This chapter describes the Disk Initialization Manager, the part of the Operating System that allows you to initialize disks and erase the contents of previously initialized disks. The Disk Initialization Manager provides a routine that allows you to present the standard user interface for initializing and naming disks. It also provides routines that allow you to initialize disks without presenting that standard user interface.

You need to read this chapter if your application does not mask out disk-inserted events. When your application receives a disk-inserted event, it must determine whether the inserted disk is valid. If the disk is not valid, your application can use the Disk Initialization Manager to present the user with the standard interface for initializing the disk.

To use this chapter, you should already be familiar with the Event Manager, which sends your application a disk-inserted event whenever a disk is inserted (unless you have masked out such events). You need to examine the `message` field of that event to determine whether the inserted disk is already initialized. You also need to be familiar with the File Manager if your application changes the default volume characteristics of newly initialized volumes.

This chapter begins by describing the operation of the Disk Initialization Manager, including

- formatting, verifying, and zeroing a disk
- the standard user interface for initializing and naming a disk
- bad block sparing

Then this chapter shows how you can

- determine whether an inserted disk is valid
- present the standard user interface to initialize and name an invalid disk
- present the standard user interface to erase a disk
- initialize or erase a disk without using the standard user interface
- change the default volume characteristics of newly initialized volumes

About the Disk Initialization Manager

The Disk Initialization Manager is the part of the Macintosh Operating System that manages the process of initializing disks. This package accepts requests to initialize a disk and translates them into control calls for the corresponding disk driver. The Disk Initialization Manager itself does not perform the low-level formatting or verification of the disk; instead, it simply manages the communication between the software requesting that a particular disk be initialized and the appropriate disk driver.

Note
In theory, you can use the Disk Initialization Manager to initialize any writable disk drive. In practice, however, most SCSI disk drivers ignore formatting control calls. Instead, low-level disk operations such as formatting and verification are usually performed by a utility program supplied with the disk. As a result, this chapter assumes that the disk to be initialized is a 3.5-inch floppy disk or an Apple Hard Disk 20SC, all of which are accessed through the Disk Driver. ◆

Usually, the Finder or the Standard File Package calls the Disk Initialization Manager when the user inserts an uninitialized disk. Occasionally the user will insert a disk when your application is frontmost. At that time, the Operating System generates a disk-inserted event. If your application has not masked out such events, it receives an event record for that event when it makes an event call and no events with higher priority are pending. You then need to determine whether the inserted disk is valid (as indicated by a value in the event record). If the disk is not valid, you should call the Disk Initialization Manager to allow the user to initialize the disk or, if desired, eject it.

If your application masks out disk-inserted events, the event stays in the event queue until your application calls the Standard File Package (which automatically processes disk-inserted events) or until the current application can handle disk-inserted events. In general, it's best not to mask out disk-inserted events and to handle them as described later in this chapter; otherwise, the user is likely to become confused when, after inserting an uninitialized or damaged disk, no disk icon appears on the desktop and no standard disk initialization dialog box appears. (Icons of initialized and undamaged disks always appear on the desktop, even if the current application ignores disk-inserted events.)

Disk Initialization

Disk initialization is the process of making a disk usable by the Macintosh Operating System. When shipped, most floppy disks are uninitialized because different operating systems have different initialization requirements. On Macintosh computers, disk initialization consists of three independent steps:

- disk formatting
- disk verification
- disk zeroing

All three steps must be performed successfully before the disk is considered initialized (or valid). You can use a single Disk Initialization Manager function, `DIBadMount`, to perform all three operations in sequence, or you can perform any one of them by calling a corresponding low-level function (either `DIFormat`, `DIVerify`, or `DIZero`). In general, your application should use the standard user interface described in the following section to initialize a disk.

The first step in the initialization process is **disk formatting.** Formatting a disk consists of writing special information onto a disk so that the disk driver can read from and write to the disk. This involves dividing the total usable space into sectors and tracks. See the chapter "Disk Driver" in *Inside Macintosh: Devices* for a description of how a disk is divided into tracks and sectors.

The second step in the disk-initialization process is **disk verification.** Verifying a disk consists of reading every bit on the disk to ensure that the disk has been formatted correctly and contains no bad blocks. If an error occurs during the reading of any single bit, the verification is considered unsuccessful.

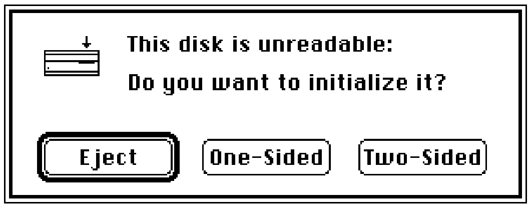
The third and final step in the disk-initialization process is **disk zeroing.** Zeroing a disk consists of creating on the disk the data structures and files necessary for the disk to be recognized as a hierarchical file system (HFS) volume. In particular, zeroing a disk places a master directory block (MDB), a volume bitmap, and a catalog file in appropriate locations on the disk. (For information on the locations and sizes of these items, see the description of the organization of data in a volume in the chapter "File Manager" in this book.) The volume bitmap and catalog file are set up to represent a volume containing no user files. As a result, zeroing a disk makes any files previously located on the disk inaccessible.

