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.

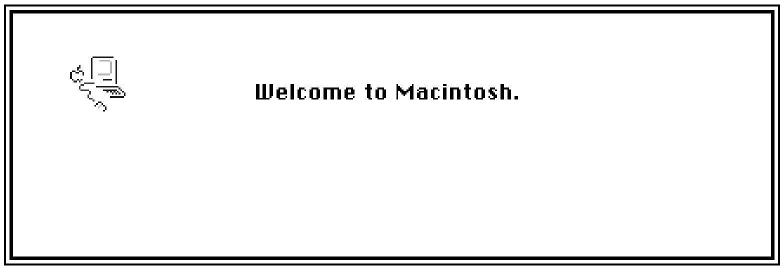
About the System Error Handler

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.

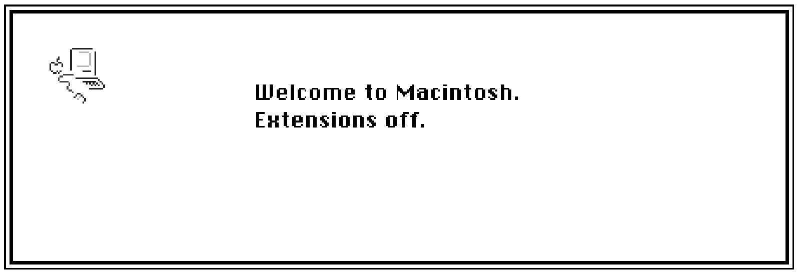
Figure 2-1 The system startup alert box



Welcome to Macintosh.

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

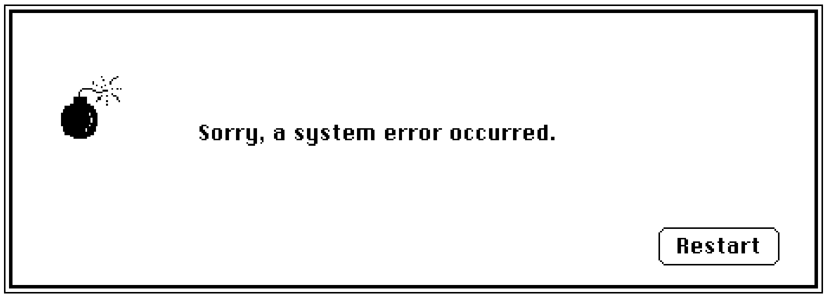


Welcome to Macintosh.
Extensions off.

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


Sorry, a system error occurred.

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. ◆

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)

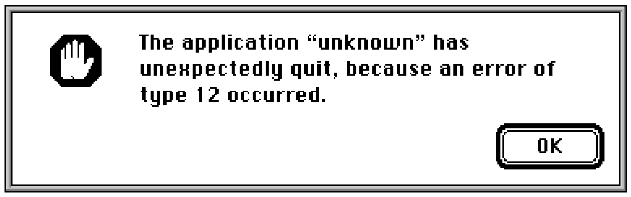
ID and name	Explanation
33 (Negative `zcbFree` value)	The Memory Manager's calculation of the number of bytes free in a heap zone (that is, the value of the `zcbFree` field) resulted in a negative number. Your application might have used up too much memory in the heap zone, or the heap is corrupted
41 (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


The application "unknown" has unexpectedly quit, because an error of type 12 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.

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.

▲ **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. ▲

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

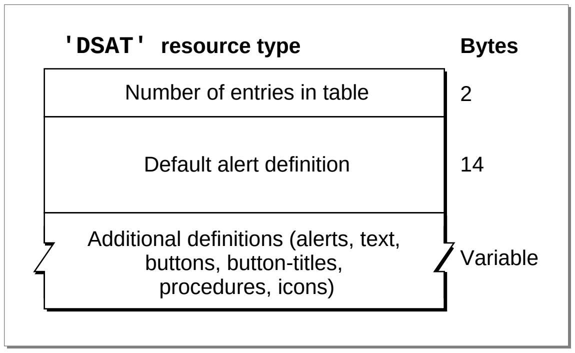
The System Error Handler stores system error alert tables in resources with resource type `'DSAT'`. During system startup, the system error alert table resource with resource ID 0 is loaded. This resource describes the "Welcome to Macintosh" alert box. Immediately thereafter, that table is disposed of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

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.

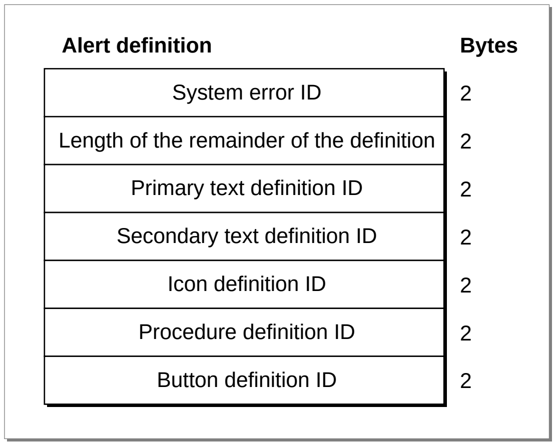
Figure 2-5 The structure of a system error alert table


