INSIDE MACINTOSH

Display Manager

11/1/99

Contents

Introduction 9
About the Display Manager 10
  When the User Removes a Display 11
  Display Manager Problems Moving Windows 13
  When the User Moves the Menu Bar 16
  Display Modes 18
Using the Display Manager 23
  Handling Events in Response to Display Manager Changes 24
  Handling the Display Notice Event as a High-Level Event 29
  Handling the Display Notice Event Outside of an Event Loop 32
  Managing Windows In Response to the Display Notice Event 32
  Determining the Characteristics of the Video Devices 34
  Setting Configurations and Display Modes for Video Devices 35
Display Manager Reference 41
  Gestalt Constants 41
    Determining Display Manager Version 41
    Determining Display Manager Attributes 41
  Functions 42
    Getting Video Devices 43
      `.DMGetFirstScreenDevice` 43
      `DMGetNextScreenDevice` 44
      `DMGetNextMirroredDevice` 45
      `DMGetDisplayIDByGDevice` 45
      `DMGetGDeviceByDisplayID` 46
    Determining Display Modes and Display Configurations 47
      `DMCheckDisplayMode` 47
      `DMQDIsMirroringCapable` 48
      `DMCanMirrorNow` 49
      `DMIsMirroringOn` 49
      `DMGetNameByAVID` 50
      `DMGetGraphicInfoByAVID` 51
      `DMGetAVPowerState` 52
      `DMSetAVPowerState` 52
      `DMGetDisplayMode` 53
      `DMSaveScreenPrefs` 54

Changing Display Modes and Display Configurations 55
`DMBeginConfigureDisplays` 55
`DMSetDisplayMode` 56
`DMMoveDisplay` 57
`DMDisableDisplay` 58
`DMEnableDisplay` 60
`DMSetMainDisplay` 61
`DMMirrorDevices` 62
`DMUnmirrorDevices` 63
`DMBlockMirroring` 64
`DMUnblockMirroring` 64
`DMEndConfigureDisplays` 65
Adding and Removing Video Devices From the Device List 65
`DMNewDisplay` 66
`DMAddDisplay` 67
`DMRemoveDisplay` 69
`DMDisposeDisplay` 70
`DMNewDisplayModeList` 71
`DMGetIndexedDisplayModeFromList` 72
`DMDisposeList` 73
Registering and Unregistering Your Program 73
`DMRegisterExtendedNotifyProc` 74
`DMRemoveExtendedNotifyProc` 75
`DMSendDependentNotification` 76
Application-Defined Functions 77
`MyExtendedNotificationProc` 77
Data Types 79
`AVIDType` 79
`AVLocationRec` 79
`AVPowerStateRec` 80
`DependentNotifyRec` 80
`DisplayIDType` 81
`DMDepthInfoBlockRec` 81
`DMDepthInfoRec` 82
`DMDisplayModeListEntryRec` 83
`DMDisplayTimingInfoRec` 85
`DMListType` 86
`DMListIndexType` 86
`DMMakeAndModelRec` 86
`DMDisplayModeListIteratorProcPtr` 87
`DMExtendedNotificationProcPtr` 88

Constants 88
- Active Device Only Values 88
- Apple Event Notification Keywords 89
- Dependent Notification Constants 93
- Display/Device ID Constants 94
- Display Mode Flags 94
- Display Version Values 95
- Notification Messages 95
- Notification Types 97
- Switch Flags 98
- Video Depth Mode Values 99

Result Codes 100

C H A P T E R 1

About the Display Manager

Contents

Introduction 9
About the Display Manager 10
  When the User Removes a Display 11
  Display Manager Problems Moving Windows 13
  When the User Moves the Menu Bar 16
  Display Modes 18

Introduction

This chapter explains how the Display Manager allows users to dynamically change the arrangement and display modes of the monitors attached to their computers. For example, users can move their displays, add or remove displays, switch displays to higher or lower screen resolutions, and move the menu bar from one display to another—all without restarting their computers. When the user changes the display environment (as when disconnecting a display, for example), the Display Manager further assists the user by repositioning standard windows so that the user can find them in the new display environment.

This chapter helps you determine whether your application must move its own windows instead of relying on the Display Manager to move them. For example, if your application implements a tool palette that lacks a title bar, and the user disconnects the monitor that displays the tool palette, your application must move your tool palette to the main screen where the user can find it. Because the Display Manager never resizes windows, this chapter helps you determine whether to resize your application's windows after a display configuration change.

The Display Manager is available on all Power Macintosh computers and on color-capable Macintosh computers running system software version 7.5 and later. Applications that use only the standard window definition functions provided by the Window Manager generally do not need to use the Display Manager.

Users indirectly inform the Display Manager of changes they wish to make to their display environment by using the Monitors control panel or by adding and removing additional displays. The Monitors control panel in turn calls the Display Manager to change the display environment. The Display Manager sends an Apple event—the Display Notice event—to notify applications that it changed the display environment. In addition, the Display Manager generates an update event to notify all current applications to update their windows.

The Display Manager provides your application with functions that obtain `GDevice` structures for the video devices controlling the displays connected to the user's computer system. When repositioning a window, for example, your application can use the `GDevice` structures stored in the device list to determine which video device supports the largest display area or the greatest pixel depth.

This chapter explains the capabilities of the Display Manager and describes its default behavior when repositioning windows. This chapter helps determine whether your application needs to perform its own window positioning or sizing. If your application needs to perform its own window management in a changing environment, the next chapter, "Using the Display Manager," discusses how your application can determine if the user changed the display environment and how to manage its windows accordingly.

About the Display Manager

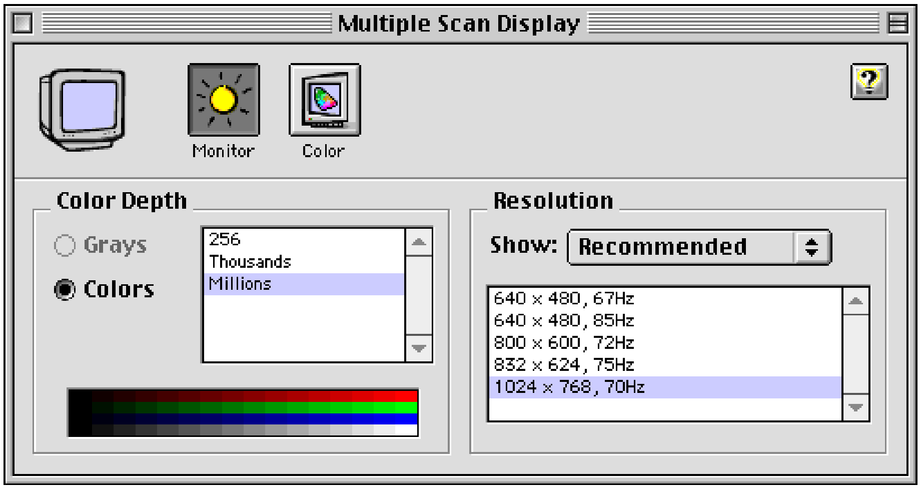
The Display Manager is a set of system software functions that support dynamic changes to the arrangement and display modes of the displays attached to a user's computer. (This book uses the term displays to represent output devices—such as video monitors and flat-panel displays—on which applications can show interactive visual information to the user. A video device is the hardware, such as the plug-in video card or the built-in video interface, that controls a display.)