Beginning in system software version 7.0, zeroing a disk also causes the Disk Initialization Manager to attempt to remove any bad blocks (as identified during the disk-verification process) from the pool of available blocks on the disk. See "Bad Block Sparing" on page 5-7 for a description of this capability.

The Disk Initialization User Interface

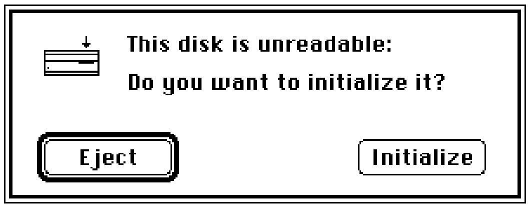
The Finder and the Standard File Package both handle disk-inserted events for uninitialized disks by presenting a **disk initialization dialog box** asking the user whether the disk should be ejected or initialized. Your application too can easily call a Disk Initialization Manager routine that generates such a dialog box when the user inserts an invalid disk. Figure 5-1 illustrates one configuration of the dialog box.

Figure 5-1 The disk initialization dialog box


This disk is unreadable:
Do you want to initialize it?

The appearance of the disk initialization dialog box changes to reflect changing conditions. For example, the icon changes to show which drive contains the disk. Also, the text of the dialog box changes according to what is wrong with the disk. The text might read "This is not a Macintosh disk" if the Disk Initialization Manager detects that the disk has been formatted for use on another operating system. Or, it might notify the user that a high-density disk can be used only on an Apple SuperDrive. Finally, if a user inserts a single-sided disk into any disk drive, or a high-density disk into a high-density disk drive, then the Disk Initialization Manager changes the buttons in the dialog box, as illustrated in Figure 5-2, because such disks can be formatted in only one way.

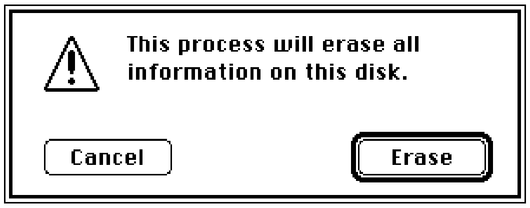
Figure 5-2 Alternate buttons for the disk initialization dialog box



This disk is unreadable:
Do you want to initialize it?

Regardless of the initial appearance of the disk initialization dialog box, it disappears if the user clicks Eject or Cancel. If, however, the user decides to initialize the disk, the text in the dialog box changes to warn the user that initialization erases any previous data on the disk, as illustrated in Figure 5-3.

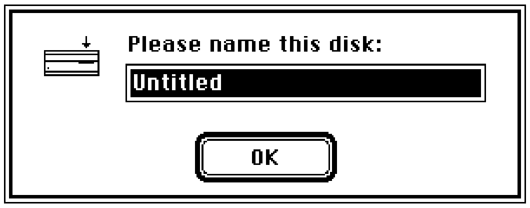
Figure 5-3 The disk initialization warning



This process will erase all information on this disk.

Finally, if the user decides to initialize the disk, the contents of the dialog box change so that the user can name the new disk, as illustrated in Figure 5-4.

Figure 5-4 The disk naming dialog box

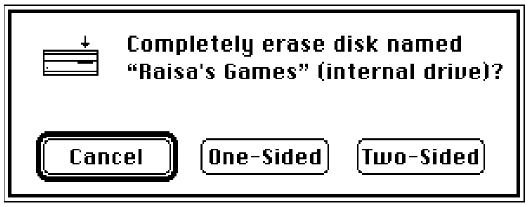


Please name this disk:
Untitled

After the user names the disk, the Disk Initialization Manager attempts to initialize it. If an error occurs and the initialization fails, an alert box notifies the user, and the disk is ejected.

The Disk Initialization Manager also provides a mechanism for using the standard interface to reinitialize disks that are already formatted. (This mechanism is useful, for example, when the user wants to reinitialize single-sided disks as double-sided disks.) The Finder takes advantage of this mechanism with its Erase Disk command, illustrated in Figure 5-5. After the user selects the erase operation from this dialog box, the reinitialization begins immediately, without further warnings. If desired, your application can use this same standard interface to allow users to reinitialize mounted disks (other than the startup volume). Your application can customize the text to be displayed in such a dialog box. Note that only a few utility applications actually need to provide users with this capability.

Figure 5-5 The Finder's disk erasing dialog box


Completely erase disk named
“Raisa's Games” (internal drive)?

If you are writing a utility program such as a disk-copying application, you might wish to initialize new disks or reinitialize valid disks without displaying the standard disk initialization dialog box. For example, your application might allow users to initialize multiple disks without having to respond to the standard dialog box each time. The Disk Initialization Manager provides low-level routines that allow you to do so. Unless you are writing a utility program of this type, you don't need to use these routines.

Bad Block Sparing

Beginning with system software version 7.0, the Disk Initialization Manager tries to initialize a disk even if it contains some bad blocks; this feature is called **bad block sparing.** Without bad block sparing, the Disk Initialization Manager considers a disk unusable even if just one block is bad. With bad block sparing, however, the Disk Initialization Manager attempts to work around the bad block by removing it from the pool of available free blocks. This prevents the File Manager from allocating the block to a file. Except in cases (described later) involving critical blocks on a disk, the Disk Initialization Manager can usually initialize a disk that would previously have been rejected as invalid. This section describes the operation of bad block sparing.

