how your application can use the Trap Manager to augment or override an existing system software routine.

Although this chapter describes patching in some depth, you should rarely, if ever, find a need to use patches in an application. The primary purposes of patches, as their name suggests, are to fix problems and augment routines in ROM code.

To use this chapter, you should have some knowledge of assembly language. For information about the instruction sets of microprocessors in the Motorola MC680x0 family, see the appropriate user's manual, for example, the *MC68020 32-Bit Microprocessor User's Manual.*

This chapter describes how the Trap Manager works and then shows how you can use the Trap Manger to

- check for the availability of a system software routine
- alter the behavior of a system software routine

About the Trap Manager

The Trap Manager is a collection of routines that lets you add extra capabilities to system software routines.

In order to execute system software routines, system software takes advantage of the unimplemented instruction feature of the MC680x0 family of microprocessors, which are the central processing units (CPUs) used in the Macintosh family of computers.

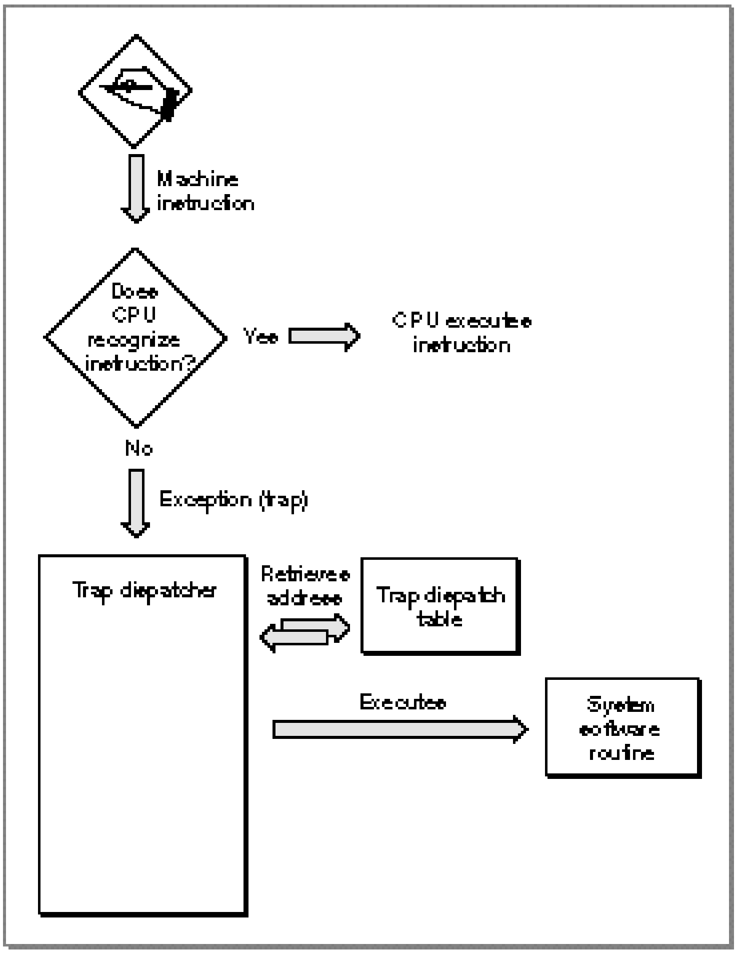
The MC680x0, like other microprocessors, executes a stream of instructions. Information encoded in an instruction indicates the operation to be performed by the microprocessor. The MC680x0 family of microprocessors recognizes a defined set of instructions. When the microprocessor encounters an instruction that it doesn't recognize, an exception is generated. An exception refers to bus errors, interrupts, and unimplemented instructions. When an exception occurs, the microprocessor suspends normal execution and transfers control to an appropriate exception handler.

In the MC680x0 family of microprocessors, all instructions starting with the hexadecimal digit $A are unimplemented instructions. These unimplemented instructions are also called **A-line instructions**. System software uses these unimplemented A-line instructions to execute system software routines. When you call a system software routine, the call to the system software routine is translated into an A-line instruction. The MC680x0 microprocessor doesn't recognize this A-line instruction, and transfers control to an exception handler.

System software provides an exception handler, called a **trap dispatcher**, to handle exceptions generated by A-line instructions. Whenever a MC680x0 microprocessor encounters an A-line instruction, an exception is generated, and the microprocessor transfers control to the trap dispatcher. An exception generated by an A-line instruction is called a **trap**.

When the trap dispatcher receives the A-line instruction, it looks into a table, called a **trap dispatch table**, to find the address of the called system software routine. After the trap dispatcher retrieves the address, it transfers control to the specified system software routine. Figure 8-1 illustrates the processing of instructions that include the A-line instructions that the microprocessor does not recognize.

Figure 8-1 How the CPU processes A-line instructions

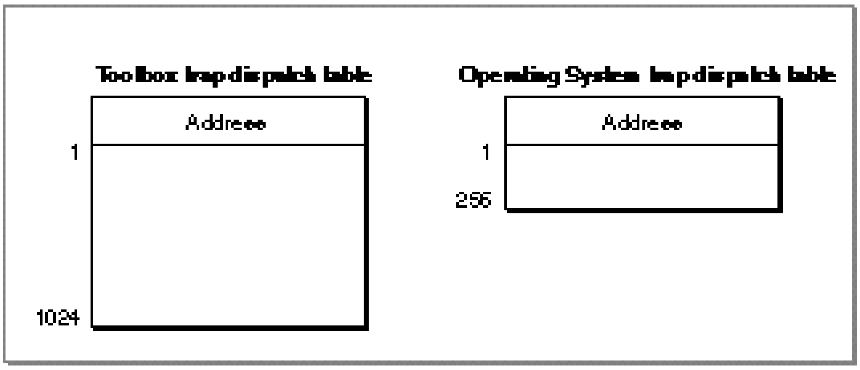


You can use the Trap Manager routines to read from and write to the two trap dispatch tables maintained by system software.

Trap Dispatch Tables

System software uses trap dispatch tables to locate the address of system software routines. System software maintains two trap dispatch tables: an Operating System trap dispatch table and a Toolbox trap dispatch table. Figure 8-2 illustrates the two trap dispatch tables.

Figure 8-2 Trap dispatch tables

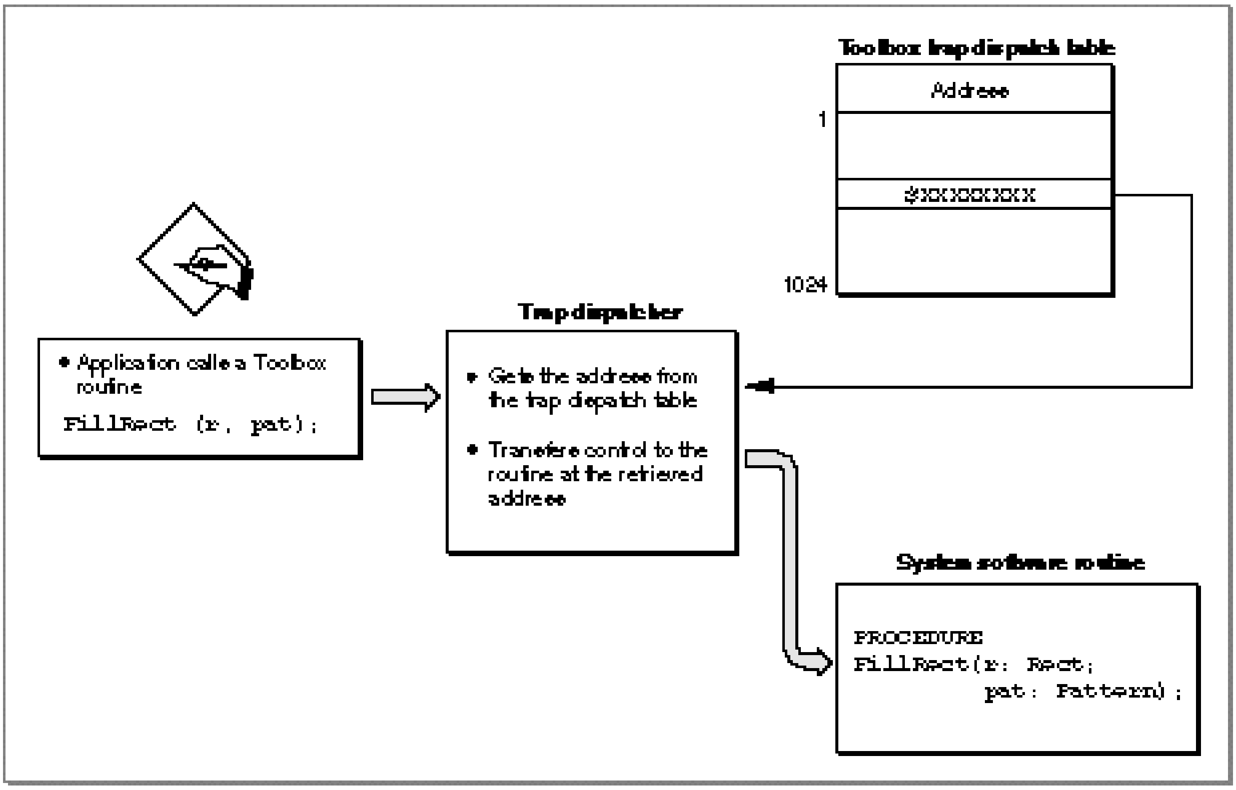


At system startup time, system software builds the trap dispatch tables and places them in RAM. The **Operating System trap dispatch table** contains 256 entries, and the **Toolbox trap dispatch table** contains 1024 entries. Each entry in the Operating System trap dispatch table contains a 32-bit address of an Operating System routine, and each entry in the Toolbox trap dispatch table contains a 32-bit address of a Toolbox routine. The system software routines can be located in either ROM or RAM.

Process for Accessing System Software Routines

As previously described, when your application calls a system software routine, an A-line instruction is sent to the microprocessor. The microprocessor does not recognize this instruction, and an exception is generated. This exception is then handled by the trap dispatcher. When the trap dispatcher receives the A-line instruction, it looks into one of the two trap dispatch tables to find the address of the called system software routine. When the trap dispatcher retrieves the address, it transfers control to the specified system software routine. For example, Figure 8-3 illustrates a call to the Toolbox procedure, `FillRect`. When the application calls the `FillRect` procedure, an exception is generated. The trap dispatcher looks into the Toolbox trap dispatch table to find the address of the `FillRect` procedure. When the address is found, the trap dispatcher transfers control to the `FillRect` procedure.

Figure 8-3 Accessing the `FillRect` procedure



Note
Not all A-line instructions are defined. When the trap dispatcher receives an undefined A-line instruction, the trap dispatcher returns the address of the Toolbox procedure `Unimplemented`. When called, the `Unimplemented` procedure triggers a system error. u

Patches and System Software Routines

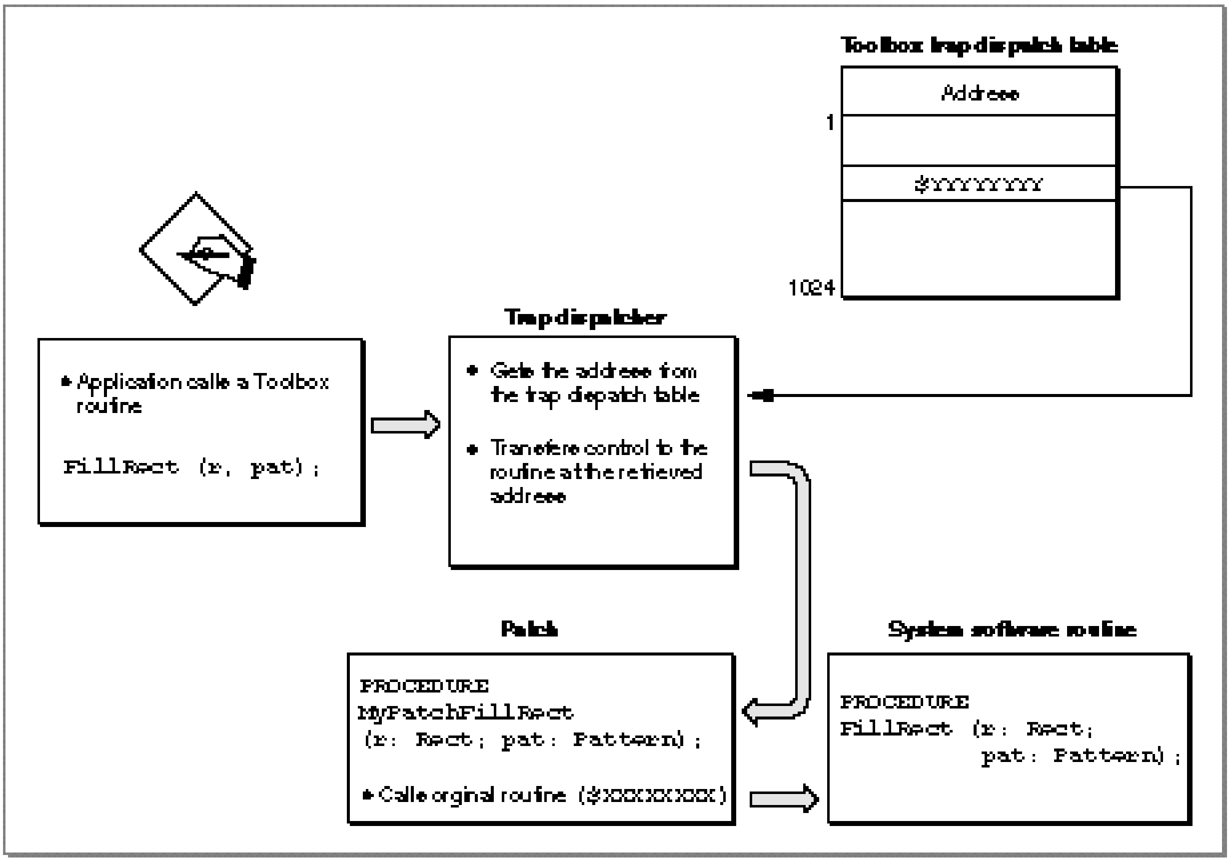
You can modify the trap dispatch table so that the address that gets returned to the trap dispatcher points to a different routine instead of the intended system software routine; this is useful if you want to augment or override an existing system software routine. The routine that augment an existing system software routine is called a **patch**. The method of augmenting or overriding a system software routine is called *patching a trap.*

For example, you can augment the `FillRect` procedure with your own procedure `MyPatchFillRect`. Figure 8-4 illustrates another call to the Toolbox procedure `FillRect`. When the application calls the `FillRect` procedure the application-defined patch `MyPatchFillRect` is executed first. After the application-defined patch `MyPatchFillRect` completes its primary action, it transfers control (through a JMP instruction) to the original `FillRect` procedure.

IMPORTANT

Although this chapter describes patching in some detail, you should avoid any unnecessary patching of the system software. One very good reason to avoid patching is that is causes a performance reduction. The performance reduction is especially substantial when your patch is executed on a PowerPC processor-based Macintosh computer, where it is necessary to switch execution environments when entering and exiting your patch code. For more information about patching PowerPC system software, see *Inside Macintosh: PowerPC System Software.* s

Figure 8-4 Augmenting the `FillRect` procedure with a single patch



Note

To prevent dangling patch addresses, you must ensure that your patch routine is in a locked memory block while its address is in the trap dispatch table. u

Daisy Chain of Patches

It is possible to patch a system software routine with more than just one patch; this is called a **daisy chain** of patches. Typically, you extract from the trap dispatch table the address of the routine you wish to patch, save this address, and then install your own patch routine. When your patch has completed its tasks, it should jump to the address you previously extracted from the trap dispatch table. In this way, the patches take the general form of a daisy chain. Each patch will execute in turn and jump to the next patch until the last link in the chain, which returns control to the trap dispatcher.

IMPORTANT

Although this chapter describes patching in some depth, you should rarely, if ever, find a need to use patches in an application. The primary purposes of patches, as their name suggests, are to fix problems and augment routines in ROM code. s

A patch can be implemented as either a head patch, tail patch, or come-from patch. These are described in the next sections.

Head Patch (Normal Patch)

A **head patch**, also referred to as a **normal patch**, is a routine that gets executed before the original system software routine. A head patch performs its primary action and then uses a jump instruction (`JMP`) to jump to the system software routine. Thus the head patch does not regain control after the execution of the system software routine. After the execution of the system software routine, control is transferred back to the trap dispatcher.

Tail Patch

A **tail patch** is a routine that gets executed before the original system software routine and regains control after the execution of the system software routine. A tail patch uses a jump-subroutine instruction (`JSR`) to transfer control to the system software routine. After the system software routine returns control to the tail patch, the tail patch returns control to the trap dispatcher.

s **WARNING**

You should never install tail patches in system software versions earlier than System 7. Tail patches may conflict with come-from patches, installed by Apple. s

Come-From Patch (Used Only by Apple)

A **come-from patch,** also called a **system patch**, is a type of patch used only by Apple. Come-from patches are used to replace erroneous code or to add capabilities not in ROM.

When a come-from patch is invoked, it examines the stack to determine where it was called from. If the come-from patch was invoked from a particular place in ROM (a spot where the code needs to be augmented or deleted), the come-from patch executes the modifying code. Otherwise, if the come-from patch was called from a part of the system that does not need to be augmented, it transfers control to the next routine in the daisy chain. This routine could be another patch or the system software routine.