The Monitors control panel mostly uses the Display Manager functions. After opening the Monitors control panel, the user can choose to

- move displays
- switch multiple-resolution displays to use higher or lower screen resolutions
- move the menu bar from one display to another
- select different pixel depths for video devices that support multiple depths

For example, a user can use a PowerBook computer that comes with an external video port to attach a second display. After the user opens the Monitors control panel, the user can move the menu bar from one display to another and the menu bar immediately moves to the user's desired location without the user restarting the computer.

Figure 1-1 The Monitors control panel



The user can also add or remove displays without restarting the computer. For example, a user can attach an external monitor to a sleeping PowerBook computer, wake the computer, and use both the external and built-in displays. If the user puts the PowerBook computer to sleep, detaches the external monitor, then wakes the computer, the Display Manager automatically moves windows that previously appeared on the external monitor onto the PowerBook built-in display.

The next several sections illustrate the default window positioning behaviors of the Display Manager.

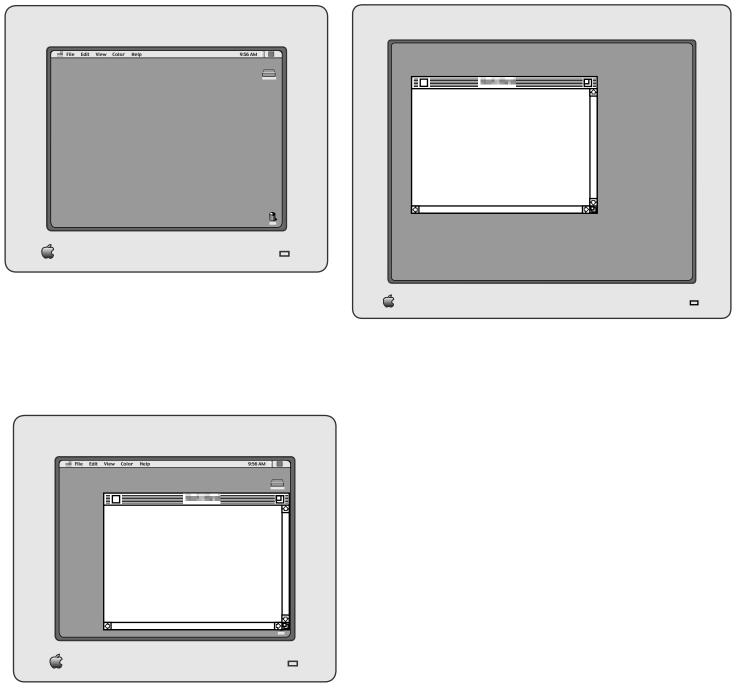
When the User Removes a Display

When a user removes a display, the Display Manager moves the windows that previously appeared on the disconnected display to the next closest display.

The Display Manager attempts to center the window of an alert or modal dialog box on the next closest display. If the alert or modal dialog box is larger than the screen, the Display Manager aligns its lower-left corner with the lower-left corner of the next closest display, thereby providing access to the area of the alert or modal dialog box with the OK and Cancel buttons.

The Display Manager assumes that any other type of window has a standard title bar. As illustrated in Figure 1-2 and Figure 1-3, the Display Manager then moves the window to the closest display by the shortest distance necessary to show the entire title bar.

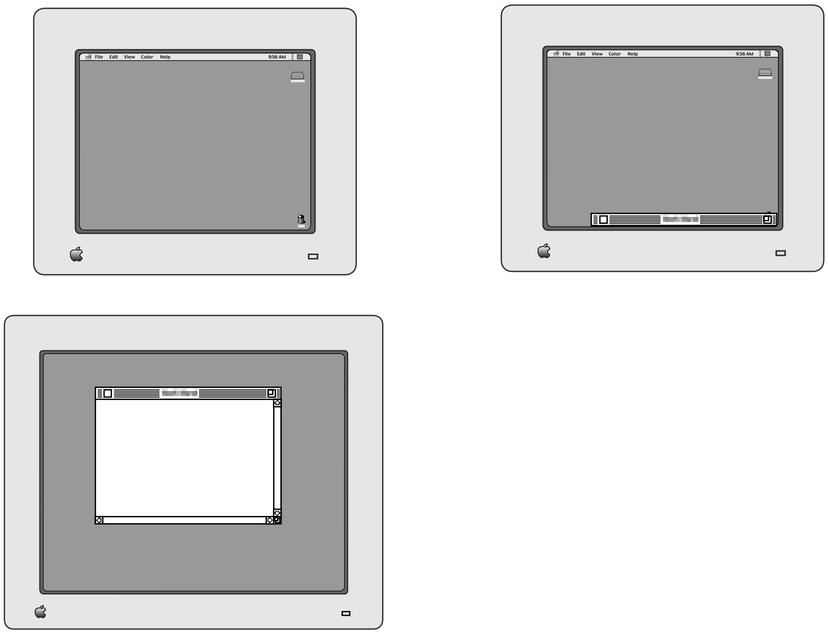
Figure 1-2 Default window repositioning when the user removes the right display



As shown in Figure 1-3, the content region of the window may still lie offscreen; but in a standard window, the user has access to the drag region of the title bar and to the zoom box. The user can therefore easily move the entire window onto the screen.

If the window is wider than the screen, the Display Manager fits the area in the title bar where the close box should appear onscreen.

Figure 1-3 Default window repositioning when the user removes the bottom display

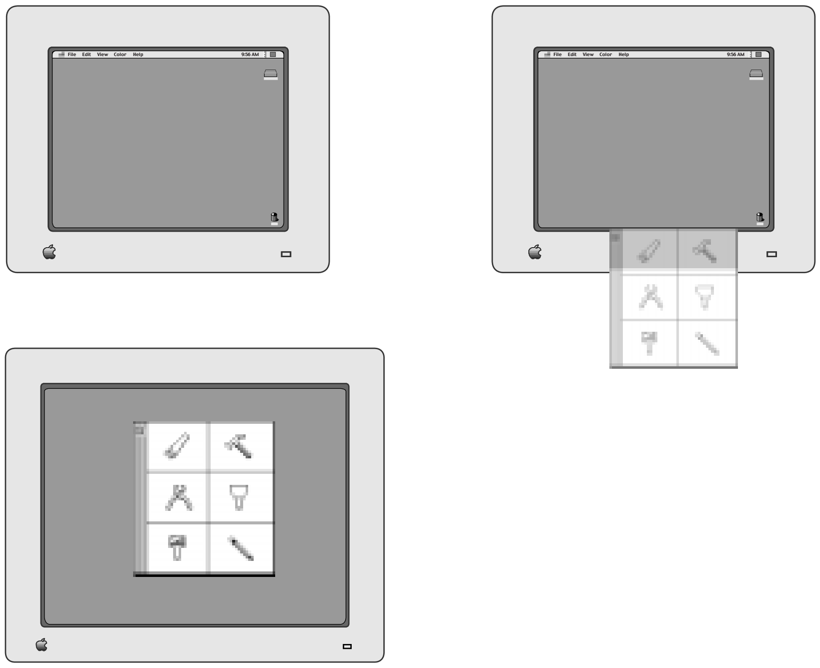


Display Manager Problems Moving Windows

When repositioning any window other than a window of type `dBoxProc`, the Display Manager assumes that the window has a standard title bar and moves the window to the closest display so that the title bar appears to the user. However, if the window does not have a title bar, the Display Manager may move the window to a position where the user cannot see it.