▲ **WARNING**

Applications that manipulate disks using File Manager routines are unaffected by bad block sparing. Software that accesses blocks directly from the disk or that makes assumptions about the physical blocks on a disk (such as a disk scavenger, recovery, or backup utility) is likely to fail or cause a loss of data on disks containing spared blocks. ▲

The bad block sparing occurs during the disk-zeroing phase of disk initialization. As a result, sparing occurs only when you call `DIZero` or `DIBadMount` (which internally calls `DIZero`), never when you call `DIFormat` or `DIVerify`. The only visible sign of the sparing process is an additional dialog box that contains the message "Re-Verifying Disk."

Disks without bad blocks are initialized exactly as in previous versions of system software. The sparing algorithm is invoked only if the disk verification fails during a call to the `DIBadMount` function or if the `DIZero` function encounters bad blocks during its zeroing. The sparing algorithm proceeds by making a second pass over the disk, writing and then reading back a test pattern. This testing is done a single track at a time. If any retries or errors occur during this test, all the sectors in the track are deemed bad.

If more than 25 percent of the disk is found to contain bad blocks, if the I/O errors appear to be due to hardware failure rather than media failure, or if certain critical sectors (described later) are bad, then the initialization fails just as it would have without the bad block sparing. Otherwise, the HFS volume structure is written to the disk. After the volume structure has been written, the Disk Initialization Manager performs several further operations during bad block sparing.

1. It sets the appropriate bits in the volume bitmap to indicate that the bad blocks are allocated to a file.
2. It creates file extent descriptors for the bad blocks and inserts them into the volume extents B*-tree so that the free-space scavenging that occurs at volume mount time (or that is done by disk utilities such as Disk First Aid) does not reintroduce the bad blocks into the volume bitmap. A special file ID (5) is used for these extents.
3. It sets bit 9 in the `drAtrb` field of the master directory block to indicate that bad blocks in the disk have been spared.
4. On 800K floppy disks only, it reduces the number of allocation blocks on the disk by 1 (from 1594 to 1593), to prevent previous versions of the Finder from doing disk-to-disk copies physically (that is, sector by sector). This copying operation would fail during an attempt to copy the bad blocks. The Finder does physical copies as an optimization only on disks containing exactly 1594 allocation blocks.

The critical sectors (those that must be good even on a spared disk) include the boot blocks, the master directory block and the spare master directory block, the volume bitmap, and the initial extents for the catalog and extents B*-tree files of the volume.

Notice that the bad block sparing algorithm does not create any new entries in the volume's catalog file. In other words, steps 1 and 2 of the algorithm trick the File Manager into thinking that the bad blocks have been allocated to some file, although no file is actually created to contain those blocks. For this reason, directory enumerations and file-by-file copies can proceed as they would have without bad block sparing. (If a file were created for the bad blocks, that file would need a parent directory; in that case, reading the catalog file to determine how many files that directory contains would produce erroneous results.)

Note

The bad block sparing capability described in this section applies only during disk initialization. The Operating System cannot correct problems that occur after a disk has been initialized (except by reinitializing the disk). ◆

Using the Disk Initialization Manager

The Disk Initialization Manager provides standard interfaces that allow your application

- to respond to the user's insertion of an unformatted or damaged disk by presenting the standard disk initialization dialog box
- to reinitialize valid disks, preserving their names but destroying their contents

You can override these standard interfaces by calling low-level Disk Initialization Manager routines, and you can also override the default volume characteristics that the Disk Initialization Manager gives to hierarchical volumes.

Responding to Disk-Inserted Events

When the user inserts a disk, the Operating System attempts to mount the volume on the disk by calling the File Manager function `PBMountVol`. If the volume is successfully mounted, an icon representing the disk appears on the desktop. The Operating System then generates a disk-inserted event. If the user is interacting with a standard file dialog box, the Standard File Package intercepts the disk-inserted event and handles it. Otherwise, the event is left in the event queue for your application to retrieve.

Your application must either mask out disk-inserted events or process them by checking to see whether the inserted disk is invalid. If you mask out such events, then each disk-inserted event needlessly occupies a position in the event queue until the user brings an application that can handle such events to the foreground or until your application invokes the Standard File Package. Also, displaying the disk initialization dialog box long after the disk has been inserted is likely to confuse the user. However, you might wish to mask out disk-inserted events when you create modal dialog boxes in which you process events with `WaitNextEvent` rather than `ModalDialog`. That way, your application can process disk-inserted events as soon as the modal dialog box closes.

Note

By default, the Dialog Manager's `ModalDialog` procedure automatically masks out disk-inserted events so that your application can handle them when dialog boxes close. If you wish to accept disk-inserted events in a modal dialog box in which you call `ModalDialog`, you must supply a filter procedure for the dialog box. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on how to write a filter procedure. ◆

Because handling disk-inserted events is easy, there is no good reason for your application to mask out the events in its main event loop. Listing 5-1 defines a procedure that your application can call when it receives a disk-inserted event.