Beginning with System 7, the addresses of come-from patches are permanently placed in the trap dispatch table at system startup time. The addresses of come-from patches are hidden and cannot be manipulated by any of the Trap Manger routines.

For example, if a system software routine has a come-from patch and if you use the Trap Manger function `NGetTrapAddress` to retrieve the address of the system software routine, you will not get the address in the trap dispatch table (which is the address of the come-from patch). `NGetTrapAddress` instead returns the address of the routine that is executed immediately after the come-from patch. This address could be the address of another patch or the system software routine.

If a system software routine has a come-from patch and if you use the Trap Manager procedure `NSetTrapAddress` to install a patch to the system software routine, the address of the patch is not written into the trap dispatch table. Instead, the `NSetTrapAddress` procedure installs the address of the patch into the last come-from patch. The patch is executed after the completion of the come-from patch.

S **WARNING**
In system software before System 7, if a come-from patch is invoked by a tail-patch, the come-from patch does not work correctly. The come-from patch never sees the ROM address on the stack—only the return address of the tail-patch. s

Patch for One Application

If you install a patch into your application heap, the patch applies only to your application. When your application is switched out, your application's heap (and patch) is swapped out. For example, if you patch `FillRect` with the patch `MyPatchFillRect`, the `MyPatchFillRect` patch is executed only when the `FillRect` procedure is called from your application.

Note
When running in System 7 or under MultiFinder in System 6, each application has its own copy of the trap dispatch tables. This ensures that an application's patches apply only when it is running and that they're discarded when the application quits. u

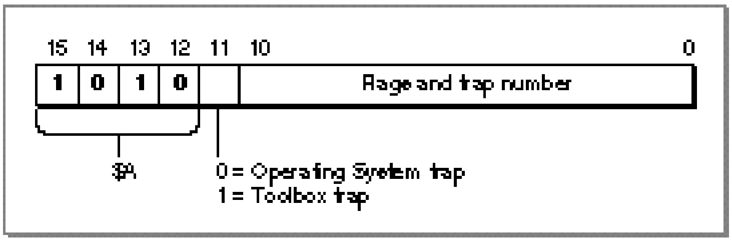
Patch for All Applications

If you install a patch from a system extension during system startup, your patch is placed in the system heap and applies to all applications. For example, if you patch the `FillRect` procedure with the patch `MyPatchFillRect` from a system extension, the `MyPatchFillRect` patch is executed every time the `FillRect` procedure is called, no matter which application calls it.

A-Line Instructions

When your application calls a Toolbox or an Operating System routine, an A-line instruction is sent to the microprocessor. Each A-line instruction contains information about the called system software routine. Figure 8-5 shows the layout of an A-line instruction.

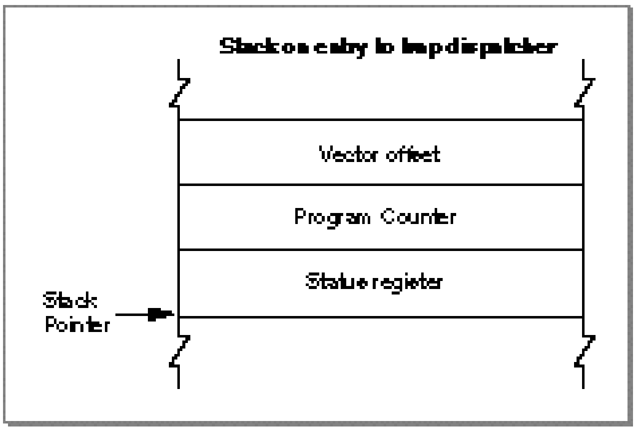
Figure 8-5 A-line instruction format



The high-order 4 bits of an A-line instruction have the hexadecimal value $A, hence the name A-line instruction. Bit 11 of the A-line instruction indicates the type of system software routine to be invoked: a value of 0 in bit 11 indicates an Operating System routine, a value of 1 in bit 11 indicates a Toolbox routine. The trap number in an A-line instruction is used as an index into the appropriate dispatch table. The meaning of the flags vary accordingly to the type of A-line instruction.

When your application calls a system software routine (thereby generating an exception), the microprocessor pushes an **exception stack frame** onto the stack. Figure 8-6 shows a typical exception stack frame. After pushing the exception stack frame on the stack, the microprocessor transfers control to the trap dispatcher.

Figure 8-6 Exception stack frame (on Macintosh computers with a MC68020 microprocessor or greater)



The trap dispatcher discards the status register and vector offset. Depending on whether the A-line instruction is used to invoke an Operating System routine or a Toolbox routine, the trap dispatcher deals with the stack and registers in two very different ways, as described in the next section, “A-line Instructions for Operating System Routines,” and in the section “A-Line Instructions for Toolbox Routines” beginning on page 8-14.

Note

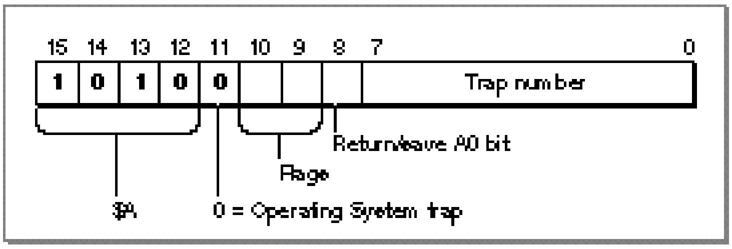
The exception handler is located at address $28 on computers with an MC68000 microprocessor and at address $28 offset from the address in the microprocessor’s Vector Base Register (VBR) on computers with other MC680x0 microprocessors. Consult the relevant microprocessor handbook for the precise details of exception handling on the MC680x0 microprocessor of interest to you. u

A-Line Instructions for Operating System Routines

An **Operating System trap** is an exception that is caused by an A-line instruction that executes an Operating System routine.

When dispatching an Operating System trap, the trap dispatcher extracts the trap number from the A-line instruction and uses it as an index into the Operating System trap dispatch table. The entry in the Operating System trap dispatch table contains the address of the desired Operating System routine. Figure 8-7 illustrates an A-line instruction for an Operating System routine.

Figure 8-7 An A-line instruction for an Operating System routine

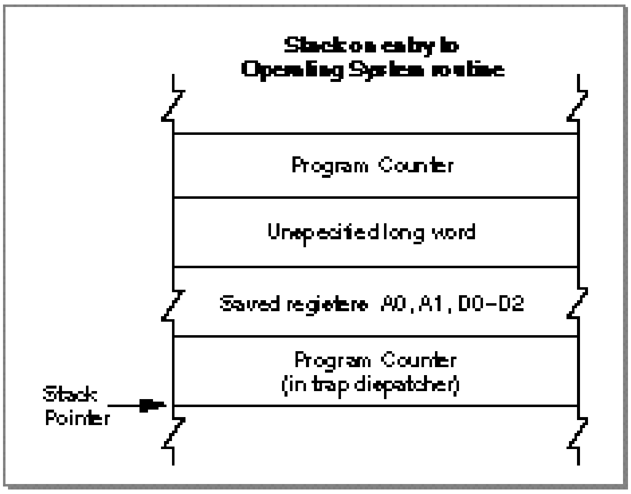


Bit 11 tells the trap dispatcher that this A-line instruction invokes an Operating System routine. Two flag bits, bit 10 and bit 9, are reserved for use by the Operating System routine itself and are discussed in detail in “Flag Bits” on page 8-14. Bit 8 indicates whether the value in register A0 is returned from the Operating System routine. If bit 8 is 0, the value in register A0 is returned from the Operating System routine. If bit 8 is 1, the value in register A0 is not returned by the Operating System routine. As previously described, the trap number is in bits 7–0 and is used to determine which of the 256 possible Operating System routines is executed.

For example, a call to the Operating System function `GetPtrSize` is translated to the A-line instruction $A021. This A-line instruction causes the microprocessor to transfer control to the trap dispatcher, which deals with any instruction of the form $A*xxx*. The trap dispatcher first saves registers D0, D1, D2, A1, and, if bit 8 is 0, A0. The trap dispatcher places the A-line instruction itself into the low-order word of register D1 so that the Operating System routine can inspect the flag bits. Next, the trap dispatcher examines the other bits in the A-line instruction. The value (0) of bit 11 indicates that `GetPtrSize` is an Operating System routine, and that the value in bits 7–0 is the index into the Operating System trap dispatch table. The trap dispatcher uses the index (which is 33 in this example) to find the address of the `GetPtrSize` function in the Operating System trap dispatch table. When the address is found, the trap dispatcher transfers control to the `GetPtrSize` function.

Figure 8-8 illustrates the stack after the trap dispatcher has transferred control to an Operating System routine.

Figure 8-8 The stack on entry to an Operating System routine



The Operating System routine may alter any of the registers D0–D2 and A0–A2, but it must preserve registers D3–D7 and A3–A6. The Operating System routine may return information in register D0 (and A0 if bit 8 is set). To return to the trap dispatcher, the Operating System routine executes the RTS (return from subroutine) instruction.

When the trap dispatcher resumes control, first it restores the value of registers D1, D2, A1, A2, and, if bit 8 is 0, A0. The values in registers D0 and, if bit 8 is 1, in A0 are not restored.

Calling Conventions for Register-Based Routines

Register-based routines receive their parameters from microprocessor registers, and they pass their results in microprocessor registers. Virtually all Operating System routines are register-based routines.

An Operating System routine returns information only in registers D0 and, if bit 8 is 1, A0. The stack and all other registers are unchanged.

Many Operating System routines return a result code in the low-memory word of register D0 to report whether the requested operation was performed successfully. A result code of 0 indicates that the routine completed successfully; any other value typically indicates an error. Just before the trap dispatcher finishes execution, it tests the low-order word of register D0 with a TST.W instruction to set the condition codes of the microprocessor.

Note

Calling conventions for PowerPC microprocessor-based Macintosh computers are different from the calling conventions described for in this section. For information about calling conventions for PowerPC processor-based Macintosh computers, see *Inside Macintosh: PowerPC System Software*. u

Parameter-Passing Conventions for Operating System Routines

By convention, register-based routines normally use register A0 for passing addresses (such as pointers to data objects) and register D0 for other data values (such as integers).

For routines that take more than two parameters, the parameters are normally collected in a parameter block in memory and a pointer to the parameter block is passed in register A0. See the description of an individual routine in the appropriate *Inside Macintosh* book for exact details.

Function Results

Most Operating System functions return their function result (or result code) in register D0. Parameters are returned through register A0, usually as a pointer to a parameter block.

Whether the trap dispatcher preserves register A0 depends on the setting of bit 8 in the A-line instruction. If bit 8 is 0, the trap dispatcher saves and restores register A0; if it’s 1, the routine passes back register A0 unchanged. Thus, bit 8 of the A-line instruction should be set to 1 only for those routines that use register A0 to return information. The trap macros automatically set this bit correctly for each routine.

To see in which register the function passes the function result, see the description of the individual function in the appropriate *Inside Macintosh* book.

Flag Bits

Many Operating System routines use the flag bits in an A-line instruction to encode additional information used by the routine. For example, the A-line instructions that invoke Memory Manager routines define the two flag bits like this:

Bit	Explanation
9	If 1, initialize all bytes in the allocated memory to 0. If 0, do not initialize all bytes in the allocated memory to 0.
8	If 1, allocate memory from the system heap. If 0, allocate memory from the application heap.

These two bits are defined in assembly language as:

```
CLEAR    EQU      $200     ;initialize block to zero
SYS      EQU      $400     ;use the system heap
```

When used with a Memory Manager A-line instruction, these modifiers cause flag bits 9 and 10, respectively, to be set. They could be used in an assembly-language instruction sequence like

```
         MOVEQ    #124,D0        ;need 124 bytes
         _NewPtr     SYS,CLEAR   ;allocate requested memory in
                                 ; system heap and initialize to
                                 ; zeroes
```

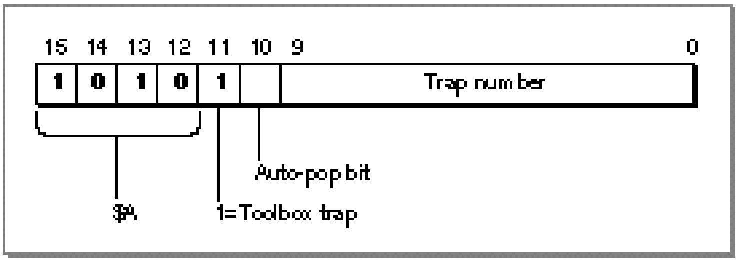
The SYS modifier specifies allocation from the system heap, regardless of the value of the global variable TheZone, and the CLEAR modifier specifies that the Memory Manager should initialize the block contents to zero. For further details, consult *Inside Macintosh: Memory.*

A-Line Instructions for Toolbox Routines

A **Toolbox trap** is an exception that is caused by an A-line instruction that executes a Toolbox routine.

When dispatching a Toolbox trap, the trap dispatcher extracts the trap number from the A-line instruction and uses it as an index into the Toolbox trap dispatch table. The index points to the entry in the Toolbox trap dispatch table that contains the address of the desired Toolbox routine. Figure 8-9 illustrates an A-line instruction that is used to access a Toolbox routine.

Figure 8-9 An A-line instruction for a Toolbox routine

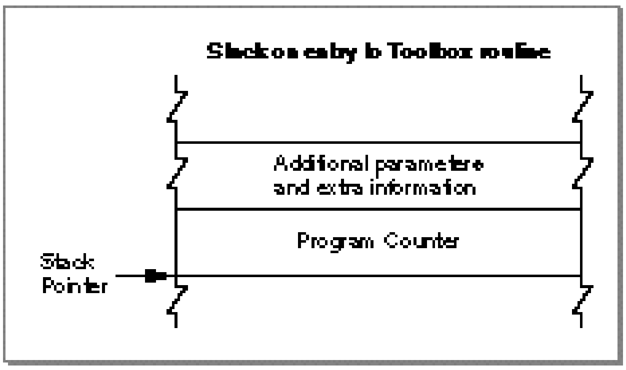


Bit 11 tells the trap dispatcher that this A-line instruction is used to access a Toolbox routine. Bit 10 is the auto-pop bit. Bits 9–0 contain the trap number which, as previously described, determine which of the 1024 possible Toolbox routines is executed. The auto-pop bit is described in detail in "The Auto-Pop Bit" on page 8-20.

For example, a call to the Toolbox function `WaitNextEvent` is translated to the A-line instruction $A860. This A-line instruction causes the microprocessor to transfer control to the trap dispatcher, which deals with any instruction of the form $A*xxx*. The trap dispatcher examines the other bits in the A-line instruction. The value (0) of bit 11 indicates that `WaitNextEvent` is a Toolbox routine and that the value in bits 9–0 is the index into the Toolbox trap dispatch table. The trap dispatcher uses the index (which is $60 in this example) to find the address of the `WaitNextEvent` function in the Toolbox trap dispatch table. When the address is found, the trap dispatcher transfers control to the `WaitNextEvent` function.

Figure 8-10 illustrates the stack after the trap dispatcher has transferred control to a Toolbox routine.

Figure 8-10 Stack when entering a Toolbox routine



The value of the Program Counter that is left on the stack before entry to the Toolbox routine points to the instruction that is executed after the completion of the Toolbox routine.

After the trap dispatcher completes execution, the internal status of the stack is restored, and normal execution resumes from the point at which processing was suspended.

A Toolbox routine changes the Stack Pointer in register A7 and pops the return address and any input parameters. A routine might also alter registers D0–D2, A0, and A1.

s **WARNING**

Some Toolbox routines (for example the `LongMul` procedure described in the chapter “Mathematical and Logical Utilities” in this book) preserve more than the required set of registers. However, you should assume all of registers D0–D2, A0, and A1 are altered by Toolbox routines. s

Calling Conventions for Stack-Based Routines

Stack-based routines receive their parameters on the stack and return their results on the stack. Virtually all Toolbox routines are stack-based routines.

Most Toolbox routines follow Pascal calling conventions; that is, Toolbox routine parameters are evaluated from left to right and are pushed onto the stack in the order in which they are evaluated. Function results are returned by value or by address on the stack. Space for the function result is allocated by the caller before the parameters are pushed on the stack. The caller is responsible for removing the result from the stack after the call.

Note

Calling conventions for PowerPC microprocessor-based Macintosh computers are different from the calling conventions described in this section. For information about calling conventions for PowerPC processor-based Macintosh computers, see *Inside Macintosh: PowerPC System Software.* u

Figure 8-11 illustrates Pascal calling conventions. In this example, a routine calls the application-defined function `MyPascalFn`. When the application calls the function `MyPascalFn`, the application must first make room on the stack for the function result, then push the parameters on the stack in left-to-right order.