For example, on the left side of Figure 1-4 a window containing a tool palette and a nonstandard drag region appears in the lower display. When the user removes the lower display, as shown in the right side of the figure, the Display Manager moves the tool palette onto the main screen by the shortest distance necessary to display a standard title bar for the window. However, the window does not have a standard title bar, and so no part of the window appears onscreen. Applications that use windows without standard title bars must reposition their own windows as described in the chapter “Using the Display Manager.”

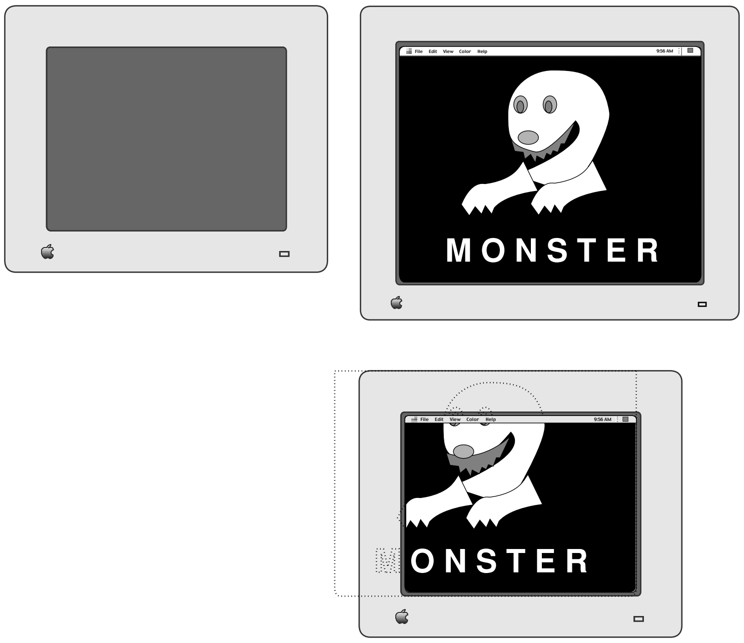
Figure 1-4 A problem with repositioning a nonstandard window



The Display Manager makes no attempt to stack or tile windows so that the user can see all of their titles bars simultaneously. Multiple windows repositioned by the Display Manager may obscure each other's title bars.

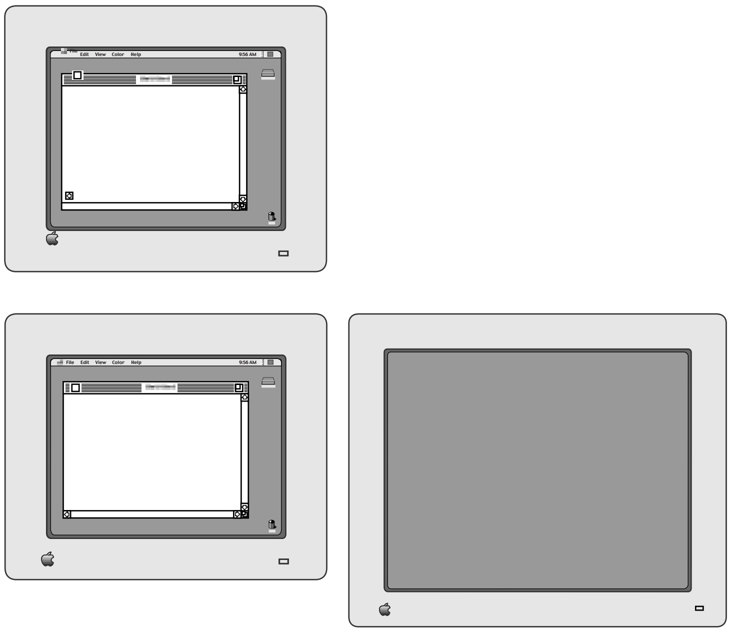
The Display Manager never resizes windows. Because of this, fixed size windows can present a problem. If a fixed size window appears on a large display, and the user removes that display, only part of the window appears when the Display Manager repositions it on a smaller display. Figure 1-5 illustrates how the Display Manager might reposition the window of a game that draws into a fixed size window.

Figure 1-5 Default repositioning of a fixed-size window



When the user adds a display, the Display Manager does not move any windows to that display. For example, in Figure 1-6 either the user or the application must move the window on the main screen to the display added on the right. If your application works best on the largest available screen or on the one displaying the greatest number of colors, you may want your application to move its windows to the added display.

Figure 1-6 Default window positioning when the user adds a display

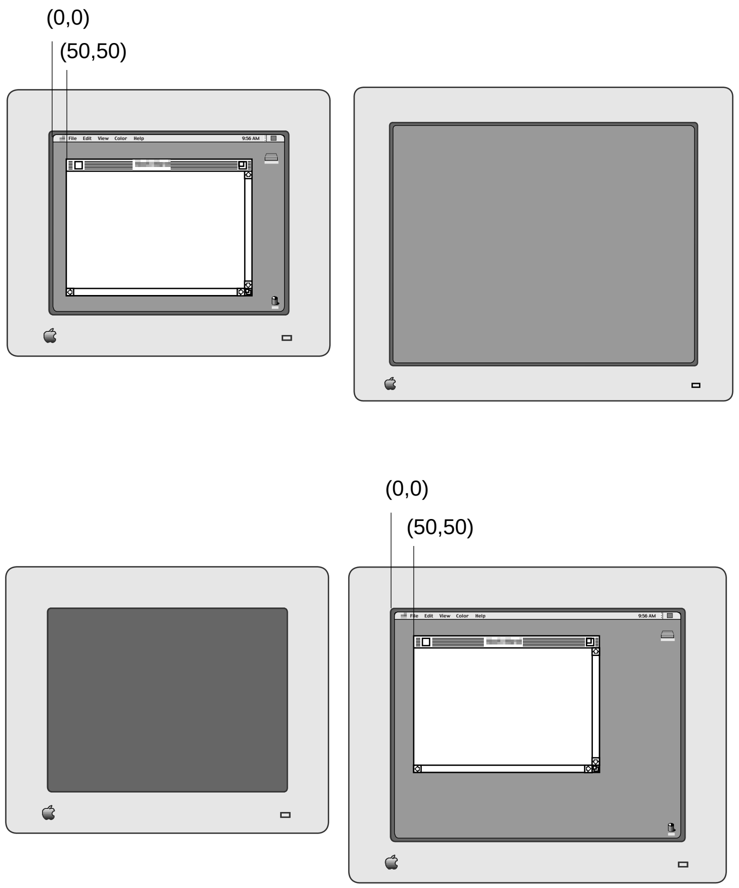


When the User Moves the Menu Bar

On a computer with multiple screens, the user can use the Monitors control panel to change the main screen—that is, the one that contains the menu bar. Color QuickDraw maps the (0,0) origin point of the global coordinate system to the main screen's upper-left corner, and other screens are positioned adjacent to it. The Window Manager automatically maintains window positions according to this global coordinate system.

When the user changes the main screen, the upper-left corner of the new main screen becomes the (0,0) origin point of QuickDraw's global coordinate system, and all windows initially maintain their position relative to this new origin point. When a user moves the menu bar, the user sees the windows that previously appeared beneath the menu bar on one display moved to the display that now contains the menu bar.