Listing 5-1 Responding to disk-inserted events

```
PROCEDURE DoDiskEvent (myEvent: EventRecord);
VAR
   myPoint:    Point;
   myErr:      OSErr;
BEGIN
   IF HiWord(myEvent.message) <> noErr THEN
      BEGIN                             {attempt to mount was unsuccessful}
         DILoad;                        {load Disk Initialization Manager}
         SetPt(myPoint, 120, 120);      {set top left of dialog box}
         myErr := DIBadMount(myPoint, myEvent.message);
                                        {notify the user}
         DIUnload;                      {unload Disk Initialization Manager}
      END
   ELSE                                 {attempt to mount was successful}
      ;                                 {do other processing}
END;
```

The `DoDiskEvent` procedure in Listing 5-1 checks the high word of the event message to see if the disk is mounted properly. If it has not been mounted, `DoDiskEvent` calls the Disk Initialization Manager's `DIBadMount` function, which displays the disk initialization dialog box. Before doing so, `DoDiskEvent` calls `DILoad` to ensure that the Disk Initialization Manager and its dialog box are loaded into memory. If you did not call `DILoad` and the user started up with a floppy system startup disk, the Operating System might require that the user reinsert the system disk and might then attempt to initialize that disk. In Listing 5-1, if the user did start up with a floppy system startup disk on a single floppy-drive system, the `DILoad` procedure requests that the user insert the system disk so that it can read the necessary resources, and then it ejects that disk so that the user can again put the disk to be initialized into the drive. After calling `DIBadMount` to handle the uninitialized disk, `DoDiskEvent` calls `DIUnload` to release the resources `DILoad` read into memory.

Beginning with system software version 7.0, the first parameter to `DIBadMount` is ignored, and the disk initialization dialog box is automatically centered on the screen.

The procedure in Listing 5-1 ignores the result code returned by `DIBadMount` because ordinarily it does not concern your application. If an error does occur during initialization, `DIBadMount` informs the user and ejects the disk.

Erasing Initialized Disks

You can use the standard interface provided by the `DIBadMount` function to reinitialize disks that are already initialized correctly. Doing so permanently erases their contents, but does not change their names.

To reinitialize a disk, call `DIBadMount` with the high word of the event message equal to the result code `noErr`. The `DIBadMount` function presents the standard interface to initialize the disk in the drive whose number is specified by the low word of the event message. However, because the Disk Initialization Manager cannot know why your application wishes to reinitialize a disk, it cannot provide the initial text for the disk initialization dialog box. Therefore, your application must use the Dialog Manager's `ParamText` procedure to create a customized message, as illustrated in Listing 5-2.

If you need to reinitialize a valid disk but do not have access to the event message from when the disk was formatted, you can artificially create an event message by setting the event message to an integer representing the drive number, as follows:

```
myEvent.message := driveNum;
```

Doing so sets each of the high-order bits of the artificial event message to 0, which is desired because the constant `noErr` is equal to 0.

Listing 5-2 defines a procedure for displaying a disk initialization dialog box that allows the user to reinitialize the disk in the drive specified by `driveNum`. The disk initialization dialog box displays the text specified in the `myString` parameter. The procedure in Listing 5-2 in turn calls a procedure named `DoError`. You must define `DoError` to process the result code if the initialization did not successfully complete. The disk initialization dialog box does alert the user if the operation is not successfully completed, and the disk is ejected. However, your application might need to know that a formerly mounted disk is no longer mounted because reinitialization failed.

Listing 5-2 Reinitializing a valid disk

```
PROCEDURE DoEraseDisk (driveNum: Integer; myString: Str255);
VAR
   myPoint:          Point;
   myErr:            Integer;          {result code}
BEGIN
   DILoad;                             {load Disk Initialization Manager}
   ParamText(myString, '', '', '');    {set dialog text}
   SetPt(myPoint, 120, 120);           {set top left of dialog box}
   myErr := DIBadMount(myPoint, driveNum);
                                       {allow user to confirm erase}
   IF myErr <> noErr THEN
      DoError(myErr);                  {respond to error, if necessary}
   DIUnload;                           {unload Disk Initialization Manager}
END;
```

Overriding the Standard Initialization Interface

The disk initialization dialog box provides an easy-to-use, standard interface for initializing and reinitializing disks. However, if you wish, you can use three low-level Disk Initialization Manager functions that accomplish the three stages of disk initialization without presenting any user interface. The three functions are `DIFormat`, `DIVerify`, and `DIZero`. The `DIFormat` function attempts to format the disk, the `DIVerify` function verifies whether the format was successful, and the `DIZero` function updates the newly initialized volume's characteristics and attempts to spare any bad blocks on the disk.

Listing 5-3 shows how to reinitialize a disk without using the standard interface. The low-level functions work only if the disk is not already mounted in the disk drive. Therefore, Listing 5-3 uses high-level File Manager calls to unmount the volume and to remember the volume's name, so that it can be restored later. Because you are no longer using the standard interface, you must define the `DoError` procedure so that you can alert the user about an error.