'DSAT' resource type
Bytes
Number of entries in table
2
Default alert definition
14
Additional definitions (alerts, text, buttons, button-titles, procedures, icons)
Variable

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



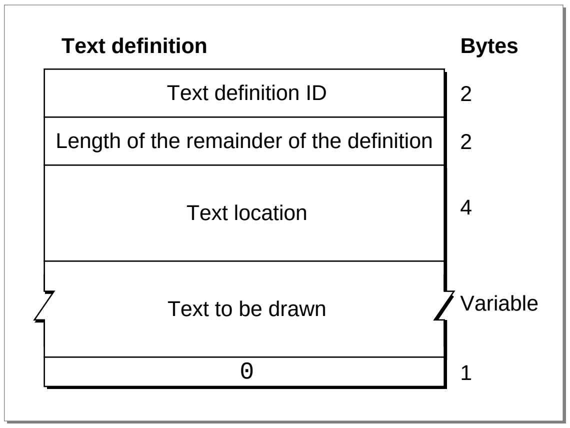
Alert definition	**Bytes**
System error ID	2
Length of the remainder of the definition	2
Primary text definition ID	2
Secondary text definition ID	2
Icon definition ID	2
Procedure definition ID	2
Button definition ID	2

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.

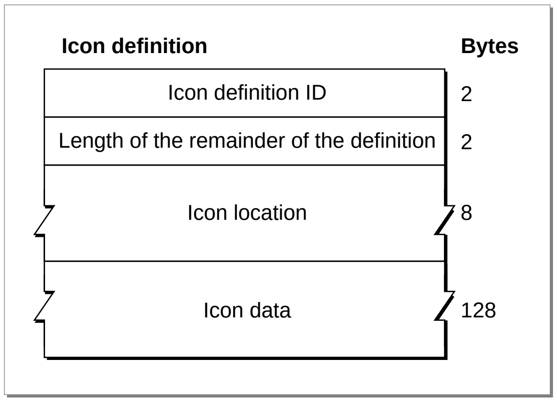
Figure 2-7 The structure of a text definition



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.

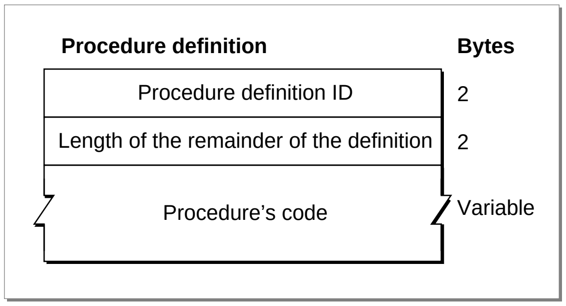
Figure 2-8 The structure of an icon definition



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.

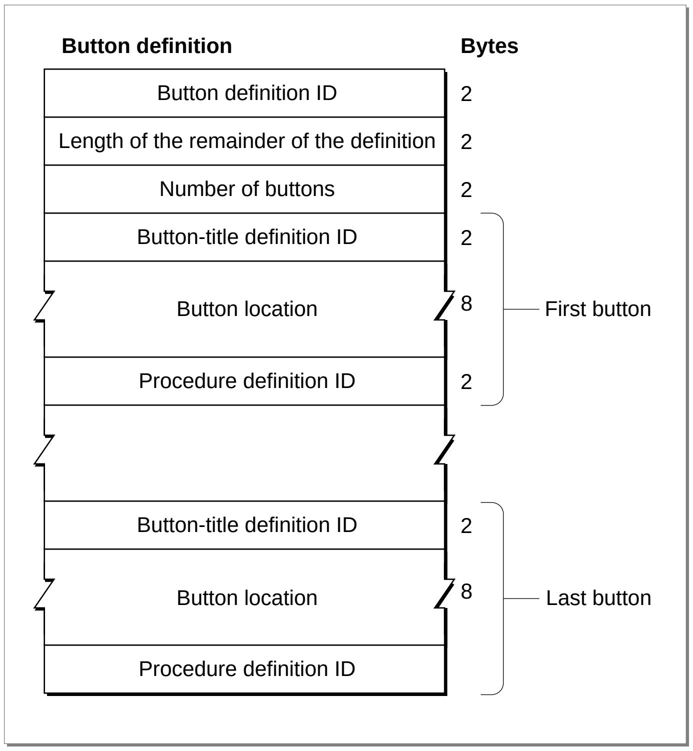
Figure 2-9 The structure of a procedure definition



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.

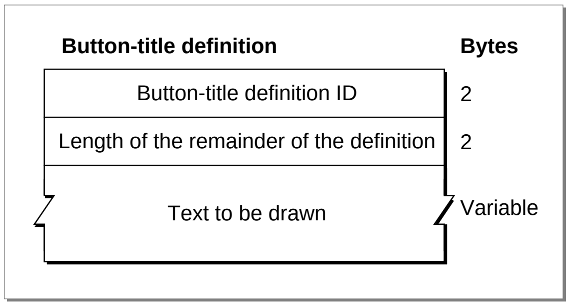
Figure 2-10 The structure of a button definition



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


Button-title definition
Bytes
Button-title definition ID
2
Length of the remainder of the definition
2
Text to be drawn
Variable

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.