Figure 1-7 Default window positioning when the user moves the menu bar



For example, the top of Figure 1-7 shows a window on the left display. The left display is the main screen, and the upper-left corner of the window is at coordinates (50,50) on the global coordinate system. At the bottom of the figure, the user moves the menu bar to the right display. The window retains its upper-left coordinates of (50,50), but because the (0,0) origin of the global coordinate system moved to the right screen, the window now appears in the right display.

If the Display Manager finds that any windows move offscreen after the user moves the menu bar, the Display Manager repositions the windows as previously described—that is, it tries to move the title bar onto the closest screen or it tries to center the alert or modal dialog box on the closest screen.

Display Modes

The Display Manager allows users to choose from the various display modes available on their displays. A display mode is a combination of several interrelated capabilities that you can alter using the Display Manager to affect the display. You can characterize a display mode by

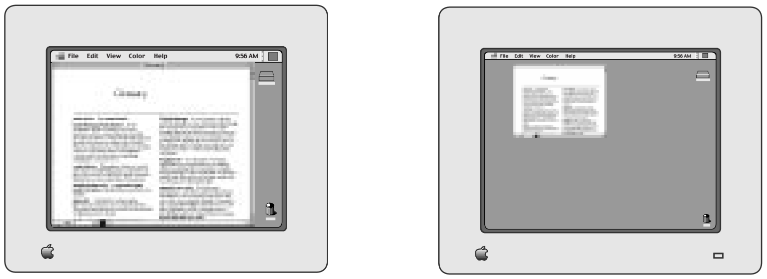
- the screen resolution, which determines the number of pixels that appear on the display screen
- the horizontal and vertical scan timings in use by the display
- the display's refresh rate

In addition to these capabilities, a display mode may also support multiple pixel depths, which determine the number of colors available on the display. You refer to the pixel depths available for a display mode as depth modes, and in various Display Manager data structures, depth modes are represented by constants or by their values from an enumerated list. A depth mode is also called a video mode.

Single-resolution grayscale or color monitors support multiple pixel depths only. Some multiple-resolution displays support display modes that change only the screen resolution and the pixel depth. For example, by choosing a lower screen resolution, a user with limited RAM can set the display to show a greater number of colors. Multiple-scan displays, however, are also capable of operating at multiple horizontal and vertical scan timings and at different refresh rates.

For example, a multiple-scan display might support display modes with screen resolutions of 640 by 480 pixels and 1152 by 870 pixels. The left side of Figure 1-8 illustrates a multiple-scan display operating at a screen resolution of 640 by 480 pixels. The right side of the figure illustrates the same display after it has been switched to a screen resolution of 1152 by 870 pixels.

Figure 1-8 Lower and higher screen resolutions on a multiple-scan monitor



When editing a bitmap image with a paint application, a user might wish to use the lower screen resolution, which, compared to the higher resolution, displays fewer pixels on the screen but displays them at a larger size. When using a spreadsheet application, however, the user might then want to switch to the higher resolution to increase the number of onscreen pixels and thereby view a greater number of cells in a spreadsheet.

To change the screen resolution, the user opens the Monitors control panel and selects the display mode for that resolution. The Display Manager then sends the video device driver a control request to switch the display to the newly selected display mode.

All required display modes appear when the user opens the Monitors control panel. For a particular type of display (for example, a 21-inch video monitor), a required display mode is one that Apple requires the display to support. A multiple-scan display must support several required display modes, one of which is designated to be the default display mode. The default display mode appears the first time a user turns on a display. For example, the first time a user connects and starts a 21-inch video monitor, it should use a mode displaying 1152 by 870 pixels. However, a 21-inch multiple-scan display is also required to support display modes with resolutions of 640 by 480 pixels, 832 by 624 pixels, and 1024 by 768 pixels, which the user can select with the Monitors control panel.

Using Display Manager functions, your application can change the display mode and the pixel depth of any display for the user, but your application should do so only with the consent of the user. The Monitors control panel is the user interface for changing the pixel depth, color capabilities, and positions of video devices. Because the user can control the capabilities of the video devices, your application should be flexible. Although it may have a preferred pixel depth, your application should do its best to accommodate less than ideal conditions.

However, if your application must have a specific pixel depth, or a particular screen resolution, it can display a dialog box that offers the user a choice between changing to that depth or canceling display of the image. This dialog box saves the user the trouble of going to the Monitors control panel before returning to your application. Your application can then use Display Manager functions to change the display mode or pixel depth of a display.

C H A P T E R 2

Using the Display Manager

Contents

Using the Display Manager 23
Handling Events in Response to Display Manager Changes 24
Handling the Display Notice Event as a High-Level Event 29
Handling the Display Notice Event Outside of an Event Loop 32
Managing Windows In Response to the Display Notice Event 32
Determining the Characteristics of the Video Devices 34
Setting Configurations and Display Modes for Video Devices 35

Using the Display Manager

The previous chapter explains how the Display Manager automatically repositions windows if necessary to ensure that windows are accessible when the user changes the display environment. If the Display Manager moves windows in a manner inappropriate for your application, your application should reposition them instead. Applications that use only the standard window definition functions provided by the Window Manager generally do not need to use the Display Manager.

However, you may need or want your application to perform its own window positioning under various circumstances, such as when

- your application benefits by displaying windows and their contents on the display controlled by the video device with the greatest pixel depth
- your application benefits by displaying windows on the largest available display
- your application uses nonstandard window definition functions that draw windows lacking title bars; examples include fixed-sized windows without title bars (games often use such windows), tool palettes with drag regions on the left sides of their windows, and floating windows

When necessary, the Display Manager automatically repositions windows of type `dBoxProc` (that is, alert boxes and modal dialog boxes) so that the lower-left corners of the windows appear onscreen. This gives users access to the area with the OK and Cancel buttons.

In addition, your application should respond to Display Manager changes if your application relies on display information that it stores internally. For example, if your application caches display positions, `GDevice` structures for displays other than the main screen, or the value in the `screenBits.bounds` field of the `screenBits` global variable, this information may become invalid after the user changes the display configuration. Therefore, your application should update its internal values accordingly after a display configuration change.

To determine whether the Display Manager is available, use the `Gestalt` function with the `gestaltDisplayMgrAttr` selector. Test the bit field indicated by the `gestaltDisplayMgrPresent` constant in the `response` parameter. If the bit is set, then the Display Manager is present.

Presence of the Display Manager does not guarantee that a computer also supports video mirroring. To determine whether QuickDraw supports video mirroring on the user's computer system, use the `DMQDIsMirroringCapable` function.

Handling Events in Response to Display Manager Changes

Users indirectly inform the Display Manager of changes they wish to make to their display environment by using the Monitors control panel, or by attaching or removing additional displays. The Display Manager in turn sends an Apple event—the Display Notice event—to notify applications that the display environment has changed.

After changing the display environment, the Display Manager also generates an update event to notify all current applications to update their windows.

Your application should always handle update events for its windows. However, your application needs to respond to the Display Notice event only if your application repositions its own windows, uses nonstandard windows, or must update any display information that it stores internally.

To receive the Display Notice event informing you of changes to the user's display configuration, you must either

- handle the Display Notice event as a high-level event in your application's normal event loop; or
- use the `DMRegisterExtendedNotifyProc` function to register a function that handles the Display Notice event as soon as the Display Manager issues it

If you write a utility—such as a control panel—that does not handle events through a normal event loop, or if you want your application to handle the Display Notice event as soon as it is issued instead of waiting for it to appear in the event queue, you should use the `DMRegisterExtendedNotifyProc` function.