Figure 8-11 Pascal calling convention

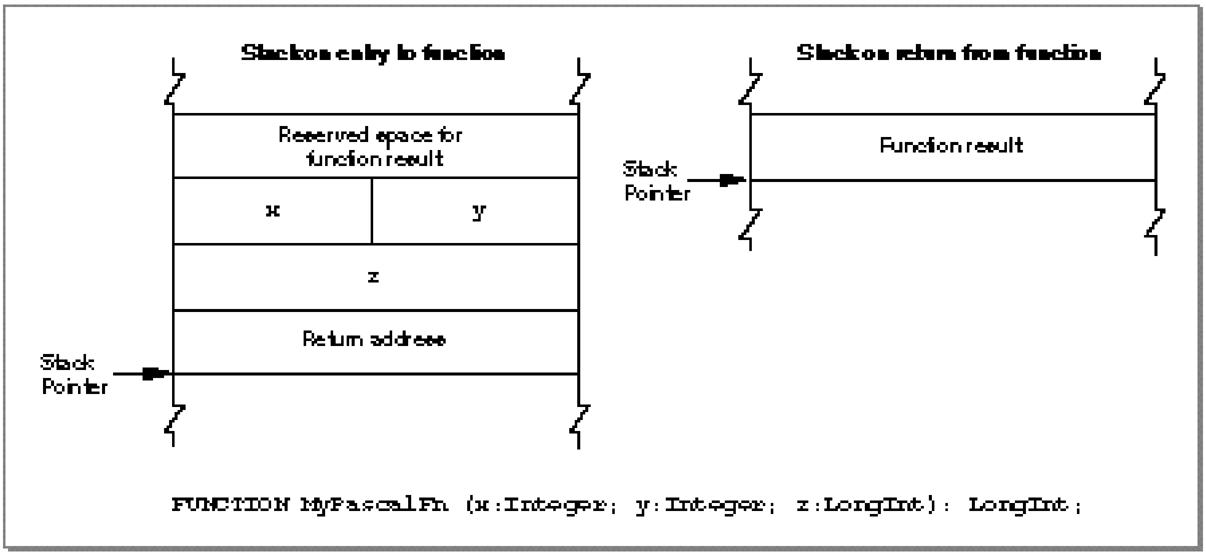
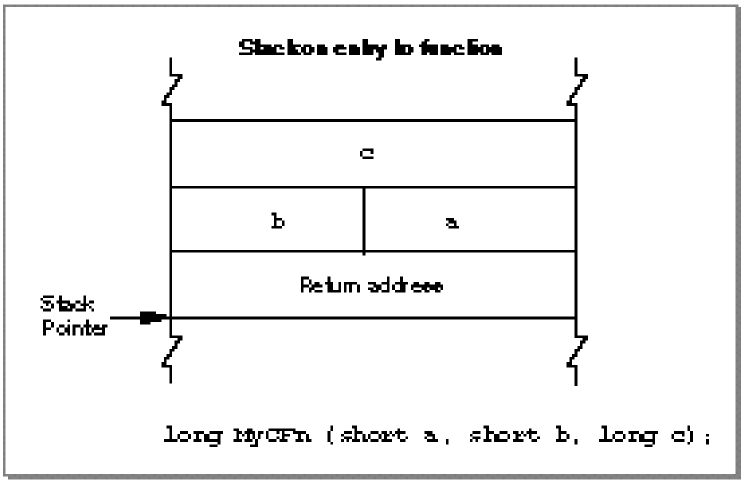


Figure 8-12 illustrates C calling conventions. In this example, a routine calls the application-defined function `MyCFn`. When the application calls the function `MyCFn`, the application pushes the parameters on the stack in right-to-left order. The function result is returned in register D0, and not on the stack.

Figure 8-12 C calling convention



Parameter-Passing Conventions for Toolbox Routines

All variable parameters (parameters of type VAR) are passed as pointers to the actual storage location. In the case of byte-sized types, parameters of type VAR may have odd values.

Nonvariable parameters are passed in different ways, depending on the type of the parameter. Values of type `Boolean`, elements of an enumerated type with fewer than 128 elements, and subranges within the range –128 to 127 are passed as signed byte values. Values of type `Integer` and, `Char` and all other enumerations and subranges are passed as signed word values. Pointers and values of type `LongInt` are passed as signed 32-bit values. Table 8-1 summarizes the parameter-passing conventions.

Table 8-1 Toolbox parameter-passing conventions

Parameter type	Data object pushed on stack
Boolean	Byte: range 0 to 1
Char	16 bits: range 0 to 255
Integer	16 bits: range –32768 to 32767
LongInt	32 bits
Pointer	32 bits
Enumeration: range 0 to 127	Byte: range 0 to 127
Enumeration: range 0 to 32767	16 bits: range 0 to 32767
Subrange: range –128 to 127	16 bits: range –128 to 127
Subrange: range –32768 to 32767	Word: range –32768 to 32767
Real	Address of `Extended` copy
Double	Address of `Extended` copy
Comp	Address of `Extended` copy
Extended	Address of argument
ARRAY, RECORD, string ≤ 4 bytes	Value (word or long word)
ARRAY, RECORD, string > 4 bytes	Address of value
SET	SET value rounded to whole number of words

A parameter of type SET is passed by rounding its size up to the next whole word, if necessary, then pushing its value so that the lowest-order word is pushed last. In the case of a byte-size SET, the called procedure accesses only the low-order half of the word that is pushed.

Note
A byte pushed on the stack occupies the high-order byte of the word allocated for it, according to conventions for the MC680x0 microprocessors. u

s **WARNING**
A value of type `Char` is passed as a word value. The value occupies the low-order half of the word. s

Function Results

Function results are returned by value or by address on the stack. Space for the function result is allocated by the caller before the parameters are pushed. The caller is responsible for removing the result from the stack after the call.

For types `Boolean`, `Char`, and `Integer` and for enumerated and subrange types, the caller allocates a word on the stack to make space for the function result. Values of type `Boolean`, enumerated types with fewer than 128 elements, and subranges within the range –128 to 127 are returned as signed byte values. The value is placed in the high-order byte of the word.

Values of type `Integer` and `Char` and all enumerated and subrange types not covered above are returned as signed word values.

Pointers and values of type `LongInt` are returned as signed 32-bit values. Values of type `Real` are returned as 32-bit real values. For types whose values are greater than 4 bytes in size, the caller pushes a pointer to a temporary location into which the function places the result; these types include `Double` (8 bytes), `Comp` (8 bytes), and `Extended` (10 or 12 bytes); types SET, ARRAY, RECORD; and strings greater than 4 bytes in size.

For a 1-byte SET, for types SET, ARRAY, and RECORD, and for strings whose size is one word, the caller allocates a word on the stack. For types SET, ARRAY, and RECORD and strings whose size is two words, the caller allocates a long word on the stack.

The conventions for returning results of functions are summarized in Table 8-2.

Table 8-2 Conventions for returning results from Toolbox functions

Function result type	Data object left on stack or returned through pointer on stack
Boolean	Byte: range 0 to 1
Char	16 bits: range 0 to 255
Integer	16 bits: range –32768 to 32767
LongInt	32 bits
Pointer	32 bits
Enumeration: range 0 to 127	Byte: range 0 to 127
Enumeration: range 0 to 32767	16 bits: range 0 to 32767

continued

Table 8-2 Conventions for returning results from Toolbox functions (continued)

Function result type	Data object left on stack or returned through pointer on stack
Subrange: range –128 to 127	Byte: range –128 to 127
Subrange: range –32768 to 32767	16 bits: range –32768 to 32767
Real	Real
Double	`Double` at address given by pointer
Comp	`Comp` at address given by pointer
Extended	`Extended` at address given by pointer
ARRAY, RECORD, string ≤ 4 bytes	Value (word or long word)
ARRAY, RECORD, string > 4 bytes	Value at address given by pointer
SET: one byte	Byte value
SET: one word	16-bits value
SET: two words	32-bits value
SET > two words	Value at address given by pointer

Note

A 1 byte-size return value occupies the high-order byte of the word allocated for it. u

The Auto-Pop Bit

The **auto-pop bit** is bit 10 in an A-line instruction for a Toolbox routine. Some language systems prefer to generate jump-subroutine calls (JSR) to intermediate routines, called glue routines, which then call Toolbox routines instead of executing the Toolbox routine directly. This glue method would normally interfere with Toolbox traps because the return address of the glue subroutine is placed on the stack between the Toolbox routine's parameters and the address of the place where the glue routine was called from (where control returns once the Toolbox routine has completed execution).

The auto-pop bit forces the trap dispatcher to remove the top 4 bytes from the stack before dispatching to the Toolbox routine. After the Toolbox routine completes execution, control is transferred back to the place where the glue routine was called from, not back to the glue routine.

Most development environments, including MPW, do not use this feature.

About Trap Macros

A **trap macro** is an assembly-language macro that assembles into an A-line instruction, used for calling a Toolbox or Operating System routine from assembly language. 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. For example, to call the Window Manager function `NewWindow`, you should use an instruction with the macro name `_NewWindow`. There are some exceptions, however, in which the spelling of the macro differs from the name of the Pascal routine itself; these are noted in the documentation for the individual routines.

Trap macros for Toolbox routines take no arguments; any parameters must be pushed on the stack before invoking the routine. See "Calling Conventions for Stack-Based Routines" on page 8-16 for more information. Trap macros for Operating System routines 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. The remaining arguments control the settings of the various flag bits in the A-line instruction.

About Routine Selectors

A routine selector is a value that is pushed on the stack to select a particular routine from a group of routines to be executed. Many trap macros take routine selectors. For example, the trap macro `_HFSDispatch` has the possibility of calling 42 different system software routines. Hence, the trap macro has 42 different routine selectors. The routine selector that is passed on the stack (for `_HFSDispacth` to access) selects which of the 42 software routines `_HFSDispatch` executes.

Most system software routines that are accessed through a trap macro and a routine selector also have a corresponding macro that expands to call the original trap macro and automatically puts the correct routine selector on the stack. For example, the trap macro `_GetCatInfo` expands to call `_HFSDispatch` and places the selector $0009 on the stack after the parameters.

Using the Trap Manager

You can use the Trap Manger to read from and write to a trap dispatch table. To read an address from a trap dispatch table, you can call the `NGetTrapAddress`, `GetOSTrapAddress`, or `GetToolboxTrapAddress` functions. To write an address to a trap dispatch table, you can use the `NGetTrapAddress`, `GetOSTrapAddress`, or `GetToolboxTrapAddress` procedures.

This section shows how you can use the Trap Manager to

- determine if a system software routine is available
- patch a system software routine

Determining If a System Software Routine is Available

You can use the Trap Manager to determine the availability of system software routines.

The Gestalt Manager, introduced in System 6.0.4 and discussed in the chapter "Gestalt Manager" in this book, is the primary tool for querying the system about its features. But if you expect your application to run on a system older than System 6.0.4, the Gestalt Manager may not be available.

The example in this section shows how you can use the Trap Manager to check whether a particular system software routine is available on the installed system.

At startup time, system software places the address of the `Unimplemented` procedure into all entries of each trap dispatch table that do not contain an address of a Toolbox or Operating System routine (or the address of a come-from patch). Listing 8-1 illustrates how you can use these `Unimplemented` addresses to determine whether a particular system software routine is available on the user's system. If a system software routine is available, its address differs from the address of the `Unimplemented` procedure.

Listing 8-1 Determining if a system software routine is available

```
FUNCTION MySWRoutineAvailable (trapWord: Integer): Boolean;
VAR
   trType:  TrapType;
BEGIN
   {first determine whether it is an Operating System or Toolbox routine}
   IF ORD(BAND(trapWord, $0800)) = 0 THEN
      trType := OSTrap
   ELSE
      trType := ToolTrap;
   {filter cases where older systems mask with $1FF rather than $3FF}
   IF (trType = ToolTrap) AND (ORD(BAND(trapWord, $03FF)) >= $200) AND
      (GetToolboxTrapAddress($A86E) = GetToolboxTrapAddress($AA6E)) THEN
      MySWRoutineAvailable := FALSE
   ELSE
      MySWRoutineAvailable := (NGetTrapAddress(trapWord, trType) <>
                              GetToolboxTrapAddress(_Unimplemented));
END;
```

Note

Macintosh Plus and Macintosh SE computers with system software prior to System 7 masked their trap numbers with $1FF in the `GetToolboxTrapAddress` function so that the address of A-line instruction $AA6E (whether implemented or not) would be the same as A-line instruction $A86E, which invokes the `InitGraf` routine. u

You can use the application-defined procedure `MySWRoutineAvailable` to check for system software routines not supported by the Gestalt Manager. A notable example is the `WaitNextEvent` function, which has never had `Gestalt` selectors. Listing 8-2 shows two common uses of the application-defined `MySWRoutineAvailable` procedure.

Listing 8-2 Determining whether `WaitNextEvent` and `Gestalt` are available

```
VAR
   gHasWNE, gHasGestalt:   Boolean;

   {check for the availability of WaitNextEvent}
   gHasWNE := MySWRoutineAvailable(_WaitNextEvent);
   {check for the availability of Getstalt}
   gHasGestalt := MySWRoutineAvailable(_Gestalt);
```

Patching a System Software Routine

Although this chapter describes patching in some depth, you should rarely, if ever, find a need to use patches in an application. The primary purposes of patches, as their name suggests, are to fix problems and augment routines in ROM code. The examples in this section are only included for the sake of completeness.

Listing 8-3 illustrates a patch for the `SysBeep` Operating System procedure. When `SysBeep` is called, this application-defined patch `MySysBeep` is executed before transferring control to the original `SysBeep` procedure.

Listing 8-3 Patching the `SysBeep` Operating System procedure

```
PROCEDURE MySysBeep (duration: Integer);
VAR
   oldPort:    GrafPtr;
   wMgrPort:   GrafPtr;
   i:          Integer;
BEGIN
   GetPort(oldPort);
   GetWMgrPort(wMgrPort);
   SetPort(wMgrPort);
   FOR := 3 DOWNTO 0 DO BEGIN
      InvertRect(wMgrPort^.portBits.bounds);
   END;
   SetPort(oldPort);
END; {of MySysBeep}
```

To transfer control to the next routine in the daisy chain (in this example the original `SysBeep` procedure), the application-defined `MyInstallAPatch` procedure (Listing 8-5) uses the application-defined procedure `MyFollowDaisyChain`, shown in Listing 8-4. The `MyFollowDaisyChain` duplicates the parameter for the `SysBeep` procedure and then pushes the address of the `SysBeep` procedure on the stack. Listing 8-4 shows the application-defined procedure `MyFollowDaisyChain`.

Listing 8-4 Jumping to the next routine in the daisy chain

```
MyFollowDaisyChain PROC EXPORT
IMPORT MYSYSBEEP
   BRA.S    @2
@1 DC.L     $50FFC001
@2 MOVE.W   $4(A7),-(A7)   ;duplicate the parameters
   MOVE.L   @1,-(A7)       ; and push the chain link
   BRA.S    MYSYSBEEP
   NOP
ENDPROC
END
```

The application-defined procedure `MyInstallAPatch` in Listing 8-5 installs a patch into the daisy chain (in this example, the `MySysBeep` patch). First, the procedure calls the `NGetTrapAddress` function to get the address of the next routine in the daisy chain. This address could be the address of another patch or the system software routine. Next, `MyInstallAPatch` calls the `NSetTrapAddress` procedure to put the address of the new patch (in this example, the address of `MySysBeep` patch) into the trap dispatch table.

Listing 8-5 Installing a patch

```
PROGRAM MyPatchInstaller;
USES  Memory, ToolIntf, OSIntf, OSUtils,Windows,
      ToolUtils, Traps, Resources, SamplePatch;
TYPE
PatchCodeHandle = ^PatchCodePtr;
PatchCodePtr = ^PatchCodeHeader;
PatchCodeHeader =
   RECORD
      branch:           Integer;
      oldTrapAddress:   LongInt;
   END;
PROCEDURE MyFollowDaisyChain (duration: Integer); EXTERNAL;
PROCEDURE MyInstallAPatch (trapNumber: Integer; tType: TrapType;
                           pPatchCode: PatchCodePtr);
BEGIN
   pPatchCode^.oldTrapAddress := NGetTrapAddress(trapNumber,
                                                 tType);
   NSetTrapAddress (ORD4(pPatchCode), trapNumber, tType);
END; {of MyInstallAPAtch}
```

```
BEGIN
   InitGraf (@qd.thePort);
   InitFonts;
   InitWindows;
   MyInstallAPatch(_SysBeep, ToolTrap,
                   PatchCodePtr(@MyFollowDaisyChain));
   SysBeep(1);
END. {of MyPatchInstaller}
```

Note

The `MyInstallAPatch` procedure used in this example was designed to install both Operating System and Toolbox patches; it uses the `NGetTrapAddress` and `NSetTrapAddress` routines. The `NGetTrapAddress` and `NSetTrapAddress` routines both need a parameter that indicates which type of routine is being patched, an Operating System or Toolbox routine. u

Trap Manager Reference

This section describes the routines provided by the Trap Manager. You can use these routines to

- access an address in a trap dispatch table
- install a patch address into a trap dispatch table

This section also documents the `Unimplemented` procedure.

Routines

This section describes the routines provided by the Trap Manager.

Accessing Addresses From the Trap Dispatch Tables

You can access the address of a system software routine by using the `GetOSTrapAddress`, `GetToolboxTrapAddress` or `NGetTrapAddress` function. The `GetOSTrapAddress` function retrieves only an Operating System routine address, and the `GetToolboxTrapAddress` retrieves only a Toolbox routine address. The `NGetTrapAddress` function is the most general of these functions; you can use the function to retrieve the address of an Operating System routine or a Toolbox routine.

GetOSTrapAddress

You can use the `GetOSTrapAddress` function to access the address of an Operating System routine, that is located in the Operating System trap dispatch table.

```
FUNCTION GetOSTrapAddress (trapNum: Integer): LongInt;
```

`trapNum` Operating System A-line instruction or a trap number. If you specify an Operating System A-line instruction, the function extracts the trap number for you.