Listing 5-3 Reinitializing a validly formatted disk without using the standard interface

```
PROCEDURE DoEraseDisk (driveNum: Integer);
VAR
   myErr:         OSErr;                      {result code}
   volName:       Str255;                     {name of volume}
   oldVRefNum:    Integer;                    {to unmount volume}
   oldFreeBytes:  LongInt;                    {for GetVInfo call}
BEGIN
   DILoad;                                    {load Disk Init. Manager}
   myErr := GetVInfo(driveNum, @volName, oldVRefNum, oldFreeBytes);
                                              {remember name of volume}
   IF myErr = noErr THEN
      myErr := UnmountVol(@volName, oldVRefNum);
                                              {unmount the disk}
   IF myErr = noErr THEN
      myErr := DIFormat(driveNum);            {format the disk}
   IF myErr = noErr THEN
      myErr := DIVerify(driveNum);            {verify format}
   IF myErr = noErr THEN
      myErr := DIZero(driveNum, volName);     {update volume information}
   IF myErr <> noErr THEN
      DoError(myErr);                         {respond to error}
   DIUnload;                                  {unload Disk Init. Manager}
END;
```

If you wish, you can also respond to a user's insertion of an uninitialized or damaged disk by simply formatting the disk without using the standard interface. Listing 5-4 defines a procedure for this purpose. Listing 5-4 differs from Listing 5-3 only in that it does not begin by unmounting the volume (because the File Manager does not mount uninitialized or damaged disks).

Listing 5-4 Initializing an uninitialized disk without using the standard interface

```
PROCEDURE DoInitDisk (driveNum: Integer; volName: Str255);
VAR
   myErr:         OSErr;                       {result code}
BEGIN
   DILoad;                                     {load Disk Init. Manager}
   myErr := DIFormat(driveNum);                {format the disk}
   IF myErr = noErr THEN
      myErr := DIVerify(driveNum);             {verify format}
   IF myErr = noErr THEN
      myErr := DIZero(driveNum, volName);      {update volume information}
   IF myErr <> noErr THEN
      DoError(myErr);                          {respond to error}
   DIUnload;                                   {unload Disk Init. Manager}
END;
```

Changing Default Volume Characteristics

The Disk Initialization Manager must set certain volume characteristics when it creates an HFS directory on a volume. Default values for these characteristics are stored in an HFS defaults record in ROM. If you wish, you can override those default values by placing a pointer to an HFS defaults record in the low-memory global variable `FmtDefaults`. The Disk Initialization Manager uses the record stored in ROM whenever this low-memory global variable contains `NIL`.

IMPORTANT

Most applications do not need to alter the default volume characteristics. This technique is useful primarily for applications, such as backup utilities, that intelligently adjust the allocation block size and clump size to maximize the amount of data written to a backup volume. ▲

The `HFSDefaults` data structure defines the HFS defaults record.

```
TYPE HFSDefaults =
RECORD
   sigWord:    PACKED ARRAY[0..1] OF Byte;   {signature word}
   abSize:     LongInt;                      {allocation block size in bytes}
   clpSize:    LongInt;                      {clump size in bytes}
   nxFreeFN:   LongInt;                      {next free file number}
   btClpSize:  LongInt;                      {B*-tree clump size in bytes}
   rsrv1:      Integer;                      {reserved}
   rsrv2:      Integer;                      {reserved}
   rsrv3:      Integer;                      {reserved}
END;
```

Field descriptions

`sigWord` The signature word to be used for newly initialized volumes. By default, this field is set to `'BD'` (hexadecimal $4244). You must set this field to `'BD'` for the volume to be recognized as an HFS volume.

`abSize` The number of bytes in each allocation block on newly initialized volumes. If you set this field to 0, the number of bytes in each allocation block is computed according to the following formula:

```
abSize = (1 + (blocks in volume/64K)) * 512 bytes
```

By default, this field is set to 0.

`clpSize` The number of bytes to be used for the clump on newly initialized volumes. By default, this field is set to `4*abSize`.

`nxFreeFN` The next free file number on newly initialized volumes. By default, this field is set to 16.

`btClpSize` The number of bytes to be used for the B*-tree clump on newly initialized volumes. If you set this field to 0, the number of bytes to be used for the B*-tree clump is computed according to the following formula:

```
btClpSize = ((blocks in volume)/128) * 512 bytes
```

By default, this field is set to 0.

`rsrv1` Reserved. Set to 0.

`rsrv2` Reserved. Set to 0.

`rsrv3` Reserved. Set to 0.

The code in Listing 5-5 fills in an `HFSDefaults` record, stores it in the system heap (so that the record remains in memory after the application terminates), and makes the low-memory global variable `FmtDefaults` a pointer to that record. Note that changing the default volume characteristics does not affect volumes that you have already initialized, but only volumes to be initialized.

Listing 5-5 Changing default volume characteristics

```
PROCEDURE ChangeHFSDefaults;
CONST
   FmtDefaults    = $039E;                {address of low-memory global}
TYPE
   HFSDefaultsPtr = ^HFSDefaults;         {pointer to override record}
   HFSDefaultsAdd = ^HFSDefaultsPtr;      {address of above pointer}
VAR
   myDefaults:    HFSDefaultsPtr;
BEGIN                                     {allocate record in system heap}
   myDefaults := HFSDefaultsPtr(NewPtrSysClear(SizeOf(HFSDefaults)));
   WITH myDefaults^ DO
      BEGIN
         ...                              {set fields of record}
      END;
   HFSDefaultsAdd(FmtDefaults)^ := myDefaults;
                                          {change value of global}
END;
```

If you later want to restore the default settings, you can reset the low-memory global variable `FmtDefaults` to `NIL`. Remember to delete any memory you have allocated.