Here is a summary of the Display Notice event (remember that you must use Apple Event Manager functions to obtain the information contained in Apple events such as this):

Display Notice—respond to display configuration changes

Event class	`kCoreEventClass`
Event ID	`kAESystemConfigNotice`
Required parameter	

Keyword:	`kAEDisplayNotice`
Descriptor type:	`AEDesc`
Data:	A list of descriptor structures, each specified by the keyword `kDisplayID`. Each `kDisplayID` descriptor structure contains information about a video device attached to the user's system. Within each `kDisplayID` descriptor structure are a pair of additional keyword-specified descriptor structures: `keyDisplayOldConfig` and `keyDisplayNewConfig`. A description of the video device's previous state is saved in the `keyDisplayOldConfig` descriptor structure, and a description of the video device's current state is saved in the `keyDisplayNewConfig` descriptor structure.
	Descriptions of these keyword-specified descriptor structures are in Table 2-1.

Requested action

Ensure that all windows appear to the user, and update any necessary display information that your application or utility stores internally.

Table 2-1 Keyword-specified descriptor structures.

Keyword	Value	Type	Description
`keyDeviceDepthMode`	`'dddm'`	`typeLongInteger`	The depth mode for the video device; that is, the value of the `gdMode` field in the `GDevice` structure for the device
`keyDeviceFlags`	`'dddf'`	`typeShortInteger`	The attributes for the video device as maintained in the `gdFlags` field of the `GDevice` structure for the device
`keyDeviceRect`	`'dddr'`	`typeQDRectangle`	The boundary rectangle of the video device; that is, the value of the `gdRect` field in the `GDevice` structure for the device
`keyDisplayDevice`	`'dmdd'`	`typeLongInteger`	A handle to the `GDevice` structure for the video device
`keyDisplayID`	`'dmid'`	`typeLongInteger`	The display ID for the video device
`keyDisplayMode`	`'dmdm'`	`typeLongInteger`	The `sResource` number from the video device for this display mode

Keyword	Value	Type	Description
`keyDMConfigVersion`	`'dmcv'`	`typeLongInteger`	The version number for this Display Notice event
`keyPixMapAlignment`	`'dppa'`	`typeLongInteger`	Reserved for future use
`keyPixMapCmpCount`	`'dpcc'`	`typeShortInteger`	The number of components used to represent a color for a pixel; that is, the value of the `cmpCount` field in the `PixMap` structure for the `GDevice` structure for the device
`keyPixMapCmpSize`	`'dpcs'`	`typeShortInteger`	The size in bits of each component for a pixel; that is, the value of the `cmpSize` field in the `PixMap` structure for the `GDevice` structure for the device
`keyPixMapColorTableSeed`	`'dpct'`	`typeLongInteger`	The value of the `ctSeed` field of the `ColorTable` structure for the `PixMap` structure for the `GDevice` structure for the video device

Keyword	Value	Type	Description
keyPixMapHResolution	'dphr'	typeFixed	he horizontal resolution of the pixel image in the PixMap structure for the GDevice structure for the video device
keyPixMapPixelSize	'dpps'	typeShortInteger	Pixel depth for the device; that is, the value of the pixelSize field in the PixMap structure for the GDevice structure for the video device
keyPixMapPixelType	'dppt'	typeShortInteger	The storage format for the pixel image on the device; that is, the value of the pixelType field in the PixMap structure for the GDevice structure for the video device
keyPixMapRect	'dpdr'	typeQDRectangle	The boundary rectangle into which QuickDraw can draw; that is, the bounds field in the PixMap structure for the GDevice structure for the video device

Keyword	Value	Type	Description
`keyPixMapReserved`	`'dppr'`	`typeLongInteger`	Reserved for future use
`keyPixMapResReserved`	`'dprr'`	`typeLongInteger`	Reserved for future use
`keyPixMapVResolution`	`'dpvr'`	`typeFixed`	The vertical resolution of the pixel image in the `PixMap` structure for the `GDevice` structure for the video device

Handling the Display Notice Event as a High-Level Event

To handle the Display Notice event as a high-level event like any other Apple event, you need to

- set the `isHighLevelEventAware` bit in your application's '`SIZE`' resource to indicate that your application supports high-level events (in which case your application must also support the four required Apple events)
- include code to handle high-level events in your main event loop (as illustrated in Listing 2-1)
- write a function that handles the Display Notice event (as illustrated in Listing 2-2)
- use the `AEInstallEventHandler` function to install the entry for handling the Display Notice event in your application's Apple event dispatch table

If you want your application to handle all window positioning itself (that is, if you do not want the Display Manager to automatically move any of your windows), you should also set the `isDisplayManagerAware` bit in the '`SIZE`' resource.

Listing 2-1 Handling Apple events in the event loop

```
void MyDoEvent(EventRecord *event)
{
    short           part, err;
    WindowPtr       window;
    char            key;
    switch ( event->what ) {
        /* here, handle null, mouse down, key down, update, and
            other necessary events */
        case kHighLevelEvent:
            DoHighLevelEvent(event);
            break;
    }
}

void DoHighLevelEvent(EventRecord *event)
{
    OSErr   myErr;
        /* handling only Apple-event types of high-level events */
        myErr = AEProcessAppleEvent(event);
}
```

Your application must use the `AEInstallEventHandler` function to add an entry to your application's Apple event dispatch table. This entry is the function that responds to the Display Notice event. For example, the following code fragment illustrates how to use `AEInstallEventHandler` to install an application-defined function called `DoAEDisplayUpdate`.

```
err = AEInstallEventHandler (kCoreEventClass,
                            kAESystemConfigNotice,
                            (ProcPtr)DoAEDisplayUpdate, 0, false);
```

Listing 2-2 shows an application-defined function called `DoAEDisplayUpdate` that uses Apple Event Manager functions to obtain information about the various video devices reported by the Display Notice event. The function `DoAEDisplayUpdate` uses this information to update its internal data structures for its windows and then calls another application-defined function that ensures that its windows are displayed optimally in the new configuration environment.

Listing 2-2 Responding to the Display Notice event

```
pascal OSErr DoAEDisplayUpdate
            (AppleEvent theAE,AppleEvent reply,long ref) {
    #pragma unused(theAE,reply,ref)
    AEDescList      DisplayList;
    AEDescList      DisplayID;
    AERecord        OldConfig,NewConfig;
    AEKeyword       tempWord;
    AEDesc          returnType;
    OSErr           myErr;
    long            result;
    long            count;
    Rect            oldRect, newRect;
    Size            actualSizeUnused;
    /* get a list of the displays from the Display Notice event */
    myErr =
    AEGetParamDesc(&theAE,kAEDisplayNotice,typeWildCard,&DisplayList);
    /* count the elements in the list */
    myErr = AECountItems(&DisplayList,&count);
    while (count >0)        /* decode the Display Notice event */
    {
        myErr = AEGetNthDesc(&DisplayList, count, typeWildCard,
        &tempWord, &DisplayID);
        myErr = AEGetNthDesc(&DisplayID, 1, typeWildCard, &tempWord,
        &OldConfig);
        myErr = AEGetKeyPtr(&OldConfig, keyDeviceRect, typeWildCard,
        &returnType, &oldRect, 8, actualSizeUnused);
        myErr = AEGetNthDesc(&DisplayID, 2, typeWildCard, &tempWord,
        &NewConfig);
        myErr = AEGetKeyPtr(&NewConfig, keyDeviceRect, typeWildCard,
        &returnType, &newRect, 8, actualSizeUnused);
        /* update internal info about the gdRects for the devices */
        MyUpdateWindowStructures(oldRect, newRect);
        count--;
        }
        /* move and resize windows as necessary*/
        MyDisplayWindows();
        return (noErr);
}
```

Handling the Display Notice Event Outside of an Event Loop

You may want your application to handle the Display Notice event as soon as it is issued instead of waiting for it to appear in the event queue. You can use the `DMRegisterExtendedNotifyProc` function to register a function to which the Display Manager directly sends the Display Notice event. By using `DMRegisterExtendedNotifyProc`, and by not setting the `isHighLevelEventAware` bit in the '`SIZE`' resource, you cause the Display Manager to send a Display Notice event directly to your handling function; your application or utility then receives no high-level Display Notice event.