DESCRIPTION

The `GetOSTrapAddress` function returns the address of the Operating System routine specified by the `trapNum` parameter. If the desired Operating System routine is not supported on the installed system software, the `GetOSTrapAddress` function returns the address of the `Unimplemented` procedure. The `trapNum` parameter should contain a trap number in bits 0–7. `GetOSTrapAddress` masks the irrelevant high-order bits. A `GetOSTrapAddress(trapNum)` function call performs the same operation as a `NGetTrapAddress(trapNum, OSTrap)` function call.

SEE ALSO

For more information about the `Unimplemented` procedure, see page 8-32. For information about the `NGetTrapAddress` function, see page 8-27.

GetToolboxTrapAddress

You an use the `GetToolboxTrapAddress` function to access the address of a Toolbox routine, which is located in the Toolbox trap dispatch table. The `GetToolboxTrapAddress` function is also available as the `GetToolTrapAddress` function.

```
FUNCTION GetToolboxTrapAddress (trapNum: Integer): LongInt;
```

`trapNum` Toolbox A-line instruction or a trap number. If you specify a Toolbox A-line instruction, the function extracts the trap number for you.

DESCRIPTION

The `GetToolboxTrapAddress` function returns the address of the Toolbox routine specified by the `trapNum` parameter. If the desired Toolbox routine is not supported on the installed system software, the `GetToolboxTrapAddress` function returns the address of the `Unimplemented` procedure. The `trapNum` parameter should contain a trap number in bits 0–9. `GetToolboxTrapAddress` masks the irrelevant high-order bits. A `GetToolboxTrapAddress(trapNum)` function call performs the same operation as a `NGetTrapAddress(trapNum, ToolTrap)` function call.

SEE ALSO

For more information about the `Unimplemented` procedure, see page 8-32. The `NGetTrapAddress` function is described next. For an example of how to use the `GetToolboxTrapAddress` function, see Listing 8-1 on page 8-22.

NGetTrapAddress

You can use the `NGetTrapAddress` function to retrieve the address of either an Operating System routine or a Toolbox routine.

```
FUNCTION NGetTrapAddress (trapNum: Integer; tTyp: TrapType)
                                 :LongInt;
```

`trapNum` A-line instruction or a trap number. If you specify an A-line instruction, the function extracts the trap number for you.

`tTyp` The trap type. If you supply the `tTyp` parameter with the constant `OSTrap`, the `NGetTrapAddress` function retrieves the address from the Operating System trap dispatch table. If you supply `tTyp` parameter with the constant `ToolTrap`, the `NGetTrapAddress` function retrieves the address from the Toolbox trap dispatch table.

DESCRIPTION

The `NGetTrapAddress` function returns the address of the system software routine specified by the `tTyp` and `trapNum` parameters. If `tTyp` is `OSTrap`, the `NGetTrapAddress` function retrieves the address from the Operating System trap dispatch table. If `tTyp` is `ToolTrap`, the `NGetTrapAddress` function retrieves the address from the Toolbox trap dispatch table. If the desired system software routine is not supported on the installed system software, `NGetTrapAddress` returns the address of the `Unimplemented` procedure. The `trapNum` parameter should contain a trap number in bits 0–7 if `tTyp` is `OSTrap`, and in bits 0–9 if `tTyp` is `ToolTrap`.
The `trapNum` parameter may have any word value; its irrelevant high-order bits are masked according to the value of the `tTyp` parameter.

Note
If the system software routine has a come-from patch, the `NGetTrapAddress` function returns the address of the routine immediately following the come-from patch. u

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for the `_GetTrapAddress` macro are

Registers on entry

D0	An A-line trap word

Registers on exit

A0	Address of next routine in the daisy chain (a system software routine or a patch)

When calling the `_GetTrapAddress` macro, you set bit 9 of the A-line instruction to indicate a "new" system; that is, any version since the Macintosh Plus or Macintosh 512K. You use bit 10 to indicate whether the trap in question is a Toolbox routine (by setting bit 10 to 1) or an Operating System routine (by setting bit 10 to 0). Macintosh development environments provide the modifier words `newTool` and `newOS` to be used as arguments in the `_GetTrapAddress` macro.

To obtain the address of a Toolbox trap whose number is in register D0, you use the macro

```
_GetTrapAddress newTool
```

This is equivalent to calling `NGetTrapAddress (trapNum, newTool)`. The `trapNum` parameter is the A-line trap word placed in register D0 for the assembly-language call. Similarly, to obtain the address of an Operating System routine whose A-line trap word is in register D0, you use the macro

```
_GetTrapAddress newOS
```

This is equivalent to calling `NGetTrapAddress(trapNum, newOS)`.

SEE ALSO

For information about the `Unimplemented` procedure, see page 8-29. For information about the `NSetTrapAddress` function, see page 8-30.

Installing Patch Addresses Into the Trap Dispatch Tables

You can install the address of a patch into a trap dispatch table by using the `SetOSTrapAddress`, `SetToolboxTrapAddress`, or `NSetTrapAddress` procedure. The `SetOSTrapAddress` procedure installs a patch address into the Operating System trap dispatch table, and the `SetToolboxTrapAddress` installs a patch address into the Toolbox trap dispatch table. The `NSetTrapAddress` procedure is the most general of these procedures. You can use the `NSetTrapAddress` procedure to install a patch address into the Operating System trap dispatch table or into the Toolbox trap dispatch table.

SetOSTrapAddress

You can use the `SetOSTrapAddress` procedure to install an Operating System patch address into an Operating System trap dispatch table.

```
PROCEDURE SetOSTrapAddress (trapAddr: LongInt; trapNum: Integer);
```

`trapAddr` The Operating System patch address.

`trapNum` Operating System A-line instruction or a trap number. If you specify an Operating System A-line instruction, the function extracts the trap number (located in bits 0–7) for you.

DESCRIPTION

The `SetOSTrapAddress` procedure places the Operating System patch address specified by the `trapAddr` parameter into the Operating System trap dispatch table. The `trapNum` parameter specifies the location of the Operating System patch address in the Operating System trap dispatch table. The procedure call `SetOSTrapAddress(trapAddr, trapNum)` performs the same operation as a `NSetTrapAddress(trapAddr, trapNum, OSTrap)` procedure call.

Note

If the system software routine that is being patched has any come-from patches, the `SetOSTrapAddress` procedure installs the address of the patch into the exit JMP instruction of the last come-from patch in the chain rather than into the trap dispatch table. u

SEE ALSO

For information about the `Unimplemented` procedure, see page 8-32. For more information about the `NSetTrapAddress` function, see page 8-30.

SetToolboxTrapAddress

You can use the `SetToolboxTrapAddress` procedure to install a Toolbox patch address into the Toolbox trap dispatch table. The `SetToolboxTrapAddress` procedure is also available as the `SetToolTrapAddress` procedure.

```
PROCEDURE SetToolboxTrapAddress (trapAddr: LongInt;
                                 trapNum: Integer);
```

`trapAddr` The Toolbox patch address.

`trapNum` Toolbox A-line instruction or a trap number. If you specify a Toolbox A-line instruction, the function extracts the trap number (located in bits 0–9) for you.

DESCRIPTION

The `SetToolboxTrapAddress` procedure places the Toolbox patch address specified by the `trapAddr` parameter into the Toolbox trap dispatch table. The `trapNum` parameter specifies the location of the Toolbox patch address in the Toolbox trap dispatch table. The `SetToolboxTrapAddress(trapAddr, trapNum)` procedure performs the same operation as a `NSetTrapAddress(trapAddr, trapNum, ToolTrap)` procedure call.

Note

If the system software routine that is being patched has any come-from patches, the `SetToolboxTrapAddress` procedure installs the address of the patch into the exit JMP instruction of the last come-from patch in the chain rather than into the trap dispatch table. u

SEE ALSO

For information about the `Unimplemented` procedure, see page 8-32. The `NSetTrapAddress` function is described next.

NSetTrapAddress

You can use the `NSetTrapAddress` procedure to install a patch address into either an Operating System trap dispatch table or a Toolbox trap dispatch table.

```
PROCEDURE NSetTrapAddress (trapAddr: LongInt; trapNum: Integer;
                           tTyp: TrapType);
```

`trapAddr` The patch address.

`trapNum` A-line instruction or a trap number. If you specify a A-line instruction, the function extracts the trap number you.

`tTyp` The trap type. If you supply the `tTyp` parameter with the constant `OSTrap`, the `NSetTrapAddress` procedure installs the address into the Operating System trap dispatch table. If you supply the `tTyp` parameter with the constant `ToolTrap`, the `NGetTrapAddress` function installs the address into the Toolbox trap dispatch table.

DESCRIPTION

The `NSetTrapAddress` procedure places the patch address specified by the `trapAddr` parameter into a trap dispatch table. Use the `tTyp` parameter to specify whether the patch address belongs in the Operating System trap dispatch table or the Toolbox trap dispatch table. If `tTyp` is `OSTrap`, the `NSetTrapAddress` procedure installs the address into the Operating System trap dispatch table. If `tTyp` is `ToolTrap`, the `NGetTrapAddress` function installs the address into the Toolbox trap dispatch table. Use the `trapNum` parameter to specify the location of the patch address in the dispatch table. The trap number may be any word value; its irrelevant high-order bits are masked according to the value of the `tTyp` parameter.

Note

If the system software routine that is being patched has a come-from patch, the `NSetTrapAddress` procedure installs the address of the patch into the exit JMP instruction of the come-from patch (rather than into the trap dispatch table). u

s **WARNING**

If the first 4 bytes of the `trapAddr` parameter is $60064EF9 (indicating a come-from patch), `NSetTrapAddress` triggers a system error. s

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry for the `_SetTrapAddress` macro are

Registers on entry

D0	An A-line trap word
A0	Address of next routine in the daisy chain (a system software routine or a patch)

When calling the `_SetTrapAddress` macro, you set bit 9 of the A-line trap word to indicate a "new" system; that is, any version since the Macintosh Plus or Macintosh 512K. You use bit 10 to indicate whether the system software routine that is being patched is a Toolbox routine (by setting bit 10 to 1) or an Operating System routine (by setting bit 10 to 0).

Macintosh development environments provide the modifier words `newTool` and `newOS` to be used as arguments in the `_SetTrapAddress` macro.

Given an A-line instruction in register D0 and a system software address in register A0, you set the Toolbox routine with the trap number in register D0 to have the address in A0, you use the macro

```
_SetTrapAddress newTool
```

This is equivalent to calling `NSetTrapAddress(trapAddr, trapNum, newTool)`. The `trapAddr` parameter is the address placed in register A0. The `trapNum` parameter is the A-line instruction placed in D0 for the assembly-language call. Similarly, to set the address of an Operating System trap whose A-line instruction is in register D0 to the address in register A0 you use the macro

```
_SetTrapAddress newOS
```

This is equivalent to calling `NSetTrapAddress(trapAddr, trapNum, newOS)`.

SEE ALSO

The `Unimplemented` procedure is described next. For information about the `NGetTrapAddress` function, see page 8-27. For an example of how to use the `NSetTrapAddress` function, see Listing 8-5 on page 8-24.

Detecting Unimplemented System Software Routines

This section describes the `Unimplemented` procedure. The address of this procedure is placed in all undefined entries of a trap dispatch table. When invoked, the `Unimplemented` procedure triggers a system error.

Unimplemented

The `Unimplemented` procedure triggers a system error when called.

```
PROCEDURE Unimplemented;
```

DESCRIPTION

The address of the `Unimplemented` procedure is at system startup time placed into all entries of each trap dispatch table that do not contain an address of a system software routine. When called, the `Unimplemented` procedure triggers the system error 12, `dsCoreErr`, which crashes the currently running application.

s **WARNING**
Your application should never use this procedure. s

Manipulating *One* Trap Dispatch Table (Obsolete Routines)

This section describes two obsolete Trap Manager routines: `GetTrapAddress` and `SetTrapAddress`. Though a description of the routines are included here, any use of these routines is discouraged.

GetTrapAddress

The `GetTrapAddress` function is obsolete and is documented here only for the sake of completeness.

```
FUNCTION GetTrapAddress (trapNum: Integer): LongInt;
```

`trapNum` Toolbox A-line instruction or a trap number. If you specify an A-line instruction, the function extracts the trap number for you.

DESCRIPTION

The GetTrapAddress function was used when both the Operating System trap addresses and Toolbox trap addresses were located in the same trap dispatch table. Today, any system software routine with the trap number $00 to $4F, $54, or $57 is drawn from the Operating System dispatch table; any other software routine is taken from the Toolbox dispatch table.

s **WARNING**
The GetTrapAddress function is not supported under Power PC. s

s **WARNING**
The GetTrapAddress procedure ignores the high-order bits in the trapNum parameter; the procedure is not able to differentiate between Operating System routines and Toolbox routines. The GetTrapAddress procedure is not reliable on any computer today. s

SetTrapAddress

The SetTrapAddress procedure is obsolete, and is documented here only for the sake of completeness.

```
PROCEDURE SetTrapAddress (trapAddr: LongInt; trapNum: Integer);
```

trapAddr	The address of the system software routine.
trapNum	A-line instruction or a trap number. If you specify an A-line instruction, the function extracts the trap number you.

DESCRIPTION

The SetTrapAddress procedure was used when both the Operating System routine addresses and Toolbox routine adddresses were located in the same trap dispatch table. Today, any routine address with the trap number $00 to $4F, $54, or $57 is installed into the Operating System dispatch table; any other system software routine is installed into the Toolbox dispatch table.

s **WARNING**
The SetTrapAddress procedure is not supported under Power PC. s

s **WARNING**
The SetTrapAddress procedure ignores the high-order bits in the trapNum parameter; the procedure is not able to differentiate between Operating System routines and Toolbox routines. The SetTrapAddress procedure is not reliable on any computer today. s

Summary of the Trap Manager

Pascal Summary

Constants

```
CONST
   {Gestalt selectors}
   gestaltOSTable          = 'ostt';   {base of Operating System dispatch }
                                       { table}
   gestaltToolboxTable     = 'tbtt';   {base of Toolbox dispatch table}
   gestaltExtToolboxTable  = 'xttt';   {0, unless Toolbox dispatch table }
                                       { is disjoint, in which case base }
                                       { of upper half}

   {system errors triggered by the Trap Manager}
   dsCoreErr               = 12;       {unimplemented trap error}
   dsBadPatchHeader        = 83;       {attempt to install a come-from patch}
```

Data Types

```
TYPE TrapType     = (OSTrap, ToolTrap);
```

Routines

Accessing Addresses From the Trap Dispatch Tables

```
FUNCTION GetOSTrapAddress   (trapNum: Integer): LongInt;
{GetToolboxTrapAddress is also spelled as GetToolTrapAddress}
FUNCTION GetToolboxTrapAddress
                            (trapNum: Integer): LongInt;
FUNCTION NGetTrapAddress    (trapNum: Integer; tTyp: TrapType): LongInt;
```

Installing Patch Addresses Into the Trap Dispatch Tables

```
PROCEDURE SetOSTrapAddress  (trapAddr: LongInt; trapNum: Integer);
{SetToolboxTrapAddress is also spelled as SetToolTrapAddress}
```

```
PROCEDURE SetToolboxTrapAddress
                             (trapAddr: LongInt; trapNum: Integer);
PROCEDURE NSetTrapAddress    (trapAddr: LongInt; trapNum: Integer;
                              tTyp: TrapType);
```

Detecting Unimplemented System Software Routines

```
PROCEDURE Unimplemented;
```

Manipulating *One* Trap Dispatch Table (Obsolete Routines)

```
FUNCTION GetTrapAddress      (trapNum: Integer): LongInt;
PROCEDURE SetTrapAddress     (trapAddr: LongInt; trapNum: Integer);
```

C Summary

Constants

```
/*Gestalt selectors*/
#define gestaltOSTable        'ostt'   /*base of Operating System dispatch */
                                       /* table*/
#define gestaltToolboxTable   'tbtt'   /*base of Toolbox dispatch table*/
#define gestaltExtToolboxTable'xttt'   /*0, unless Toolbox dispatch table */
                                       /* is disjoint, in which case base */
                                       /* of upper half*/

/*values of TrapType*/
enum {OSTrap, ToolTrap};

/*system errors triggered by Trap Manager*/
enum {
   dsCoreErr         = 12,             /*unimplemented trap error*/
   dsBadPatchHeader  = 83              /*attempt to install come-from patch*/
};
```

Data Types

```
typedef unsigned char TrapType;
```

Routines

Accessing Addresses From the Trap Dispatch Tables

```
pascal long NGetTrapAddress
                            (short trapNum, TrapType tTyp);
pascal long GetOSTrapAddress
                            (short trapNum);
/*GetToolboxTrapAddress is also spelled as GetToolTrapAddress*/
pascal long GetToolboxTrapAddress
                            (short trapNum);
```

Installing Patch Addresses Into the Trap Dispatch Tables

```
pascal void NSetTrapAddress
                            (long trapAddr, short trapNum,
                             TrapType tTyp);
pascal void SetOSTrapAddress
                            (long trapAddr, short trapNum);
/*SetToolboxTrapAddress is also spelled as SetToolTrapAddress*/
pascal void SetToolboxTrapAddress
                            (long trapAddr, short trapNum);
pascal void SetToolTrapAddress
                            (long trapAddr, short trapNum);
```

Detecting Unimplemented System Software Routines

```
pascal void Unimplemented   (void);
```

Manipulating *One* Trap Dispatch Table (Obsolete Routines)

```
pascal long GetTrapAddress  (short trapNum);
pascal void SetTrapAddress  (long trapAddr, short trapNum);
```

Assembly-Language Summary

Constants

```
newOS          EQU   $200     ;access Operating System trap dispatch table;
newTool        EQU   $600     ;access Toolbox trap dispatch table
```

Trap Macros

Trap Macros Requiring Register Setup

Trap macro name	Registers on entry	Registers on exit
`_GetTrapAddress`	D0: trap number	A0: address of patch
`_SetTrapAddress`	D0: trap number A0: address of patch	
`_Unimplemented`		

C H A P T E R 9

Start Manager

Contents

This chapter describes the system initialization and system startup process performed by the Macintosh computer. It describes the Start Manager, which lets you specify a few global settings that affect the startup process, and it describes initialization-dependent code, such as system extensions, that the system runs while starting up the computer.