Disk Initialization Manager Reference

This section describes the routines that are specific to the Disk Initialization Manager. See "Changing Default Volume Characteristics" on page 5-13 for a description of the Pascal data structure for the HFS defaults record.

Routines

The Disk Initialization Manager provides two routines (`DILoad` and `DIUnload`) that allow you to load and unload the package. The `DIBadMount` routine has two uses: to format uninitialized disks that the user inserts and to reinitialize volumes by erasing their data without changing their names. Last, three low-level routines (`DIFormat`, `DIVerify`, and `DIZero`) allow you to perform the steps of formatting, verifying, and zeroing the disk separately.

Loading and Unloading the Disk Initialization Manager

Even a user with a hard disk drive might occasionally use a floppy disk to start up the computer. When you call the Disk Initialization Manager to initialize a disk, it might need to read a resource from the System resource file. If the disk containing the System resource file is not already mounted, the user might need to switch disks, and system software might accidentally try to reinitialize the startup volume. The DILoad procedure allows you to avoid this problem by ensuring that the resources the Disk Initialization Manager needs are preloaded into memory. The DIUnload procedure reverses the effects of DILoad.

DILoad

You can use the DILoad procedure to ensure that the Disk Initialization Manager and its associated dialog box and dialog items are in memory.

```
PROCEDURE DILoad;
```

DESCRIPTION

The DILoad procedure reads the Disk Initialization Manager and its associated dialog box and dialog items into memory and makes them unpurgeable. Depending on which Macintosh model the user is using, the Disk Initialization Manager and the dialog box and dialog items are either in ROM or in the System file.

Ordinarily, you call the DILoad procedure when you anticipate that the user will need to format a disk. The Standard File Package automatically calls DILoad when you call StandardGetFile or StandardPutFile. If you are writing a utility program that frequently needs to initialize disks, such as a disk-copying program, you might call DILoad at the beginning of your application.

When you use the low-level disk-initialization routines DIFormat, DIVerify, and DIZero, the Disk Initialization Manager does not need to load a dialog box. Therefore, if you use only these routines, you can (if you wish) call the Resource Manager to read just the package resource into memory and the Memory Manager procedure to make it unpurgeable. To read just the package resource into memory, you can call the GetResource function with a resource ID of 2 and a resource type of 'PACK'. Then, you need to use the HNoPurge procedure to make the package resource unpurgeable.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for DILoad are

Trap macro	Selector
_Pack2	$0002

SPECIAL CONSIDERATIONS

Because the DILoad procedure allocates memory, you should not call it at interrupt time.

DIUnload

To free the memory space occupied by the Disk Initialization Manager, you can call the `DIUnload` procedure.

```
PROCEDURE DIUnload;
```

DESCRIPTION

The `DIUnload` procedure makes the Disk Initialization Manager and its associated dialog box and dialog items purgeable. They remain in memory until the Memory Manager purges the heap zone.

If you are using the low-level disk initialization routines and read just the package resource into memory, you can free the memory the package occupies by calling the `ReleaseResource` procedure.

To force the Memory Manager to purge the heap zone so that it really frees the memory occupied by the Disk Initialization Manager and its dialog box and dialog items, you can call one of the Memory Manager routines `PurgeMem` and `MaxMem`. For more information, see the chapter "Memory Manager" in *Inside Macintosh: Memory*.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `DIUnload` are

Trap macro	Selector
`_Pack2`	$0004

SPECIAL CONSIDERATIONS

Because `DIUnload` might affect memory, you should not call it at interrupt time.

Initializing a Disk

You can use the Disk Initialization Manager to initialize uninitialized disks and to reinitialize previously initialized disks. The `DIBadMount` function accomplishes both tasks.

DIBadMount

To respond to the user's insertion of an uninitialized or damaged disk, you can call the `DIBadMount` function.

```
FUNCTION DIBadMount (where: Point; evtMessage: LongInt): Integer;
```

`where`	The desired location, in global coordinates, of the upper-left corner of the disk initialization dialog box. In system software versions 7.0 and later, this parameter is ignored, and the dialog box is automatically centered on the screen.
`evtMessage`	The event message received when the disk is inserted. The high word of this message contains the result code associated with the disk insertion. The low word of this message indicates the number of the drive into which the user inserted the disk.

DESCRIPTION

The `DIBadMount` function evaluates the result code in the high word of the `evtMessage` parameter and responds appropriately. If the result code is `noErr`, the function allows the user to erase the contents of the disk. If the result code is `ioErr`, `badMDBErr`, or `noMacDskErr`, initializing the disk might correct the problem, and so `DIBadMount` displays a dialog box that explains the problem and allows the user to initialize the disk. If the result code is `extFSErr`, `memFullErr`, `nsDrvErr`, `paramErr`, or `volOnLinErr`, then initializing the disk would not correct the problem. In this case, `DIBadMount` ejects the disk from the drive and returns the result code.

Before presenting the disk initialization dialog box, `DIBadMount` checks whether the drive contains an already mounted volume. If so, it ejects the disk and returns 2 as its result. This happens rarely and could reflect an error in your application (for example, you forgot to call `DILoad`, and the user had to switch to the disk containing the System resource file).