To remove your Display Notice event-handling function, use the `DMRemoveExtendedNotifyProc` function.

Managing Windows In Response to the Display Notice Event

Using the Monitors control panel, the user can switch displays to use a different display mode and to change the display configurations. When your application receives the Display Notice event as described in the previous section, your application must determine whether it needs to reposition and perhaps resize its windows.

Listing 2-3 illustrates how an application can check whether its nonstandard window appears onscreen after Display Manager configuration changes have occurred. In this example, the application has a window with a title bar on its left side, as shown in the tool palette illustrated in Figure 1-4. After receiving the Display Notice event as shown in Listing 2-2, the application calls its `MyDisplayWindows` function, which in turn calls its `MyMakeToolWindowVisible` function. If `MyMakeToolWindowVisible` determines that the nonstandard title bar does not appear on any displays (in which case the user cannot move the window), `MyMakeToolWindowVisible` moves the entire window to the main screen where the user has access to the window.

Listing 2-3 Ensuring that a nonstandard window appears onscreen

```
static pascal OSErr MyMakeToolWindowVisible (WindowPeek window) {
    if (window->windowKind == applicationFloatKind) {
        Rect        checkRect;
        Rect        mainRect;
        GDHandle    maxAreaDevice;
        short       theWVariant;
        Rect        windowRect;
        theWVariant = GetWVariant(&window->port);
        MyGetWindowGlobalRect(window, &windowRect);
        /*get rectangle of window, in global coordinates, here */
        if (0 != (kVerBarFW & theWVariant))
        /* check if this is the window with a vertical title bar */
        {
        /* following line gets the rectangle of the title bar */
        SetRect(&checkRect, windowRect.left-kMyVertTitleWidth+kMyMinVisX,
                                     windowRect.top+kkMyMinVisV,
                                     windowRect.left-1-kMyMinVisX,
                                     windowRect.bottom-kMyMinVisV);
        /* following line calls an application-defined function that
        determines which screen contains the largest amount of the title
        bar */
        maxAreaDevice = MyFindMaxCoverageDevice(&checkRect);
        if (nil == maxAreaDevice)
        /* if the title bar doesn't appear on any screen, move window to
        the main screen */
        {   mainRect = (*GetMainDevice()) -> gdRect;
            MoveWindow(&Window->port, mainRect.left+10+kMyVertTitleWidth,
            mainRect.bottom-10-(windowRect.bottom-windowRec.top, FALSE);
        } }
        MyKeepWindowOnscreen(window, nil);
        /* handle other nonstandard window variants here */
        }
        return noErr;
}
```

Your application may find it useful to resize a window after moving it, or to optimize the color for its newly configured video device. You can use Display Manager functions to determine the characteristics of video devices, as explained in the next section.

Determining the Characteristics of the Video Devices

To determine the characteristics of available video devices, your application can use the `DMGetFirstScreenDevice` function to obtain a handle to the `GDevice` structure for the first video device in the device list. The `DMGetFirstScreenDevice` function is similar to the QuickDraw function `GetDeviceList`, except that when returning `GDevice` structures, `GetDeviceList` does not distinguish between the `GDevice` structures for video devices and the `GDevice` structures associated with no video devices. (For example, if system software uses the function `DMDisableDisplay` to disable the last remaining device in the device list, then `DMDisableDisplay` inserts into the device list a `GDevice` structure that is not associated with any video device. The `DMGetFirstScreenDevice` function will not return this `GDevice` structure, but `GetDeviceList` might.)

After using the `DMGetFirstScreenDevice` function to obtain a handle to the first `GDevice` structure for a display in the device list, your application can use the `DMGetNextScreenDevice` function to loop through all of the video devices in the device list. The `DMGetNextScreenDevice` function is similar to the QuickDraw function `GetNextDevice`, except that when returning `GDevice` structures, `GetNextDevice` does not distinguish between the `GDevice` structures for video devices and the `GDevice` structures associated with no video devices.

Another important difference between these two Display Manager functions (`DMGetFirstScreenDevice` and `DMGetNextScreenDevice`) and their related QuickDraw functions (`GetDeviceList` and `GetNextDevice`) is that with both Display Manager functions, your application can specify that the Display Manager return handles only to active video devices. (An active device is a video device whose display area is included in the user's desktop; the display area of an inactive device does not appear on the user's desktop.)

To get a handle to the `GDevice` structure for a video device that mirrors another, your application can use the `DMGetNextMirroredDevice` function.

Your application can pass the `GDevice` handle returned for any of these video devices to a QuickDraw function like `TestDeviceAttribute` or `HasDepth` to determine various characteristics of the video device, or your application can examine the `gdRect` field of the `GDevice` structure to determine the dimensions of the screen it represents.

Macintosh system software uses the `DMCheckDisplayMode` function to determine whether a video device supports a particular display mode and pixel depth. Typically, your application does not need to know whether a display mode is supported, but only whether a specific pixel depth is supported, in which case your application can use the Color QuickDraw function `HasDepth`.

To determine whether QuickDraw supports video mirroring on the user's computer system, your application can use the `DMQDIsMirroringCapable` function. Your application can use the `DMCanMirrorNow` function to determine whether video mirroring can activate. And to determine whether the user's computer system currently uses video mirroring, your application can use the `DMIsMirroringOn` function.

Finally, your application can use the `DMGetDisplayIDByGDevice` function to determine the display ID for a video device. A display ID is a long integer used by the Display Manager to uniquely identify a video device. Associating a display by its display ID is helpful when using functions such as `DMRemoveDisplay` that could change the `GDevice` structure associated with a video device. You can first determine the display ID for a device by using the `DMGetDisplayIDByGDevice` function. To later retrieve that device's `GDevice` structure after calling various Display Manager functions, your application can use the `DMGetGDeviceByDisplayID` function. Display IDs are not guaranteed to be persistent across reboots or sleep.

Setting Configurations and Display Modes for Video Devices

The Monitors control panel is the user interface for changing the pixel depth, color capabilities, and positions of video devices. Because the user can control the capabilities of the video devices, your application should be flexible. For instance, although your application may have a preferred pixel depth, it should do its best to accommodate less than ideal conditions.

Your application can use Display Manager functions to change the display mode and display configuration of the user's video devices, but your application should do so only with the consent of the user.

If your application must have a specific pixel depth, for example, it can display a dialog box that offers the user a choice between changing to that depth or canceling display of the image. This dialog box saves the user the trouble of going to the control panel before returning to your application. If it is absolutely necessary for your application to draw on a video device of a specific pixel depth, your application can then use either the `SetDepth` function or the `DMSetDisplayMode` function.