You should read this chapter if you are developing a device driver or other code that is installed at some point during the system initialization and startup process, or if you want to use the Start Manager routines.

This chapter begins with a description of the initialization and startup process performed on Macintosh computers. It then

- describes the boot blocks and defines the fields in the boot block header
- defines global variables that provide timing information
- discusses the Start Manager routines you can use to identify and set default devices and to get and set the timeout interval for the startup drive
- describes how to write a system extension

System Initialization and Startup

When a Macintosh computer is first turned on, but before it can load and run an application, it must go through system initialization and system startup. At **system initialization**, the system initialization code located in ROM is executed: memory is tested and initialized, slot cards are initialized, ROM drivers are installed, device drivers are located, and more. The next section, "System Initialization," describes the various steps included in system initialization. At **system startup**, the system code that is located on the startup disk is executed: various software modules are initialized and system extensions are run. The section "System Startup" on page 9-4 describes various steps included in system startup.

WARNING

The system initialization and system startup process is not the same for all Macintosh models. In addition, the system initialization sequence and system startup sequence listed in this chapter are both subject to change; therefor use the information in these sections only for informational purposes.

You should read this section if you provide a system extension that installs software, such as a device driver or other code, during system initialization or system startup.

System Initialization

Initialization on a Macintosh computer begins as soon as the power is first supplied to it. Built-in hardware circuits initialize the main processor and other ICs and temporarily alter the memory mapping to make an image of the ROM appear at the location where RAM normally starts (address 0), while making RAM appear at a location higher in memory. This mapping scheme allows the startup routines in the initialization code to obtain critical low-memory vectors. After the initialization code begins executing and obtains the low-memory vectors, it resets the memory mapping back to normal. For further details on this process, see the *Guide to Macintosh Family Hardware.*

The following list summarizes the events that typically take place when the initialization code in ROM is executed.

IMPORTANT

The system initialization sequence is subject to change; the information in this section is provided for informational purposes only. s

1. Hardware is initialized. The initialization code performs a set of diagnostic tests to verify functionality of some vital hardware components. If the diagnostics succeed, the initialization code initializes these hardware components. If diagnostics fail, the initialization code issues diagnostic tones to indicate the type of hardware failure.
   The initialization code determines how much RAM is available and tests it, then validates the parameter RAM (PRAM). Parameter RAM contains a user's preferences for settings of various control panel settings and port configurations.
   The initialization code determines the global timing variables, `TimeDBRA`, `TimeSCCDB`, and `TimeSCSIDB`. (See "Global Timing Variables" on page 9-9 for more information) and initializes the Resource Manager, Notification Manager, Time Manager, and Deferred Task Manager.
2. On machines with expansion slots, the initialization code initializes the Slot Manager. The Slot Manager then initializes any installed cards by executing the primary initialization code in each card's declaration ROM. Video expansion cards, including built-in video, initialize themselves by determining the type of connected monitor, and then set the display to 1 bit per pixel, and display a gray screen (alternating black and white dots).
3. The initialization code initializes the Vertical Retrace Manager and Gestalt Manager. ROM drivers for all built-in functionality are installed in the unit table and initialized. The initialization code initializes the Apple Desktop Bus (ADB) Manager that then initializes each ADB device. The initialization code initializes the Sound Manager and SCSI Manager.
4. The initialization code loads drivers from all on-line SCSI devices.
5. The initialization code chooses the boot device, and calls the boot blocks to begin initialization of the System Software.

Having initialized the computer's slots, drivers, and hardware, as well as some of the Operating System managers, the initialization code dispatches to the startup code, which immediately begins the startup procedure described in the next section, "System Startup."

System Startup

System startup begins as soon as the initialization code in ROM transfers control to the system startup code. The system startup code is responsible for initializing AppleTalk, the debugger, and system extensions. System extensions are covered in detail in the section “Writing a System Extension” beginning on page 9-10.

This section covers the startup sequence for Macintosh computers running System 7 or later; it then describes the boot blocks and defines the boot block header.

The following list summarizes the events that take place when the system startup code is executed.

IMPORTANT

The system startup sequence is subject to change; the information in this section is provided for informational purposes only. s

1. The system startup code looks for an appropriate startup device. It first checks the internal 3.5-inch floppy drive. If a disk is found, it attempts to read it and looks for a System file. If it doesn’t find a disk or System file, it checks the default startup device specified by the user in the Startup Disk control panel. If no default device is specified or if the device specified is not connected, it checks for other devices connected to the SCSI port, beginning with the internal drive and proceeding successively from drive 6 through drive 1. If it doesn’t find a startup device, it displays the question-mark disk icon until a disk is inserted. If the startup device itself fails, the startup code displays the sad Macintosh icon until the computer is turned off.
2. After selecting a startup device, the system startup code reads system startup information from the startup device. The system startup information is located in the boot blocks, the logical blocks 0 and 1 on the startup disk. The boot blocks contain important information such as the name of the System file and the Finder. The boot blocks are described in detail in the next section.
3. The system startup code displays the happy Macintosh icon.
4. The system startup code reads the System file and uses that information to initialize the System Error Handler and the Font Manager.
5. The system startup code verifies that the necessary hardware is available to boot the system software and displays on the startup screen an alert box with the message “Welcome to Macintosh.”
6. The system startup code performs miscellaneous tasks: it verifies that enough RAM is available to boot the system software, it loads and turns on Virtual Memory if it is enabled in the Memory control panel, it loads the debugger, if present. (The system startup information contains the name of the debugger —usually MacsBug), it sets up the disk cache for the file system, and it loads and executes CPU-specific software patches. At this point, the system begins to trace mouse movement.
7. For any NuBus cards installed, the system startup code executes the secondary init code on the card’s declaration ROM.
8. The system startup code loads and initializes all script systems, including components for all keyboard input methods. It also executes the initialization resources in the System file.
9. The system startup code loads and executes system extensions. (System extensions can be located in the Extensions folder, in the Control Panels folder, and in the System Folder).

10. The system startup code launches the Process Manager, which takes over at this point and launches the Finder. The Finder then displays the desktop and the menu bar. The desktop shows all mounted volumes; it also shows any windows that were open the last time the computer was shut down. The Memory Manager sets up a large, unsegmented application heap, which is divided into partitions as applications start up.

At this point, the system has successfully booted.

The next section, "Boot Blocks," describes the format of the boot block header. This header contains information that the startup code uses to start up the system.

Boot Blocks

The first two logical blocks on every Macintosh volume are **boot blocks.** These blocks contain **system startup information:** instructions and information necessary to start up (or "boot") a Macintosh computer. This information consists of certain configurable system parameters (such as the capacity of the event queue, the number of open files allowed, and so forth) and is contained in a boot block header. The system startup information also includes actual machine-language instructions that could be used to load and execute the System file. Usually these instructions follow immediately after the boot block header. Generally, however, the boot code stored on disk is ignored in favor of boot code stored in a resource in the System file.

The boot block header has a structure that can be described by the `BootBlkHdr` data type.

s **WARNING**
The format of the boot block header is subject to change. If your application relies on the information presented here, it should check the boot block header version number and react gracefully if that number is greater than that documented here. s

Note that there are two boot block header formats. The current format includes two fields at the end that are not contained in the older format. These fields allow the Operating System to size the system heap relative to the amount of available physical RAM. A boot block header that conforms to the older format sizes the system heap absolutely, using values specified in the header itself. You can determine whether a boot block header uses the current or the older format by inspecting a bit in the high-order byte of the `bbVersion` field, as explained in its field description.

```
TYPE BootBlkHdr =          {boot block header}
RECORD
   bbID:          Integer; {boot blocks signature}
   bbEntry:       LongInt; {entry point to boot blocks}
   bbVersion:     Integer; {boot blocks version number}
   bbPageFlags:   Integer; {used internally}
   bbSysName:     Str15;   {System filename}
   bbShellName:   Str15;   {Finder filename}
   bbDbg1Name:    Str15;   {first debugger filename}
```

```
   bbDbg2Name:    Str15;   {second debugger filename}
   bbScreenName:  Str15;   {name of startup screen}
   bbHelloName:   Str15;   {name of startup program}
   bbScrapName:   Str15;   {name of system scrap file}
   bbCntFCBs:     Integer; {number of FCBs to allocate}
   bbCntEvts:     Integer; {number of event queue elements}
   bb128KSHeap:   LongInt; {system heap size on 128K Mac}
   bb256KSHeap:   LongInt; {system heap size on 256K Mac}
   bbSysHeapSize: LongInt; {system heap size on all machines}
   filler:        Integer; {reserved}
   bbSysHeapExtra:LongInt; {additional system heap space}
   bbSysHeapFract:LongInt; {fraction of RAM for system heap}
END;
```

Field descriptions

`bbID` — A signature word. For Macintosh volumes, this field always contains the value $4C4B.

`bbEntry` — The entry point to the boot code stored in the boot blocks. This field contains machine-language instructions that translate to `BRA.S *+$90` (or `BRA.S *+$88`, if the older block header format is used), which jumps to the main boot code following the boot block header. This field is ignored, however, if bit 6 is clear in the high-order byte of the `bbVersion` field or if the low-order byte in that field contains $D.

`bbVersion` — A flag byte and boot block version number. The high-order byte of this field is a flag byte whose bits have the following meanings:

Bit	Meaning
0-4	Reserved; must be 0
5	Set if relative system heap sizing is to be used
6	Set if the boot code in boot blocks is to be executed
7	Set if new boot block header format is used

If bit 7 is clear, then bits 5 and 6 are ignored and the version number is found in the low-order byte of this field. If that byte contains a value that is less than $15, the Operating System ignores any values in the `bb128KSHeap` and `bbSysHeapSize` fields and configures the system heap to the default value contained in the `bbSysHeapSize` field. If that byte contains a value that is greater than or equal to $15, the Operating System sets the system heap to the value in `bbSysHeapSize`. In addition, the Operating System executes the boot code in the `bbEntry` field only if the low-order byte contains $D.

If bit 7 is set, the Operating System inspects bit 6 to determine whether to execute the boot code contained in the `bbEntry` field and inspects bit 5 to determine whether to use relative sizing of the

	system heap. If bit 5 is clear, the Operating System sets the system heap to the value in `bbSysHeapSize`. If bit 5 is set, the system heap is extended by the value in `bbSysHeapExtra` plus the fraction of available RAM specified in `bbSysHeapFract`.
`bbPageFlags`	Used internally.
`bbSysName`	The name of the System file.
`bbShellName`	The name of the shell file. Usually, the system shell is the Finder.
`bbDbg1Name`	The name of the first debugger installed during the boot process. Typically this is Macsbug.
`bbDbg2Name`	The name of the second debugger installed during the boot process. Typically, this is Disassembler.
`bbScreenName`	The name of the file containing the information (welcome message) initially displayed on the startup screen. Usually, this is StartUpScreen.
`bbHelloName`	The name of the startup program. Usually, this is the Finder.
`bbScrapName`	The name of the system scrap file. Usually, this is the Clipboard.
`bbCntFCBs`	The number of file control blocks (FCBs) to put in the FCB buffer. In System 7 and later, this field specifies only the initial number of FCBs in the FCB buffer because the Operating System can usually resize the FCB buffer if necessary. See the chapter "File Manager" in *Inside Macintosh: Files* for details on the file control block (FCB) buffer.
`bbCntEvts`	The number of event queue elements to allocate. This number determines the maximum number of events that can be stored by the Event Manager at any one time. Usually this field contains the value 20.
`bb128KSHeap`	The size of the system heap on a Macintosh computer having 128 KB of RAM.
`bb256KSHeap`	Reserved.
`bbSysHeapSize`	The size of the system heap on a Macintosh computer having 512 KB or more of RAM. This field might be ignored, as explained in the description of the `bbVersion` field.
`filler`	Reserved.
`bbSysHeapExtra`	The minimum amount of additional system heap space required. If bit 5 of the high-order word of the `bbVersion` field is set, this value is added to the `bbSysHeapSize`.
`bbSysHeapFract`	The fraction of RAM available to be used for the system heap. If bit 5 of the high-order word of the `bbVersion` field is set, this fraction of available RAM is added to the `bbSysHeapSize`.

Global Timing Variables

During system initialization, the initialization code initializes the following global variables with timing information.

Variable	Contents
TimeDBRA	The number of times the DBRA (decrement branch always instruction) can be executed per millisecond.
TimeSCCDB	The number of times the SCC can be accessed per millisecond.
TimeSCSIDB	The number of times the SCSI can be accessed per millisecond.

Note

The TimeDBRA value is calculated in ROM and is affected by the processing method of the CPU. Accordingly, for routines running in RAM, it is not necessarily a good measure of how fast the computer is. u

About the Start Manager

The Start Manager lets you set the Macintosh computer's default startup and video devices. The Start Manager also lets you get or set the timing interval for the startup drive.

The Start Manager provides routines that let you specify a default startup device, a default video device, a default operating system, and a default timeout interval for the startup drive. Because all Start Manager routines run under the Macintosh Operating System, you cannot execute them early enough in the initialization process to transfer control to another operating system. Start Manager routines constitute just a small part of the process required to boot another operating system on a Macintosh computer. Most programmers should have no reason to use these routines.

The next section gives an overview of how to use the Start Manager routines.

Using the Start Manager

The Start Manager provides a set of simple routines that get and set information in a word in parameter RAM. This information indicates the default status of some peripheral devices connected to the Macintosh computer. Three of these routines get information about the default startup device, default video device, and the default operating system. Another three routines enable you to set this information. The remaining two routines get and set the timeout interval for the startup drive.

The GetDefaultStartup procedure returns information about the default startup device, and the SetDefaultStartup procedure lets you specify a slot or SCSI device as the default startup device. The **default startup device** is the drive on which the startup code first attempts to start up the Operating System. The Startup Disk control panel calls the GetDefaultStartup and SetDefaultStartup procedures when the user changes the startup disk. Another pair of routines, the GetVideoDefault and SetVideoDefault procedures, get information about and set the **default video device**— essentially, the monitor on which the Macintosh computer displays the message “Welcome to Macintosh” and other startup information. The Monitors control panel calls the GetVideoDefault and SetVideoDefault procedures when the user changes the startup screen. Any changes made to settings in the Monitors control panel take affect at the next system startup.

A third pair of routines, the GetOSDefault and SetOSDefault procedures, enable you to get information about and set the **default operating system**—the operating system that the processor attempts to initialize and start up. At present, the only default operating systems allowed is the Macintosh Operating System.

The last two routines, the GetTimeout and SetTimeout procedures, get or set the timeout interval for the startup drive. The **timeout interval** is the interval of time the system waits for the startup drive to respond while the computer is booting. A disk driver might need to change the timeout interval, for example if the drive takes a long time to reach operating speed.

Writing a System Extension

This section discusses

- the profile of a system extension
- the user interface for a system extension
- how to create additional resources for a system extension
- how to compile a system extension

Before you begin to write a system extension, consider whether the feature that you have in mind is best governed by a system extension. A system extension does not enjoy the full status of an application. The user cannot launch a system extension. During system startup, each system extension is simply loaded and executed in a temporary heap that the system deallocates after the extension is called.

Profile of a System Extension

A **system extension** is a file (of file type 'INIT') containing a code resource of type 'INIT' and additional other resources. A system extension typically contains code that provides a system-level service, such as a printer driver or a patch to a system software routine, and it contains code that loads this system-level service into the system at system startup time.

Listing 9-1 illustrates code for a simple system extension called MySampleINIT. When launched at system startup, MySampleINIT loads the MyShutDownBeep code resource into the system heap, installs a pointer to the shutdown code in the shutdown queue, and displays an icon indicating whether the installation succeeded or failed. The `MyShutDownBeep` procedure is executed just before the Macintosh computer shuts down or restarts. For more information about the shutdown process and the Shutdown Manager, see the chapter "Shutdown Manager" in *Inside Macintosh: Processes*.

The code for `MySampleINIT` places the `MyShutDownBeep` procedure in the system heap, making this procedure available after system startup. The `MyShutDownBeep` procedure calls `SysBeep` just before the Macintosh computer shuts down or restarts.