The `DIBadMount` function uses just one disk initialization dialog box to cover all disk initialization situations. The dialog box contains many dialog items, which are hidden and shown as appropriate. The dialog box always contains an icon indicating the drive containing the disk to be initialized.

The initial text of the disk initialization dialog box depends on the result code received. For example, if you pass `noMacDskErr` to `DIBadMount` in the `evtMessage` parameter, the dialog box displays the text "This is not a Macintosh disk." If you pass the result code `noErr`, you can customize the message by using the Dialog Manager's `ParamText` procedure.

The disk initialization dialog box contains a button allowing the user to cancel the initialization and one or two buttons allowing the user to request initialization of the disk. Usually, the cancel button is labeled Eject, but if the result code passed to `DIBadMount` within the `evtMessage` parameter is `noErr`, then the cancel button is labeled Cancel. If the user responds to the disk initialization dialog box by clicking the Eject button, `DIBadMount` ejects the disk and returns 1 as its result. If the user clicks the Cancel button, `DIBadMount` returns 1 but does not eject the disk.

In most cases, the Initialize button is the only alternative to the Eject or Cancel button. However, if the user inserts a double-sided (but not high-density) disk into a double-sided or high-density disk drive, `DIBadMount` presents buttons labeled One-Sided and Two-Sided. The user can then decide whether to make the disk single-sided or double-sided. If the user clicks the Initialize button, the One-Sided button, or the Two-Sided button, `DIBadMount` warns the user that the initialization process erases any existing data on the disks. If the user proceeds, `DIBadMount` allows the user to name the disk if it is not already named and then updates the text of the dialog box to inform the user of the progress of the operation. If the operation fails, `DIBadMount` alerts the user and ejects the disk, returning an appropriate result code.

You can use `DIBadMount` to format hard disks as well as floppy disks. However, you should not attempt to format the startup volume.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `DIBadMount` are

Trap macro	Selector
`_Pack2`	$0000

SPECIAL CONSIDERATIONS

Because the `DIBadMount` function might allocate memory, you should not call it at interrupt time.

RESULT CODES

[no name]	2	Disk in specified drive is already mounted
[no name]	1	User canceled initializing
`noErr`	0	No error
`paramErr`	–50	Drive number specified is bad
`volOnLinErr`	–55	Volume is already online
`nsDrvErr`	–56	No such drive
`extFSErr`	–58	Disk has external file system
`lastDskErr`	–64	Last of the range of low-level disk errors
...		
`firstDskErr`	–84	First of the range of low-level disk errors
`memFullErr`	–108	Not enough memory

Low-Level Disk Initialization Routines

If you do not want to use the standard interface for initializing uninitialized volumes, you can use the Disk Initialization Manager's low-level routines. For example, if you are writing a disk-copying application, initializing a disk might be only part of the copying process. In this case, you might wish to create your own dialog boxes warning the user about the repercussions of initializing a disk and giving information on the progress of the initialization.

The three low-level disk-initialization routines are `DIFormat`, `DIVerify`, and `DIZero`. Ordinarily, you call them in that order to format an uninitialized disk, to verify the format, and to set the volume's volume information block and catalog.

DIFormat

To format a disk, you can use the `DIFormat` function.

```
FUNCTION DIFormat (drvNum: Integer): OSErr;
```

`drvNum`	The number of the drive containing the disk to be formatted.

DESCRIPTION

The `DIFormat` function attempts to format the disk in the drive specified by the `drvNum` parameter and returns a result code indicating whether it completed the formatting successfully or failed. Formatting a disk consists of writing special information onto it so that the disk driver can read from and write to the disk.

You can use `DIFormat` to format any unlocked disk, including single-sided disks, double-sided disks, high-density disks, and hard disk drives. It formats both sides of a double-sided disk.

You have to unmount a disk before calling the `DIFormat` function.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `DIFormat` are

Trap macro	Selector
`_Pack2`	$0006

SPECIAL CONSIDERATIONS

You should not call `DIFormat` at interrupt time.

RESULT CODES

`noErr`	0	No error
`volOnLinErr`	–55	Volume is online
`lastDskErr`	–64	Last of the range of low-level disk errors
...		
`firstDskErr`	–84	First of the range of low-level disk errors

DIVerify

To verify a disk you have formatted, you can use the `DIVerify` function.

```
FUNCTION DIVerify (drvNum: Integer): OSErr;
```

`drvNum`	The number of the drive containing the disk to be verified.

DESCRIPTION

The `DIVerify` function verifies the format of the disk in the drive specified by the `drvNum` parameter. It reads each bit from the disk and returns a result code indicating whether all bits were read successfully or not. The `DIVerify` function does not affect the contents of the disk itself.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `DIVerify` are

Trap macro	Selector
`_Pack2`	$0008

SPECIAL CONSIDERATIONS

You should not call `DIVerify` at interrupt time.

RESULT CODES

`noErr`	0	No error
`lastDskErr`	–64	Last of the range of low-level disk errors
`...`		
`firstDskErr`	–84	First of the range of low-level disk errors

DIZero

To complete the disk-initialization process, you can use the `DIZero` function.

```
FUNCTION DIZero (drvNum: Integer; volName: Str255): OSErr;
```

`drvNum`	The number of the drive containing the disk to be zeroed.
`volName`	The name of the volume (to be included in the volume information).