With the possible exception of the `DMSetDisplayMode` function and the `DMMirrorDevices` and `DMUnmirrorDevice` functions, applications should not need to use any of the Display Manager functions that change the user's display configuration. However, they are described for completeness, in case you find a compelling need for your application to change the user's display configuration. If your application must use multiple Display Manager calls that configure the user's displays, your application should first use the `DMBeginConfigureDisplays` function to postpone Display Manager configuration checking, the rebuilding of desktop regions, and Apple event notification of Display Manager changes. When finished configuring the user's displays, use the `DMEndConfigureDisplays` function. Using `DMBeginConfigureDisplays` and `DMEndConfigureDisplays` allows your application to wait until it has made all display changes before managing its windows in response to a single Display Notice event. It is important to pass the `displayState` variable obtained in `DMBeginConfigureDisplays` to the `DMEndConfigureDisplays` function.

C H A P T E R 3

Display Manager Reference

Contents

Display Manager Reference 41
- Gestalt Constants 41
  - Determining Display Manager Version 41
  - Determining Display Manager Attributes 41
- Functions 42
  - Getting Video Devices 43
    - .DMGetFirstScreenDevice 43
    - DMGetNextScreenDevice 44
    - DMGetNextMirroredDevice 45
    - DMGetDisplayIDByGDevice 45
    - DMGetGDeviceByDisplayID 46
  - Determining Display Modes and Display Configurations 47
    - DMCheckDisplayMode 47
    - DMQDIsMirroringCapable 48
    - DMCanMirrorNow 49
    - DMIsMirroringOn 49
    - DMGetNameByAVID 50
    - DMGetGraphicInfoByAVID 51
    - DMGetAVPowerState 52
    - DMSetAVPowerState 52
    - DMGetDisplayMode 53
    - DMSaveScreenPrefs 54
  - Changing Display Modes and Display Configurations 55
    - DMBeginConfigureDisplays 55
    - DMSetDisplayMode 56
    - DMMoveDisplay 57
    - DMDisableDisplay 58

DMEnableDisplay 60
DMSetMainDisplay 61
DMMirrorDevices 62
DMUnmirrorDevices 63
DMBlockMirroring 64
DMUnblockMirroring 64
DMEndConfigureDisplays 65
Adding and Removing Video Devices From the Device List 65
DMNewDisplay 66
DMAddDisplay 67
DMRemoveDisplay 69
DMDisposeDisplay 70
DMNewDisplayModeList 71
DMGetIndexedDisplayModeFromList 72
DMDisposeList 73
Registering and Unregistering Your Program 73
DMRegisterExtendedNotifyProc 74
DMRemoveExtendedNotifyProc 75
DMSendDependentNotification 76
Application-Defined Functions 77
MyExtendedNotificationProc 77
Data Types 79
AVIDType 79
AVLocationRec 79
AVPowerStateRec 80
DependentNotifyRec 80
DisplayIDType 81
DMDepthInfoBlockRec 81
DMDepthInfoRec 82
DMDisplayModeListEntryRec 83
DMDisplayTimingInfoRec 85
DMListType 86
DMListIndexType 86
DMMakeAndModelRec 86
DMDisplayModeListIteratorProcPtr 87
DMExtendedNotificationProcPtr 88

Constants 88
  Active Device Only Values 88
  Apple Event Notification Keywords 89
  Dependent Notification Constants 93
  Display/Device ID Constants 94
  Display Mode Flags 94
  Display Version Values 95
  Notification Messages 95
  Notification Types 97
  Switch Flags 98
  Video Depth Mode Values 99
Result Codes 100

Display Manager Reference

This document describes the functions, data types, and constants provided by the Display Manager. With the Display Manager, your application can configure display settings and allow users to move and reconfigure displays without losing windows or rebooting.

Gestalt Constants

You can use the Display Manager Gestalt Constants to determine the version and attributes of the current Display Manager.

Determining Display Manager Version

To determine the version of the current Display Manager, your application should pass the selector `gestaltDisplayMgrVers` to the `Gestalt` function.

```
enum {
                 = FOUR_CHAR_CODE('dplv')
};
```

Constant description

`'dplv'`	The selector you pass to determine what version of the Display Manager is present. For example, a Gestalt result may be 0x00020500, which means that the Display Manager version 2.5 is present.

Determining Display Manager Attributes

Before calling any function dependent upon the Display Manager, your application should pass the selector to the `Gestalt` function to determine the Display Manager attributes that are present.

```
enum {
                              = FOUR_CHAR_CODE('dply'),
                              = 0,
                              = 2,
                              = 3,
                              = 4,
                              = 5
};
```

Constant descriptions

`'dply'`	The `Gestalt` selector you pass to determine which Display Manager attributes are present.
`0`	If `true`, the Display Manager is present.
`2`	If `true`, the Display Manager can switch modes on mirrored displays.
`3`	If `true`, and you have registered for notification and you will be notified of depth mode changes.
`4`	Not yet supported. Most commonly comes up for display modes that are not marked . There is currently no system support for trying an unsafe mode and then restoring if the user does not confirm. When this is supported, this bit will be set.
`5`	If `true`, Display Manager supports profiles for displays.

Functions

Because all Display Manager functions may move or purge memory blocks or access handles, your application cannot call them at interrupt time.

Getting Video Devices

.DMGetFirstScreenDevice

Returns a handle for the first video device in the device list.

```
pascal GDHandle DMGetFirstScreenDevice (Boolean
                    activeOnly);
```

`activeOnly`	If `true`, the `DMGetFirstScreenDevice` function returns a handle to the first of all active video devices. If `false`, the function returns a handle to the first of all video devices, active or not. You may use the Active Device Constants in this parameter. See "Active Device Only Values" (page 88).
function result	If `activeOnly` is `true`, a handle to the `GDevice` structure for the first active video device. If `activeOnly` is `false`, a handle to the `GDevice` structure for the first video device.

DISCUSSION

The `DMGetFirstScreenDevice` function is useful if you want to find out more about the current mode.

You can use the function `DMGetNextScreenDevice` (page 44) to loop through all of the video devices in the device list.

The `DMGetFirstScreenDevice` function is similar to the QuickDraw function `GetDeviceList`, except that when returning `GDevice` structures, `GetDeviceList` does not distinguish between inactive and active video devices or between the `GDevice` structures for video devices and the `GDevice` structures associated with no video devices.

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMGetNextScreenDevice

Returns a handle for the next video device in the device list.

```
pascal GDHandle DMGetNextScreenDevice (
                GDHandle theDevice,
                Boolean activeOnly);
```

`theDevice`	A handle to the `GDevice` structure at which you want the function to begin. You can supply the handle returned by the function `DMGetFirstScreenDevice` or `DMGetNextScreenDevice`.
`activeOnly`	If `true`, the `DMGetNextScreenDevice` function returns a handle for the next active video device. If `false`, `DMGetNextScreenDevice` returns a handle for the next video device, active or not. You may use the Active Device Constants in this parameter. See "Active Device Only Values" (page 88).
function result	If `activeOnly` is `true`, a handle to the next `GDevice` structure for an active video device. If `activeOnly` is `false`, a handle to the next `GDevice` structure for a video device. If there are no more `GDevice` structures in the list, `DMGetNextScreenDevice` returns `NULL`.