Listing 9-1 The `MySampleINIT` system extension

```
UNIT MySampleINIT    {write a Pascal system extension as a UNIT}

INTERFACE
USES
   Types, Events, Errors, Resources, Memory, Shutdown;
CONST
   kIconIDSuccess = 128;   {icon of this system extension}
   kIconIDFailure = 129;   {icon of this system extension }
                           { with an "X" on it}
   kMyShutDownResourceType = 'SHUT'
   kMyShutDownResourceID = 128;
   moveX = -1;

IMPLEMENTATION
PROCEDURE MyShowINIT(theIcon, moveX: Integer); EXTERNAL;
PROCEDURE MyShutDownBeep; FORWARD;

PROCEDURE MyINIT;
VAR
   theIcon:             Char;
   myShutDownCodeHndl:  Handle;
   myShutDownCodePtr:   ProcPtr;
BEGIN
   theIcon := kIconIDSuccess;
   {retrieve a handle to MyShutDownBeep procedure}
   myShutDownCodeHndl := GetResource(kMyShutDownResourceType,
                                     kMyShutDownResourceID);
   IF ((myShutDownCodeHndl = NIL) OR
      (ResError <> noErr) ) THEN
         theIcon := kIconIDFailed;
```

```
IF (theIcon = kIconIDSuccess) THEN
      BEGIN
         {the MyShutDownBeep code resource is present, detach it}
         { from the resource file and check for an error}
         DetachResource(myShutDownCodeHndl);
         IF (ResError <> noErr) THEN
            theIcon = kIconIDFailed;
         ELSE
            ReleaseResource(myShutDownCodeHndl);
      END;
      IF (theIcon = kIconIDSuccess) THEN
         BEGIN
            MoveHHi(myShutDownCodeHndl);
            HLock(myShutDownCodeHndl);
         END;
      MyShowINIT(theIcon, moveX);{place the icon at boot time}
      {install MyShutDownBeep procedure into shutdown queue}
      myShutDownCodePtr := myShutDownCodeHndl^);
      ShutDwnInstall(myShutDownCodePtr, sdOnUnmount);
END;

PROCEDURE MyShutDownBeep;
BEGIN
   SysBeep(40);
END;

END. {of UNIT}
```

Notice that the code for the `MySampleINIT` extension is defined as a Pascal `UNIT` rather than a `PROGRAM`. This distinction is important because Pascal programs are applications that require an application heap, an initialized A5 register, the Segment Loader, and the services of other Operating System and Toolbox managers. By comparison, a Pascal unit is merely a collection of routines. It does not enjoy the full status of an application. You cannot launch a system extension. It is simply loaded and executed in a temporary heap that the system deallocates soon after the system finishes booting the computer.

When `MySampleINIT` calls the application-defined procedure `MyShowInit`, `MyShowInit` displays an icon on the bottom left of the startup screen, and it does not erase the screen. If you want an icon displayed at system startup time, you must supply this application-defined procedure.

IMPORTANT

If you provide a procedure that displays an icon of your system extension, do no erase the screen. s

For information about compiling system extensions, see the section “Building a System Extension” beginning on page 9-17.

Note

System extensions are not well equipped to declare global variables and deal with the A5 world. Stand-alone code modules that do these things are not system extensions and thus are beyond the scope of this discussion. See the chapter “Writing Stand-Alone Code” in *Building and Managing Programs in MPW* for information on this topic. u

Because a system extension possesses no A5 world of its own, it cannot easily define global variables: the system allocates no space for them, and the A5 register contains no meaningful value. Extension code that defines global variables usually compiles and links successfully without a warning from the linker; however, the extension’s global variables typically overwrite globals defined by the current application.

s **WARNING**

Code containing references to global variables defined in the MPW libraries, such as QuickDraw globals, generate fatal link errors. s

As a general rule, a system extension can call Operating System managers at any time, but it can call only a few of the Toolbox managers before the startup process completes. It can call the routines from the File Manager, Memory Manager, Resource Manager, and the Notification Manager before the system extension is completely launched, but it must refrain from calling the `InitFonts`, `InitWindows`, `InitDialogs`, `InitMenus` and `TEInit` procedures, as well as other QuickDraw, Window Manager, Dialog Manager, and Font Manager routines. (Note that the code installed by a system extension can utilize the full set of Operating System and Toolbox routines.)

A system extension must do without the services of the Segment Loader, which divides application code into segments that the processor can handle. The size of a system extension’s code resource should not exceed 32 KB.

You should consider installing your system extension in the system heap if you want its resources to be available after the computer finishes booting. For example, some system extensions leave routines in the system heap that can be called through patches to those routines. The `MySampleINIT` system extension shown in Listing 1-1 on page 9-11 loads the `MyShutDownBeep` procedure in the system heap.

The procedure your system extension uses to install code in the system heap varies according to what you want to accomplish. Basically, you have to request a block of memory in the system heap and store the code or resources you want to preserve in the block. To allocate memory in the system heap in System 7 and later, you merely need to call the appropriate Memory Manager routines, and the system heap expands dynamically to meet your requests. In earlier versions of system software, you must use a system heap space resource of type '`sysz`' to indicate how much the Operating System should increase the size of the system zone.

See the chapter “Memory Manager” in *Inside Macintosh: Memory* for details on how to allocate memory in the system heap.

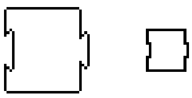
Defining the User Interface for a System Extension

The user interface for a system extension consists of

- the system extension icon
- other elements your system extension needs to communicate with the user

You should provide an icon for the file that contains your system extension. An extension icon looks like a puzzle piece. Figure 9-1 illustrates the default icon for a system extension that appears in the Finder if you don't supply a custom icon for your system extension. You can customize an extension icon by adding a graphic to the default icon. You can display the system extension icon in a horizontal or vertical orientation with the protruding part facing any direction. If you do add graphics, keep them simple so that the icon still looks good when scaled to the small, 16-by-16 pixel icon size.

Figure 9-1 The default system extension icon



The code in your system extension should also display the icon for your system extension when it is first executed at system startup time. You typically display this icon near the bottom-left corner of the startup screen. If the code installed by your extension requires resources or hardware that is not available at system startup, your extension can instead display a crossed-out version of the system extensions icon in the bottom-left corner of the screen.

You should design a system extension so that a user can install it by dragging the icon on top of the System Folder. The Finder then asks the user whether to place the system extension in the Extensions folder. Do not install system extensions in the System file.

When designing a system extension, avoid displaying dialog or alert boxes that interrupt system booting. Whenever possible, use the Notification Manager to notify users of important messages. See the chapter "Notification Manager" in *Inside Macintosh: Processes* for a description on how to send a notification request. You should also avoid calling routines like `InitWindows` that wipe the entire screen clean, obliterating any startup icons that other system extensions and drivers might have displayed.

Your system extension may only create files in the Preferences folder during execution. It is important that your system extension does not create files in the Extensions folder, the Control Panels folder, or the System Folder during execution. The system reads the files in each of these folders sequentially. Creating an additional file in one of these folders shifts the location of the other files, causing the system to either skip a system extension or execute one twice.

If your system extension requires a user interface, you can also create a control panel. If you use a system extension with your control panel, include it in the control panel file along with the required resources and any other optional resources you use. In System 7, system extensions can be installed in the Control Panels folder or in the Extensions folder (both of which are stored in the System Folder) or directly in the System Folder. However, if it contains a system extension, your control panel file must reside in the Controls Panels folder within the System Folder. At startup time, the system software opens files of type 'cdev' that reside in the Control Panels folder and executes any system extensions that it finds there. If the system extension portion of a control panel is not loaded at startup, the control panel won't function properly. For additional information about control panels, see the chapter Control Panels in *Inside Macintosh: More Macintosh Toolbox*.

Creating a System Extension's Resources

A file comprising a system extension contains a resource of type 'INIT' and additional resources. A resource of type 'INIT' contains the code that loads the system-level service into the system at system startup time, and it often contains the code that provides the system-level service. You can use additional resources to describe the icons for the system extension, specify a version number and copyright information for the information window displayed by the Get Info command, increase the size of the system heap, and more.

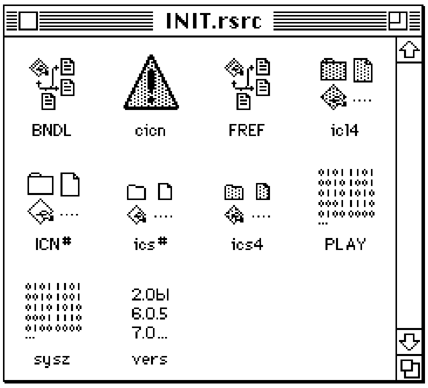
This list describes some of the additional resources you typically use when you create a system extension:

- The version ('vers') resource, which you can use to record version information for your system extension. The version resource allows you to store a version number, a version message, and a region code.
- The bundle ('BNDL') resource, which groups together your system extension's icons.
- Icon family resources ('ICN#', 'ics#', ic18', 'ic14', ics8', and 'ics4') to represent your system extension in the Finder.
- The system heap space ('sysz') resource.

The 'sysz' resource is described in this section. See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for additional information about the other resources mentioned in this section.

Figure 9-2 shows a ResEdit window containing additional resources for a system extension. These additional resources can be compiled with an 'INIT' resource into a system extension that goes in the Extensions folder.

Figure 9-2 Typical resources for a system extension



Not all of the resources in Figure 9-2 are required for all system extensions, but they do add useful features to a system extension.

Note
You can use a high-level tool such as the ResEdit application, which is available through APDA, to create your resources. See *ResEdit Reference* for details on using ResEdit. u

Creating Icons for a System Extension

You should provide two sets of icons for your system extension:

- an icon family for the file that contains your system extension
- an icon that your system extension displays at system startup time. This icon indicate whether the installation succeeded or failed

You should provide icon family resources for the file that contains your system extension. See the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials* for a detailed description of the icon family resources.

You can create a color icon resource of type 'cicn' for your system extension if you want to display a color startup icon at the bottom left of the screen. You can implement this feature by creating your own application-defined MyShowINIT procedure, or you can use a similar program called ShowInit. You can obtain the ShowInit program from various on-line services. (You can also contact APDA for further developer product information). To use ShowINIT, you pass the resource ID of your system extension’s 'cicn' resource to the ShowINIT procedure, and ShowINIT displays the 'cicn' icon on the bottom-left corner of the screen.

Creating a System Heap Zone Resource for a System Extension

You should read the information in this section only if you plan to install code from your system extension into the system heap and run your system extension on system software prior to System 7.

If you install code in the system heap and run your system extension on system software prior to System 7, you should include a system heap space resource of type 'sysz'. The 'sysz' resource tells the system software the amount of memory the system heap needs to expand by, in order to accommodate space for code installed by your system extension.

Note
It is not necessary to include a 'sysz' resource for system extensions running only on System 7 and later. The system heap in System 7 grows dynamically and expands as long as there is any unused RAM available. u

Using a 'sysz' resource, you can request the system software to increase the memory in the system heap by the amount specified in the 'sysz' resource. If the system software is able to allocate the needed memory in the system heap, your system extension will execute. If the system is unable to allocate the extra memory to the system heap, your system extension will not be able to execute.

To create a 'sysz' resource, you can use an editor like the ResEdit application. Specify, in bytes, the amount of memory you want the system heap to increase by. For example, if your system extension takes 8 KB to execute, you should increase the system heap by that amount.

You do not need to allocate memory for the actual system extension code ('INIT' resource), only for the amount of memory for any code installed by your system extension needs to execute.

Building a System Extension

Once you have created a file containing the 'INIT' resource and a file containing all the additional resources, you can build your system extension. To build a system extension, compile and link the 'INIT' resource and the additional resources into an executable file for your system extension.

When you compile the 'INIT' resource and your additional resources, you should keep the following points in mind:

- Make sure that the file type of the system extension is of type 'INIT'.
- Specify a creator if you want the Finder to use icons for your system extension.
- Specify the resource type 'INIT' and a resource ID (usually 128).
- Specify the main entry point for your system extension. When written in Pascal, the main entry point of a module is the first written instruction.
- Specify that the 'INIT' resource be loaded into the system heap if you want its resources to be available after the computer finishes booting.
- Specify the 'INIT' resource (code resource) as locked to prevent the system from moving the resource during execution.
- Make sure that all additional resources are unlocked and purgeable.

Start Manager Reference

This section describes the data structures and routines that are specific to the Start Manager. The “Data Structures” section explains the data structures for the default startup device parameter block, the default video device parameter block, and the default operating system record. The “Routines” section describes routines that get information about and set devices or values that the system uses as defaults when booting a Macintosh computer.

Data Structures

This section describes the data structures that you use to provide information to the Start Manager or the Start Manager uses to return information to your application.

The Default Startup Device Parameter Block

Two procedures, `GetDefaultStartup` and `SetDefaultStartup`, use the default startup device parameter block. You can use these procedures and the default startup device parameter block to get or set the default startup device. As defined by the `DefStartType` data type, a startup device is either a slot or a SCSI device. The `DefStartRec` data type defines the default startup device parameter block.

```
TYPE  DefStartType = (slotDev, scsiDev);

      DefStartRec =
RECORD
   CASE DefStartType OF
      slotDev:
         sdExtDevID: SignedByte; {external device ID}
         sdPartition:SignedByte; {reserved}
         sdSlotNum:  SignedByte; {slot number}
         sdSRsrcID:  SignedByte; {SResourceID}
      scsiDev:
         sdReserved1:SignedByte; {reserved}
         sdReserved2:SignedByte; {reserved}
         sdRefNum:   Integer;    {driver reference number}
END;
DefStartPtr = ^DefStartRec;
```

Field descriptions

`sdExtDevID`	The external device ID specified by a slot's driver. This ID identifies one of perhaps several devices connected through a single slot.
`sdPartition`	Reserved.
`sdSlotNum`	A number that identifies the location of the NuBus slot containing the default startup card. (Currently, these numbers range from $9 through $E on six-slot computers.)
`sdSRsrcID`	The resource ID (`SResourceID`) for the slot.
`sdReserved1`	Reserved.
`sdReserved2`	Reserved.
`sdRefNum`	A negative value in this field indicates the driver reference number for a SCSI device. A positive number indicates a slot device, in which case the fields in the `slotDev` variant.

The Default Video Device Parameter Block

Two procedures, `GetVideoDefault` and `SetVideoDefault`, use the default video device parameter block. You can use these procedures with the default video device parameter block to get or set the default video device. The `DefVideoRec` data type defines the default video device parameter block.

```
TYPE  DefVideoRec =
RECORD
   sdSlot:     SignedByte; {slot number}
   sdsResource:SignedByte; {SResourceID}
END;
DefVideoPtr = ^DefVideoRec;
```

Field descriptions

`sdSlot`	The physical slot number for the default video device. A value of 0 indicates no video device is the default.
`sdSResource`	The slot resource ID (`SResourceID`) for the default video device.

The Default Operating System Parameter Block

Two procedures, `GetDefaultOS` and `SetDefaultOS`, use the default operating system parameter block. You can use these procedures with the default operating system parameter block to get or set the default operating system. The `DefOSRec` data type defines the default operating system parameter block.

```
TYPE  DefOSRec =
RECORD
   sdReserved: SignedByte; {reserved}
```

```
   sdOSType:   SignedByte; {operating-system type}
END;
DefOSPtr = ^DefOSRec;
```

Field descriptions

`sdReserved` Reserved.

`sdOSType` A value identifying the operating system installed at startup. A 1 indicates the Macintosh Operating System. The numbers 0 through 15 are reserved.

Routines

This section describes the Start Manager routines you can use to identify and change the default startup device, the default video device, default operating system, and the default timeout value for the startup drive.

Many Start Manager routines specify a pointer to a parameter block as a parameter. For these routines, the routine description includes a list of the fields in the parameter block used by the routine. For each routine that uses a parameter block, information about the fields appears in the following format:

Parameter block

→	`input1`	`LongInt`	Input parameter comment.
←	`output1`	`LongInt`	Output parameter comment.

The arrow on the far left indicates whether the field is an input or output parameter. You must supply values for all input parameters. The routine returns values in the output parameters. The next column shows the field name as defined in the MPW interface files, followed by the data type of that field. This matches the MPW interface name of the data type as shown in the parameter block. The fourth column contains a comment about or a brief definition of the field.

Identifying and Setting the Default Startup Device

You can use the routines in this section to get information that identifies the default startup device or to supply information that sets a default startup device. These routines provide applications with the same capability that the Startup Disk control panel supplies for Macintosh users.

GetDefaultStartup

You can use the `GetDefaultStartup` procedure to return information about the default startup device.

```
PROCEDURE  GetDefaultStartup (paramBlock: DefStartPtr);
```

`paramBlock` A pointer to a default startup device parameter block.

Parameter block

←	sdExtDevID	SignedByte	External device ID.
←	sdPartition	SignedByte	Reserved.
←	sdSlotNum	SignedByte	Physical slot number.
←	sdSRsrcID	SignedByte	Slot resource ID (SResourceID).
←	sdReserved1	SignedByte	Reserved.
←	sdReserved2	SignedByte	Reserved.
←	sdRefNum	Integer	Driver reference number.

DESCRIPTION

The `GetDefaultStartup` procedure returns information about the default startup device from parameter RAM. The default startup device parameter block of data type `DefStartType` defines two kinds of startup devices: either a slot or a SCSI device. The `GetDefaultStartup` procedure returns in the `sdRefNum` field a value indicating the startup device type. A negative value indicates a SCSI device. A positive value indicates a slot device. If the value is negative, the `sdRefNum` field contains the driver reference number needed to identify that device. If the value is positive, the `slotDev` variant of the default startup device parameter block contains information about the slot device.