DESCRIPTION

On the unmounted volume in the drive specified by the given drive number, the `DIZero` function sets the volume information, the volume bitmap, a file directory, and the desktop database (or desktop file) to the settings corresponding to a volume with no files. This function completes the process of making any files previously on the volume permanently inaccessible. If the operation fails, `DIZero` returns a result code indicating that a low-level disk error occurred. Otherwise, it mounts the volume by calling the File Manager function `PBMountVol` and returns that function's result code.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `DIZero` are

Trap macro	Selector
`_Pack2`	$000A

SPECIAL CONSIDERATIONS

You should not call `DIZero` at interrupt time. In system software version 7.0 and later, `DIZero` automatically performs bad block sparing, as described in "Bad Block Sparing," beginning on page 5-7.

RESULT CODES

`noErr`	0	No error
`ioErr`	–36	I/O error
`paramErr`	–50	Drive number specified is bad
`volOnLinErr`	–55	Volume is already online
`nsDrvErr`	–56	No such drive
`noMacDskErr`	–57	Disk is not a Macintosh disk
`extFSErr`	–58	Disk has external file system
`badMDBErr`	–60	Master directory block is bad
`lastDskErr`	–64	Last of the range of low-level disk errors
`...`		
`firstDskErr`	–84	First of the range of low-level disk errors
`memFullErr`	–108	Not enough memory

Summary of the Disk Initialization Manager

Pascal Summary

Data Types

HFS Defaults Record

```
TYPE HFSDefaults =
RECORD
   sigWord:    PACKED ARRAY[0..1] OF Byte;   {signature word}
   abSize:     LongInt;                      {allocation block size in bytes}
   clpSize:    LongInt;                      {clump size in bytes}
   nxFreeFN:   LongInt;                      {next free file number}
   btClpSize:  LongInt;                      {B*-tree clump size in bytes}
   rsrv1:      Integer;                      {reserved}
   rsrv2:      Integer;                      {reserved}
   rsrv3:      Integer;                      {reserved}
END;
```

Routines

Loading and Unloading the Disk Initialization Manager

```
PROCEDURE DILoad;
PROCEDURE DIUnload;
```

Initializing a Disk

```
FUNCTION DIBadMount         (where: Point; evtMessage: LongInt): Integer;
```

Low-Level Disk-Initialization Routines

```
FUNCTION DIFormat           (drvNum: Integer): OSErr;
FUNCTION DIVerify           (drvNum: Integer): OSErr;
FUNCTION DIZero             (drvNum: Integer; volName: Str255): OSErr;
```

C Summary

Data Types

HFS Defaults Record

```
struct HFSDefaults {
      char        sigWord[2];    /*signature word*/
      long        abSize;        /*allocation block size in bytes*/
      long        clpSize;       /*clump size in bytes*/
      long        nxFreeFN;      /*next free file number*/
      long        btClpSize;     /*B-Tree clump size in bytes*/
      short       rsrv1;         /*reserved*/
      short       rsrv2;         /*reserved*/
      short       rsrv3;         /*reserved*/
};

typedef struct HFSDefaults HFSDefaults;
```

Routines

Loading and Unloading the Disk Initialization Package

```
pascal void DILoad           (void);
pascal void DIUnload         (void);
```

Initializing a Disk

```
pascal short DIBadMount      (Point where, long evtMessage);
```

Low-Level Disk-Initialization Routines

```
pascal OSErr DIFormat        (short drvNum);
pascal OSErr DIVerify        (short drvNum);
pascal OSErr DIZero          (short drvNum, const Str255 volName);
```

Assembly-Language Summary

Data Structures

`HFSDefaults` Data Structure

0	`sigWord`	word	signature word
2	`abSize`	long	allocation block size in bytes
6	`clpSize`	long	clump size in bytes
10	`nxFreeFN`	long	next free file number
14	`btClpSize`	long	B*-tree clump size in bytes
18	`rsrv1`	word	reserved
20	`rsrv2`	word	reserved
22	`rsrv3`	word	reserved

Trap Macros

Trap Macro Requiring Routine Selectors

`_Pack2`

Selector	Routine
$0000	`DIBadMount`
$0002	`DILoad`
$0004	`DIUnload`
$0006	`DIFormat`
$0008	`DIVerify`
$000A	`DIZero`

Global Variables

`FmtDefaults`	long	Pointer to substitute values for hierarchical volume directories.

Result Codes

[no name]	2	Disk in specified drive is already mounted
[no name]	1	User canceled initializing
`noErr`	0	No error
`ioErr`	–36	I/O error
`paramErr`	–50	Drive number specified is bad
`volOnLinErr`	–55	Volume is already online
`nsDrvErr`	–56	No such drive

`noMacDskErr`	–57	Disk is not a Macintosh disk
`extFSErr`	–58	Disk has external file system
`badMDBErr`	–60	Master directory block is bad
`lastDskErr`	–64	Last of the range of low-level disk errors
`firstDskErr`	–84	First of the range of low-level disk errors
`memFullErr`	–108	Not enough memory