You cannot read the system's default startup device parameter block directly. Instead, create another parameter block to which the `GetDefaultStartup` procedure can write and pass `GetDefaultStartup` a pointer to that parameter block.

ASSEMBLY LANGUAGE INFORMATION

The registers on entry and exit for this routine are

Registers on entry

A0 Address of the default startup device parameter block

Registers on exit

A0 Address of the default startup device parameter block

SEE ALSO

For more information about the default startup device parameter block see "The Default Startup Device Parameter Block" beginning on page 9-18.
To specify the default startup device, see the description of the `SetDefaultStartup` procedure described next.

SetDefaultStartup

You can use the `SetDefaultStartup` procedure to write information to parameter RAM that specifies the default startup device.

```
PROCEDURE SetDefaultStartup (paramBlock: DefStartPtr);
```

`paramBlock` A pointer to a default startup device parameter block.

Parameter block for a slot device

→	sdExtDevID	SignedByte	External device ID.
→	sdPartition	SignedByte	Reserved.
→	sdSlotNum	SignedByte	Physical slot number.
→	sdSRsrcID	SignedByte	Slot resource ID (SResourceID).

Parameter block for a SCSI device

→	sdReserved1	SignedByte	Reserved.
→	sdReserved2	SignedByte	Reserved.
→	sdRefNum	Integer	Driver reference number.

DESCRIPTION

The `SetDefaultStartup` procedure writes information to parameter RAM that specifies the default startup device. The default startup parameter block of data type `DefStartType` defines two kinds of startup devices: either a slot or a SCSI device. To specify a slot device as the default, pass the external device ID, the slot number, and the slot resource ID. The external device ID, supplied by the slot's driver, identifies a particular device connected through that slot. It's possible that the card in this slot could have several devices connected to it.

To specify a SCSI device as the default, pass its driver reference number (always negative) in the `sdRefNum` field. To specify no device as the default, pass a value of 0 in this field.

ASSEMBLY LANGUAGE INFORMATION

The registers on entry and exit for this routine are

Registers on entry

A0 Address of the default startup device parameter block

Registers on exit

A0 Address of the default startup device parameter block

SEE ALSO

For more information about the default startup device parameter block see “The Default Startup Device Parameter Block” beginning on page 9-18.

To retrieve information about the default startup device, see the description of the `GetDefaultStartup` procedure described on page 9-20.

Identifying and Setting the Default Video Device

You can use the routines in this section to get information about the default video device or to supply information that sets or changes a default video device. These routines provide applications with the same capability that the Monitors control panel supplies for Macintosh users. The default video device is equivalent to the monitor that displays the startup message “Welcome to Macintosh” as well as other startup indications.

GetVideoDefault

You can use the `GetVideoDefault` procedure to return information that identifies the default video device.

```
PROCEDURE GetVideoDefault (paramBlock: DefVideoPtr);
```

`paramBlock` A pointer to a default video device parameter block.

Parameter block

←	`sdSlot`	`SignedByte`	Physical slot number.
←	`sdSResource`	`SignedByte`	Slot resource ID (SResourceID).

DESCRIPTION

The `GetVideoDefault` procedure returns information from parameter RAM that identifies the default video device. If the `sdSlot` field returns a 0, indicating no default video device, the Start Manager chooses the first available video device when the computer starts up.

ASSEMBLY LANGUAGE INFORMATION

The registers on entry and exit for this routine are

Registers on entry

A0 Address of the default video device parameter block

Registers on exit

A0 Address of the default video device parameter block

SEE ALSO

For more information about the default startup device parameter block see "The Default Video Device Parameter Block" beginning on page 9-19.

To specify the default video device, see the description of the `SetVideoDefault` procedure described next.

SetVideoDefault

You can use the `SetVideoDefault` procedure to write information to parameter RAM that sets or changes the default video device.

```
PROCEDURE SetVideoDefault (paramBlock: DefVideoPtr);
```

`paramBlock` A pointer to a default video device parameter block.

Parameter block

→	`sdSlot`	`SignedByte`	Physical slot number.
→	`sdSResource`	`SignedByte`	Slot resource ID (SResourceID).

DESCRIPTION

The `SetVideoDefault` procedure writes information to parameter RAM that sets or changes the default video device. If you set the `sdSlot` field to 0, indicating no default video device, the Start Manager chooses the first available video device when the computer starts up.

ASSEMBLY LANGUAGE INFORMATION

The registers on entry and exit for this routine are

Registers on entry

A0 Address of the default video device parameter block

Registers on exit

A0 Address of the default video device parameter block

SEE ALSO

For more information about the default video device parameter block see "The Default Video Device Parameter Block" beginning on page 9-19.

To retrieve information about the default video device, see the description of the `GetVideoDefault` procedure on page 9-23.

Identifying and Setting the Default Operating System

You can use the routines in this section to get information about the default operating system or to supply information that sets or changes a default operating system. These routines read from and write to a byte in parameter RAM.

GetOSDefault

You can use the `GetOSDefault` procedure to identify the operating system that gets booted on the Macintosh computer.

```
Procedure GetOSDefault (paramBlock: DefOSPtr);
```

`paramBlock` A pointer to a default operating system parameter block.

Parameter block

←	`sdReserved`	`byte`	Reserved.
←	`sdOSType`	`byte`	Operating-system type.

DESCRIPTION

The `GetOSDefault` procedure identifies the operating system that gets booted on the Macintosh computer. A value of 1 returned in the `sdOSType` field indicates the Macintosh Operating System. Apple Computer, Inc. reserves the numbers 0 through 15 for its use.

When the Macintosh Operating System boots, certain startup routines call `GetOSDefault` and compare the value it returns with the value in the `ddType` field of the driver's portion of the driver descriptor record. Each driver for the startup device has its own block of fields in this record. The startup routine tries to match the operating-system type returned by `GetOSDefault` with the value in one of the `ddType` fields. If it finds a match, the computer continues to boot; if it doesn't, the startup routine searches other drives attached to the computer. The boot process does not continue until the startup routine finds a `ddType` value that matches the one returned by `GetOSDefault`.

ASSEMBLY LANGUAGE INFORMATION

The registers on entry and exit for this routine are

Registers on entry

A0 Address of the default operating system parameter block

Registers on exit

A0 Address of the default operating system parameter block

SEE ALSO

For more information about the default operating system parameter block, see "The Default Operating System Parameter Block" beginning on page 9-19.
For information about the driver descriptor record, see the chapter "SCSI Manager" in *Inside Macintosh: Devices*.

To specify the default operating system, see the description of the `SetOSDefault` procedure described next.

SetOSDefault

You can use the `SetOSDefault` procedure to set a byte in parameter RAM that indicates the operating system that gets booted on the Macintosh computer.

```
PROCEDURE SetOSDefault (paramBlock: DefOSPtr);
```

`paramBlock` A pointer to a default operating system parameter block.

Parameter block

→	`sdReserved`	`SignedByte`	Reserved.
→	`sdOSType`	`SignedByte`	Operating-system type.

DESCRIPTION

The `SetOSDefault` procedure sets a byte in parameter RAM that indicates the operating system that gets booted on the Macintosh computer. Setting a value of 1 in the `sdOSType` field indicates the Macintosh Operating System, which is currently the only default operating system allowed. The numbers 0 through 15 are reserved by Apple Computer.

Unless the value in the `sdOSType` field matches the value in one of the `ddType` fields of the driver descriptor record, the computer cannot continue booting. Every drive connected to the computer has a driver descriptor record at the beginning of physical block 0.

ASSEMBLY LANGUAGE INFORMATION

The registers on entry and exit for this routine are

Registers on entry

A0 Address of the parameter block for the default operating system record

Registers on exit

A0 Address of the parameter block for the default operating system record

SEE ALSO

For information about the driver descriptor record, see the chapter "SCSI Manager" in *Inside Macintosh: Devices*.

Getting and Setting the Timeout Interval

You can use the routines in this section to get or set the default timeout interval for the startup drive. This timeout indicates how long the system waits for the startup drive to respond while the computer is booting.

GetTimeout

You can use the `GetTimeout` procedure to identify the current timeout interval set for the startup drive.

```
PROCEDURE GetTimeout (VAR count: Integer);
```

`count` Indicates the number of seconds the system waits for the startup drive to respond during the boot cycle. A value of 0 indicates the default timeout of 20 seconds.

DESCRIPTION

The `GetTimeout` procedure identifies the current timeout interval set for the startup drive. Timeout values increment in 1-second intervals, from 1 to a maximum of 31 seconds. A `count` of 1 equals 1 second.

ASSEMBLY LANGUAGE INFORMATION

The register on exit from the routine is

Registers on exit

A0 Value of `count` field

The `_GetTimeout` macro expands to invoke another trap macro, whose routine selector is passed in the A0 register.

Trap Macro	Selector
`_InternalWait`	$0000

SetTimeout

You can use the `SetTimeout` procedure to set the timeout interval for the startup drive.

```
PROCEDURE SetTimeout (count: Integer);
```

`count` Indicates the number of seconds that you want the system to wait for the startup drive to respond during the boot cycle. A value of 0 indicates the default timeout of 20 seconds. The maximum value is 31 seconds.

DESCRIPTION

The `SetTimeout` procedure sets the timeout interval for the startup drive. Timeout values increment in 1-second intervals, from 1 to a maximum of 31 seconds. Setting the `count` parameter to a value of 1 indicates 1 second.

ASSEMBLY LANGUAGE INFORMATION

The registers on entry for this routine are

Registers on entry

A0	$0001

The `_SetTimeout` macro expands to invoke another trap macro, whose routine selector is passed in the A0 register:

Trap Macro	Selector
`_InternalWait`	$0001

Summary of the Start Manager

Pascal Summary

Data Types

```
TYPE
   DefStartType = (slotDev, scsiDev);

   DefStartRec =
   RECORD
      CASE DefStartType OF
      slotDev:
         sdExtDevID: SignedByte;    {external device ID}
         sdPartition:SignedByte;    {reserved}
         sdSlotNum:  SignedByte;    {slot number}
         sdSRsrcID:  SignedByte;    {SResourceID}
      scsiDev:
         sdReserved1:SignedByte;    {reserved}
         sdReserved2:SignedByte;    {reserved}
         sdRefNum:   Integer        {driver reference number}
   END;

   DefStartPtr = ^DefStartRec;      {pointer to a start definition record}

   DefVideoRec =
   RECORD
      sdSlot:     SignedByte;       {slot number}
      sdsResource:SignedByte;       {SResourceID}
   END;

   DefVideoPtr = ^DefVideoRec;   {pointer to a video definition record}

   DefOSRec    =
   RECORD
      sdReserved: SignedByte;    {reserved--should be 0}
      sdOSType:   SignedByte;    {operating-system type}
```

```
   END;

   DefOSPtr    = ^DefOSRec;   {pointer to a default Operating System Record}
```

Routines

Identifying and Setting the Default Startup Device

```
PROCEDURE GetDefaultStartup (paramBlock: DefStartPtr);
PROCEDURE SetDefaultStartup (paramBlock: DefStartPtr);
```

Identifying and Setting the Default Video Device

```
PROCEDURE GetVideoDefault   (paramBlock: DefVideoPtr);
PROCEDURE SetVideoDefault   (paramBlock: DefVideoPtr);
```

Identifying and Setting the Default Operating System

```
PROCEDURE GetOSDefault      (paramBlock: DefOSPtr);
PROCEDURE SetOSDefault      (paramBlock: DefOSPtr);
```

Getting and Setting the Timeout Interval

```
PROCEDURE GetTimeout        (VAR count: Integer);
PROCEDURE SetTimeout        (count: Integer);
```

C Summary

Data Types

```
struct SlotDev {
   char sdExtDevId;     /*external device ID*/
   char sdPartition;    /*reserved*/
   char sdSlotNum;      /*slot number*/
   char sdSRsrcID;      /*SResourceID*/
};

typedef struct SlotDev SlotDev;

struct SCSIDev {
   char sdReserved1;    /*reserved*/
   char sdReserved2;    /*reserved*/
```

```
   short sdRefNum;      /*driver reference number*/
};

typedef struct SCSIDev SCSIDev;

union DefStartRec {
   SlotDev slotDev;
   SCSIDev scsiDev;
};

typedef union DefStartRec DefStartRec;
typedef DefStartRec *DefStartPtr;

struct DefVideoRec {
   char sdSlot;      /*slot number*/
   char sdsResource; /*SResourceID*/
};

typedef struct DefVideoRec DefVideoRec;
typedef DefVideoRec *DefVideoPtr;

struct DefOSRec {
   char sdReserved;  /*reserved —should be 0*/
   char sdOSType;    /*operating-system type*/
};

typedef struct DefOSRec DefOSRec;
typedef DefOSRec *DefOSPtr;
```

Routines

Identifying and Setting the Default Startup Device

```
pascal void GetDefaultStartup (DefStartPtr paramBlock);
pascal void SetDefaultStartup (DefStartPtr paramBlock);
```

Identifying and Setting the Default Video Device

```
pascal void GetVideoDefault (DefVideoPtr paramBlock);
pascal void SetVideoDefault (DefVideoPtr paramBlock);
```

Identifying and Setting the Default Operating System

```
pascal void GetOSDefault    (DefOSPtr paramBlock);
```

```
pascal void SetOSDefault    (DefOSPtr paramBlock);
```

Getting and Setting the Timeout Interval

```
pascal void GetTimeout      (short *count);
pascal void SetTimeout      (short count);
```

Assembly-Language Summary

Data Structures

Default Startup Device Data Structure

0	sdExtDevID	byte	external device ID
1	sdPartition	byte	reserved
2	sdSlotNum	byte	slot number
3	sdSRsrcID	byte	slot resource ID

0	sdReserved1	byte	reserved
1	sdReserved2	byte	reserved
2	sdRefNum	word	driver reference number

Default Video Device Data Structure

0	sdSlot	byte	slot number
1	sdSResource	byte	slot resource ID

Default Operating System Data Structure

0	sdReserved	byte	reserved
1	sdOSType	byte	operating-system type

Trap Macros

Trap Macros Requiring Register Setup

Trap macro name	Registers on entry	Registers on exit
`_GetDefaultStartup`	A0: address of default video device parameter block	A0: address of default startup device parameter block
`_SetDefaultStartup`	A0: address of default video device parameter block	A0: address of default startup device parameter block
`_GetVideoDefault`	A0: address of default video device parameter block	A0: address of default video device parameter block
`_SetVideoDefault`	A0: address of default video device parameter block	A0: address of default video device parameter block
`_GetDefaultOS`	A0: address of default operating system parameter block	A0: address of default operating system parameter block
`_SetDefaultOS`	A0: address of default operating system parameter block	A0: address of default operating system parameter block
`_GetTimeout`		D0: count (word)
`_SetTimeout`	D0: count (word)	

Trap Macros Requiring Routine Selectors

`_InternalWait`

Selector	Routine
$0000	`GetTimeout`
$0001	`SetTimeout`

Global Variables

`TimeDBRA`	The number of times the DBRA instruction is executed per millisecond.
`TimeSCCDB`	The number of times the SCC is accessed per millisecond.
`TimeSCSIDB`	The number of times the SCSI is accessed per millisecond.

Package Manager

Contents

This chapter describes the Package Manager, the part of the system software that loads packages into memory. The packages include one 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.

Read the information in this chapter to get a complete list of all packages and to get a description of the Package Manager routines that load the packages into memory.

Ordinarily, you do not need to use the Package Manager routines described in this chapter. The Operating System itself is responsible for installing the packages when an application is launched. While your application probably won't ever need to use these routines, for the sake of completeness they are described in this chapter.

About the Package Manager

The Package Manager lets you load packages into memory. A **package** is a set of routines and data types that is stored as a resource of type 'PACK'. In early models of the Macintosh computer, all packages were disk-based and brought into memory only when needed; some packages are now in ROM. The System file contains the standard Macintosh packages and the resources they use or own. Table 10-1 lists the standard Macintosh packages.

Table 10-1 The standard Macintosh packages

Package	Description	Resource ID
List Manager	Provides routines that your application can use to create scrollable lists that allow the user to select one or more of a group of items.	0
Disk Initialization Manager	Provides routines that initialize and name new floppy disks. This package is called by the Standard File Package and applications.	2
Standard File Package	Provides routines that your application can use to display dialog boxes that let the user specify the locations of files to be saved or opened.	3
Floating-Point Arithmetic Package	Provides routines that support extended-precision arithmetic according to IEEE Standard 754.	4

continued

Table 10-1 The standard Macintosh packages (continued)

Package	Description	Resource ID
Transcendental Functions Package	Provides routines that support trigonometric, logarithmic, exponential, and financial functions, and a random number generator.	5
Text Utilities (formerly referred to as the International Utilities Package)	Provides routines that your application can use to specify strings for various purposes, to format numbers and currency, format date and time, search and replace text, and more.	6
Text Utilities (formerly referred to as the Binary-Decimal Conversion Package)	Provides routines that your application can use to specify strings for various purposes, to format numbers and currency, format date and time, search and replace text, and more.	7
Apple Event Manager	Provides routines that your application can use to respond, send, and record Apple events.	8
PPC Browser	Provides routines that your application can use to display the program linking dialog box, which allows a user to select a port to communicate with.	9
Edition Manager	Provides routines that your application can use to allow users to share and automatically update data and numerous documents and applications.	11
Color Picker	Provides routines that your application can use to display a standard dialog box for choosing a color, and converts color specifications from one color model to another.	12

Table 10-1 The standard Macintosh packages (continued)

Package	Description	Resource ID
Data Access Manager	Provides routines that your application can use to gain access to data in another application, and provides templates to be used for data transactions.	13
Help Manager	Provides routines that your application can use to provide Balloon Help online assistance.	14
Picture Utilities	Provides routines that obtain qualitative and quantitative information about pictures and pixel maps.	15

If the Package Manager is not able to load a package, the Package Manager adds the resource ID number of the affected package to 17 to get an error number. The System Error Handler uses this error number to display an error message. Originally this approach worked because there were only 7 packages, and the error number would fall between 17 and 24, which are the error numbers that define the "Can't load package" error. However, now there are more packages and the resulting error messages from packages with resource IDs greater than 7 are misleading.

The error messages that corresponds to packages with resource IDs greater than 7 are as follows:

Resource ID	Package	Error ID	Error
9	Apple Event Manager	25	Out of memory
9	PPC Toolbox	26	Can't launch file
11	Edition Manager	28	Stack overflow
12	Color Picker	29	*
13	Data Access Manager	30	Disk insertion required
14	Help Manager	31	Wrong disk inserted
15	Picture Utilities	32	*

* There is not a defined system error for this error ID.

The system errors are described in detail in the chapter "System Error Handler" in this book.

Using the Package Manager

The Package Manager provides two routines: the InitPack procedure and the InitAllPacks procedure. The InitPack procedure loads one specified package into memory. To specify which package to load, you pass, as a parameter to the InitPack procedure, the package's resource ID. You can use the InitAllPacks procedure to load all packages into memory. Typically, you do not need to use either of these two procedures because the InitAllPacks procedure is automatically called when your application is launched.

The InitPack and InitAllPacks procedures do not initialize the packages. Consult the description of the specific package to see if it needs to be initialized before your application can utilize all of its routines. For example, to use the Data Access Manager routines, your application must first call the InitDBPack function (an initialization routine provided by the Data Access Manager). If a package needs to be initialized, it provides an initialization routine.

Note

You can access a routine in a package through a trap macro and a routine selector. The name of the trap macro includes the word "Pack" and the resource ID of the specific package. For example, the trap macro for the routines in the Edition Manager is _Pack11. Most system software routines that are accessed through a trap macro and a routine selector also have a corresponding macro that expands to call the original trap macro and automatically puts the correct routine selector on the stack. For example, to access the Standard File Package routine StandardGetFile, you can call the _StandardGetFile macro. The _StandardGetFile macro then expands to call the _Pack3 trap macro and places the routine selector on the stack (in this example the routine selector is $0006). See the chapter "Trap Manager" in this book for more information about trap macros and routine selectors. u

Package Manager Reference

This section describes routines that are specific to the Package Manager.

Routines

This section describes the two routines in the Package Manager. One routine lets you load a specified package into memory, and one routine lets you load all packages into memory.

Initialization of Packages

You use the routines in this section to load one specified package or all packages into memory.

InitPack

You can use the `InitPack` procedure to load a specified package into memory.

```
PROCEDURE  InitPack (packID: Integer);
```

`packID` A package resource ID.

DESCRIPTION

The `InitPack` procedure loads the package specified by the `packID` parameter into memory. The `packID` parameter is the package's resource ID. To initialize a specific package or manager, consult the documentation of the specific package or manager.

InitAllPacks

You can use the `InitAllPacks` procedure to load all packages into memory.

```
PROCEDURE InitAllPack;
```

DESCRIPTION

The `InitAllPacks` procedure loads all the packages into memory. The `InitAllPacks` procedure is automatically called when your application is launched.

Summary of the Package Manager

Pascal Summary

Constants

```
CONST
   listMgr     = 0;     {List Manager}
   dskInit     = 2;     {Disk Initialization Manager}
   stdFile     = 3;     {Standard File Package}
   flPoint     = 4;     {Floating-Point Arithmetic Package}
   trFunc      = 5;     {Transcendental Functions Package}
   textUtil1   = 6;     {Text Utilities}
   textUtil2   = 7;     {Text Utilities}
   aevtMgr     = 8;     {Apple Event Manager}
   ppcBrowser  = 9;     {PPC Browser}
   editionMgr  = 11;    {Edition Manager}
   colorPicker = 12;    {Color Picker}
   dataAccess  = 13;    {Data Access Manager}
   helpMgr     = 14;    {Help Manager}
   pictUtil    = 15;    {Picture Utilities}
   intUtil     = 6;     {Text Utilities}
   bdConv      = 7;     {Text Utilities}
```

Routines

Initializing Packages

```
PROCEDURE InitPack (packID: Integer);
PROCEDURE InitAllPacks;
```

C Summary

Constants

```
enum {
   listMgr     = 0,     /*List Manager*/
   dskInit     = 2,     /*Disk Initialization Manager*/
   stdFile     = 3,     /*Standard File Package*/
   flPoint     = 4,     /*Floating-Point Arithmetic Package*/
   trFunc      = 5,     /*Transcendental Functions Package*/
   textUtil1   = 6,     /*Text Utilities*/
   textUtil2   = 7,     /*Text Utilities*/
   aevtMgr     = 8,     /*Apple Event Manager*/
   ppcBrowser  = 9,     /*PPC Browser*/
   editionMgr  = 11,    /*Edition Manager*/
   colorPicker = 12,    /*Color Picker*/
   dataAccess  = 13,    /*Data Access Manager*/
   helpMgr     = 14,    /*Help Manager*/
   pictUtil    = 15,    /*Picture Utilities*/
   intUtil     = 6,     /*Text Utilities*/
   bdConv      = 7      /*Text Utilities*/
};
```

Routines

Initializing Packages

```
pascal void InitPack        (short packID);
pascal void InitAllPacks    (void);
```

Assembly-Language Summary

Trap Macros

Trap Macros Requiring Routine Selectors

```
_Pack0     ;List Manager
_Pack2     ;Disk Initialization Manager
_Pack3     ;Standard File Package
_Pack6     ;Text Utilities
_Pack7     ;Text Utilities
_Pack8     ;Apple Event Manager
_Pack9     ;PPC Browser
_Pack11    ;Edition Manager
_Pack12    ;Color Picker
_Pack13    ;Data Access Manager
_Pack14    ;Help Manager
_Pack15    ;Picture Utilities
```

Glossary

A-line instruction An unimplemented instruction of the form $A*xxx* (the high-order 4 bits have the hexadecimal value $A).

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.

auto-pop bit Bit 10 of a Toolbox trap word, signifying that an extra return address is placed on the stack.

bit The atomic memory unit. Each bit can be either set (the value of the bit is 1) or cleared (the value of the bit is 0).

bomb box See **system error alert box.**

boot blocks The first two logical blocks on every Macintosh volume. Boot blocks contain instructions and information necessary to start up (or "boot") a Macintosh computer.

byte A bit quantity, used to store 2^8, or 256, different possible values. In the MC680x0 bit-numbering scheme, the first bit in a byte is bit number 7, and the last bit is bit number 0. See also **reversed bit-numbering**.

caret A generic term meaning a symbol that indicates where something should be inserted in text. The specific symbol used is a vertical bar(|).

caret-blink time The interval between blinks of the caret that marks an insertion point.

clock chip A special integrated circuit (IC) used for storing parameter RAM and the current date and time. This IC is powered by a battery when the system is off, thus keeping correct time and preserving the parameter RAM information.

come-from patch A system software patch used only by Apple to add enhancements to system software. Come-from patches are placed before any other types of patches in a patch daisy chain.

control panel A modeless dialog box that contains controls that let users specify basic settings and preferences for a systemwide feature, such as the speaker volume, desktop pattern, or picture displayed by a screen saver.

control panel extension A collection of routines that manages a certain part of a control panel's display area.

daisy chain A chain of any number of patches and one system software routine.

dangling reference Typically, a pointer whose target has been either destroyed or moved elsewhere in memory.

date-time record A data structure that represents date and time as a record rather than a 32-bit long integer. The date-time record is a translation of the standard date-time value, so it can represent only dates and times between midnight on January 1, 1904 and 6:28:15 A.M. on February 6, 2040.

default operating system The operating system that gets initialized and booted on a Macintosh computer. Currently, the only default operating system allowed is the Macintosh Operating System.

default startup device The first drive on which the boot code attempts to start up the Macintosh Operating System.

default video device The first monitor on which the system displays the startup message "Welcome to Macintosh." and other startup indications.

double-click time The greatest interval between a mouse-up and mouse-down event that would qualify two mouse clicks as a double-click.

environmental selector A Gestalt selector code, used with the `Gestalt` function, that returns information about the operating environment that can be used by an application to guide its actions. Compare **informational selector.**

exception Any of various situations in which the normal flow of execution of a program is interrupted, with control passing to a system exception handler.

exception handler A system routine invoked automatically by the processor in any of a variety of exceptional circumstances. For example, the trap dispatcher is an exception handler that is called by the processor, to dispatch unimplemented A-line instructions.

exception stack frame A block of data placed on the stack automatically by the processor when an exception occurs.

extended parameter RAM The 236 bytes of parameter RAM that is reserved by the system software.

fatal system error A system error that causes the entire system to crash.

Gestalt Manager The part of the Macintosh Operating System that you can use to determine the features of the current software and hardware operating environment.

glue routine A runtime library routine, usually provided by the development environment, that provides a linkage between high-level language code and a system routine with an interface protocol different from that of the high-level language.

head patch A patch that, upon completion does not regain control. A head patch jumps to the next routine. Compare **tail patch.**

high-order bit The bit contributing the greatest value in a string of bits. For example, in the MC680x0 numbering scheme bit number 7 contributes a value of 2^7, or 128. Same as **most significant bit.** Compare **low-order bit**.

informational selector A Gestalt selector code, used with the `Gestalt` function, that supplies information about the operating environment that cannot be used to determine whether a software or hardware feature is available. Compare **environmental selector.**

least significant bit The bit contributing the least value in a string of bits. For example, in the MC680x0 numbering scheme bit number 0 in a byte contributes a value of 2^0, or 1. Same as **low-order bit**. Compare **most significant bit**

long date-time record A data structure that represents date and time as a record rather than a 64-bit long integer.

long date-time value A 64-bit integer in SANE `comp` format that represents date and time purely in seconds. This format allows dates and times before and after the range of the date-time record (30,000 B.C. to 30,000 A.D.).

long word A 32-bit quantity used to store 2^{32} (or 4,294,967,296) values.

long-word boundary The memory location that divides two long words.

low-order bit The bit contributing the least value in a string of bits. For example, in the MC680x0 numbering scheme bit number 0 in a byte contributes a value of 2^0, or 1. Same as **least significant bit**. Compare **high-order bit**.

MC680x0 bit-numbering The bit-numbering scheme used by Motorola. Bit numbers are counted from right to left. (That is, the most significant bit has the highest bit number, and the least significant bit number has the lowest bit number). Compare **reversed bit-numbering.**

menu-blink time The number of times a menu item blinks when the user chooses it.

mouse-down event An event indicating that the user pressed the mouse button.

most significant bit The bit contributing the greatest value in a string of bits. For example, in the MC680x0 numbering scheme bit number 7 in a byte contributes a value of 2^7, or 128. Same as **high-order bit**. Compare **least significant bit**.

mouse scaling A feature that causes the cursor to move twice as far during a mouse stroke as it would have otherwise, provided the change in the cursor's position exceeds the mouse-scaling threshold within one tick after the mouse is moved.

mouse-scaling threshold A number of pixels that, if exceeded by the sum of the horizontal and vertical changes in the cursor's position during one tick of mouse movement, causes mouse scaling to occur (if that feature is turned on); normally six pixels.

mouse-up event An event indicating that the user released the mouse button.

operating-system queue A queue used by the Macintosh Operating System.

Operating System trap An exception that is caused by an A-line instruction that executes an Operating System routine**.**

Operating System trap dispatch table A table in RAM containing addresses of Operating System routines.

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

Package Manager A set of routines that loads the packages into memory.

pad byte The extra byte added to make 2 bytes, when you declare a variable of type Byte.

panel The area managed by a control panel extension. A panel contains controls and other dialog items related to the features managed by control panel extensions.

parameter RAM Battery-powered RAM (random-access memory) contained in the clock chip, which preserves settings such as those made with the control panels. Parameter RAM takes up 256 bytes of battery-powered RAM: 20 bytes are commonly accessible by applications, and 236 bytes are reserved by the system software. See also **clock chip**.

patch Generally, any code used to repair or augment an existing piece of code. In the context of system software, a patch repairs or augments a system software routine. See also **head patch**, **tail patch**, and **come-from patch.**

pseudo-random number generator An algorithm that is designed to return a value that is as random as possible.

queue A list of identically structured entries linked together by pointers.

queue element A data structure that contains a pointer to the next queue element in the queue, a value indicating the queue type, and a variable data field.

queue header A data structure that contains flags specific to the queue, a pointer to the first element in the queue, and a pointer to the last element in the queue.

Queue Utilities The collection of routines for directly adding a queue element to a queue or directly removing a queue element from a queue.

resume procedure A procedure within an application that allows the application to recover from system errors**.**

reversed bit-numbering A bit-numbering scheme opposite that of the MC680x0 numbering scheme. Bit numbers are counted from left to right instead of right to left. For example, using the reversed bit-numbering scheme on a byte, the first bit is bit number 0 and the last bit is bit number 7. (That is, the most significant bit has the lowest bit number, and the least significant bit number highest bit number). Compare **MC680x0 bit-numbering.**

selector See **selector code.**

selector code A parameter to the `Gestalt` function that specifies what information about the operating environment the caller requires. See **environmental selector** and **informational selector.**

selector function A function that is executed when an application calls `Gestalt` and passes the associated **selector code**.

standard date-time value A 32-bit long integer that represents date and time purely in seconds. The standard date-time value can track dates and times only between midnight on January 1, 1904 and 6:28:15 A.M. on February 6, 2040.

Start Manager A collection of routines that let you get and set system startup information located parameter RAM.

system environment record A description of the operating environment filled in by the `SysEnvirons` function and defined by the `SysEnvRec` data type.

system error An error generated by the Operating System.

system error alert box An alert box displayed by the System Error Handler when a system error has occurred.

system error alert table resource A resource that determines the appearance and function of system error alert boxes and system startup alert boxes.

System Error Handler The part of the Operating System that displays an alert box when an system error occurs and manages display of the “Welcome to Macintosh” alert box at system startup time.

system error ID An ID number that may appear in a system error alert box to identify the error.

system extension A file (with the file type 'INIT') containing a code resource of type 'INIT' and additional other resources. A system extension typically contains code that performs a system-level service and code that loads this system-level service into the system at system startup time.

system initialization The process when the system initialization code located in ROM is executed. Memory is tested and initialized, ROM drivers are installed, device drivers are located, and more.

system startup The process when the system startup code located in ROM is executed. Memory is tested and initialized, ROM drivers are installed, device drivers are located, and more.

system startup alert box The alert box displayed at system startup time. It contains the startup greeting “Welcome to Macintosh.”

system startup information Configurable system parameters and machine-language instructions needed to start up a Macintosh computer.

tail patch A patch that transfers control to routine, and then regains control after the routine completes execution. Compare **head patch.**

timeout interval The interval of time the system waits for the startup drive to respond while the computer is booting.

Toolbox trap An exception that is caused by an A-line instruction that executes a Toolbox routine**.**

Toolbox trap dispatch table A table in RAM that contains addresses to Toolbox routines.

trap An exception caused by an A-line instruction.

trap dispatcher The exception handler that deals with the occurrence of A-line instructions.

trap dispatch table A table of entry points to system routines that are invoked with A-line instructions. Compare **Operating System trap dispatch table** and **Toolbox trap dispatch table**.

Trap Manager A collection of routines that lets you add extra capabilities to system software routines.

trap number The bits of a trap word (bits 0–7 for an Operating System routine, bits 0–9 for a Toolbox routine) that serve as an index into the trap dispatch tables.

trap word See A-line instruction

vertical retrace interrupt An interrupt generated 60 times a second by the Macintosh video circuitry while the beam of the display tube returns from the bottom of the screen to the top; also known as *vertical blanking interrupt*.

word A 16-bit quantity, used to store 2^{16} (or 65,536) possible values.

word boundary The memory location that divides two words.

Index

Numerals

A

B

C

D

E

F

G

H

I, J

K

L

M

N

O

P

Q

R

S

W

X, Y

Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Pro 630 printer. Final page negatives were output directly from text files on an Optrotech SPrint 220 imagesetter. Line art was created using Adobe Illustrator™ and Adobe Photoshop™. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

LEAD WRITER
Sharon Everson

WRITERS
Ulla Hald, Tim Monroe,
Michael Abramowicz

DEVELOPMENTAL EDITORS
George Truett, Antonio Padial,
Laurel Rezeau

ILLUSTRATORS
Bruce Lee, Ruth Anderson

COVER DESIGNER
Barb Smyth

PRODUCTION EDITOR
Gerri Gray

PROJECT MANAGER
Trish Eastman

Special thanks to Tony Francis,
Jim Mensch, Alex Rosenberg

Acknowledgments to Sam Barone,
Ray Chiang, Lorraine Findlay,
Carl Hewitt, Nick Kledzik, Jim Luther,
Sue Luttner, Joseph Maurer,
Josephine Manuele, Brian McGhie,
Martin Minow, and the entire
Inside Macintosh team.