DISCUSSION

The `DMGetNextScreenDevice` function is similar to the QuickDraw function `GetNextDevice`, except that when returning `GDevice` structures, `GetNextDevice` does not distinguish between inactive and active video devices or between the `GDevice` structures for video devices and the `GDevice` structures associated with no video devices.

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMGetNextMirroredDevice

Obtains a handle for a video device that mirrors another specified video device.

```
pascal OSErr DMGetNextMirroredDevice (
                GDHandle gDevice,
                GDHandle *mirroredDevice);
```

`gDevice`	A handle to the `GDevice` structure for the video device that another video device mirrors.
`mirroredDevice`	On return, a pointer to the handle for the video device that displays a mirror image of the device specified in the `gDevice` parameter.
function result	

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMGetDisplayIDByGDevice

Obtains the display ID number for a video device.

```
pascal OSErr DMGetDisplayIDByGDevice (
                GDHandle displayDevice,
                DisplayIDType *displayID,
                Boolean failToMain);
```

`displayDevice`	A handle to the `GDevice` structure for the video device whose display ID you wish to obtain.

`displayID` On return, a pointer to the display ID for the video device specified by the `displayDevice` parameter.

`failToMain` If `true` and the specified video device does not have a display ID, on return the function sets the `displayID` parameter to a pointer to the display ID of the video device for the main screen. If `false` and the specified video device does not have a display ID, the function returns the `kDMDisplayNotFoundErr` result code.

function result

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMGetGDeviceByDisplayID

Obtains a handle for the video device with a specified display ID.

```
pascal OSErr DMGetGDeviceByDisplayID (
                 DisplayIDType displayID,
                 GDHandle *displayDevice,
                 Boolean failToMain);
```

`displayID` The display ID for the video device whose handle you wish to obtain.

`displayDevice`
On return, a pointer to the handle to the `GDevice` structure for the video device specified by the `displayID` parameter.

`failToMain` If `true` and there is no video device associated with the `displayID` parameter, on return the function sets `displayDevice` to a pointer to the handle for the video device for the main screen. If `false` and there is no video device associated with the `displayID` parameter, the function returns the `kDMDisplayNotFoundErr` result code.

function result

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

Determining Display Modes and Display Configurations

The following functions allow your application to determine the display modes and other display configurations of the user's system.

DMCheckDisplayMode

Determines if a video device supports a particular display mode and pixel depth.

```
pascal OSErr DMCheckDisplayMode (
                 GDHandle theDevice,
                 unsigned long mode,
                 unsigned long depthMode,
                 unsigned long *switchFlags,
                 unsigned long reserved,
                 Boolean *modeOk);
```

`theDevice`	A handle to the `GDevice` structure for the video device whose display mode and pixel depth you wish to check.
`mode`	The display mode you wish to check. You get a list of display modes by calling `DMGetDisplayMode` (page 53)
`depthMode`	The pixel depth you wish to check. See "Video Depth Mode Values" (page 99) for list of possible values.

`switchFlags`	On return, a pointer to a long integer that indicates if a video device will support the mode specified by the `mode` parameter and the pixel depth specified by the `depthMode` parameter. See "Switch Flags" (page 98) for a description.
`reserved`	Reserved for future expansion. Pass `NULL` in this parameter.
`modeOk`	On return, a pointer to a `Boolean`. If `modeOk` points to a value of `true`, the user or your application can switch the display mode for the video device to the one specified by `mode`.

function result

DISCUSSION

Usually, your application only needs to know if a video device supports a specific pixel depth. Thus your application can use the Color QuickDraw function `HasDepth`. The function `DMCheckDisplayMode` is essentially obsolete, and is here for completeness.

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMQDIsMirroringCapable

Determines if QuickDraw supports video mirroring on the user's system.

```
pascal OSErr DMQDIsMirroringCapable (Boolean
                *qdIsMirroringCapable);
```

`qdIsMirroringCapable`
On return, a pointer to the value `true` if QuickDraw supports video mirroring; otherwise, a pointer to the value `false`.

function result

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMCanMirrorNow

Determines if video mirroring can operate on the user's system.

```
pascal OSErr DMCanMirrorNow (Boolean *canMirrorNow);
```

`canMirrorNow`
: On return, a pointer to a Boolean value; `true` indicates that mirroring can operate; `false` indicates it cannot.

function result

DISCUSSION

When the `canMirrorNow` parameter points to a value of `true`, the computer uses a version of QuickDraw that supports video mirroring, has at least two mirrorable displays, and does not have mirror blocking in effect.

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMIsMirroringOn

Determines if video mirroring is active.

```
pascal OSErr DMIsMirroringOn (Boolean *isMirroringOn);
```

`isMirroringOn`	On return, a pointer to a Boolean value; `true` indicates that mirroring is on; `false` indicates it is not.
function result	

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMGetNameByAVID

Obtains the name of a display device.

```
pascal OSErr DMGetNameByAVID (
                    AVIDType theID,
                    unsigned long nameFlags,
                    Str255 name);
```

`theID`	The ID number of the display device whose name you want to obtain.
`nameFlags`	Reserved for future expansion. Pass `NULL` in this parameter.
`name`	On return, a string containing the name of the display device specified by the parameter `theID`.
function result	

DISCUSSION

An AVID is really a display ID as an AVID references a video display just like a display ID. Developers planned to use AVIDs for an extended set of devices, however, they never did this.

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMGetGraphicInfoByAVID

Obtains information about the graphic display of an display device.

```
pascal OSErr DMGetGraphicInfoByAVID (
                 AVIDType theID,
                 PicHandle *theAVPcit,
                 Handle *theAVIconSuite,
                 AVLocationRec *theAVLocation);
```

`theID` The ID number of the display device whose information you want to obtain.

`theAVPcit` On return, a pointer to the handle for the picture structure you want to get.

`theAVIconSuite`
On return, a pointer to a handle whose structure reports the icon suite for a display device.

`theAVLocation`
On return, a pointer to the location structure for the device you want information about.

function result

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMGetAVPowerState

Obtains the current power state of a display.

```
pascal OSErr DMGetAVPowerState (
                 AVIDType theID,
                 AVPowerStatePtr getPowerState,
                 unsigned long reserved1);
```

`theID`	The ID number of the display device whose power state you want to obtain.
`getPowerState`	A pointer to type `AVPowerStateRec` (page 80). On return, this parameter points to a value specifying the current power state of display device.
`reserved1`	Reserved for future expansion. Pass `NULL` in this parameter.
function result	

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMSetAVPowerState

Sets the power state of an display device.

```
pascal OSErr DMSetAVPowerState (
                 AVIDType theID,
                 AVPowerStatePtr setPowerState,
                 unsigned long powerFlags,
                 Handle displayState);
```

`theID`	The ID number of the display device whose power state you want to change.
`setPowerState`	On return, this parameter points to a value that your application can use to set the power state of a display device.
`powerFlags`	A value that specifies the power state to which a display device can be set.
`displayState`	A handle to internal Display Manager information about the current display state.
function result	

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMGetDisplayMode

Obtains the current display mode of a specified video display.

```
pascal OSErr DMGetDisplayMode (
                GDHandle theDevice,
                VDSwitchInfoPtr switchInfo);
```

`theDevice`	A handle to the `GDevice` structure for the video device whose display mode you wish to obtain.
`switchInfo`	On return, a pointer to an internal Display Manager structure containing display mode information.
function result	

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMSaveScreenPrefs

Saves the user's screen configuration preferences.

```
pascal OSErr DMSaveScreenPrefs (
                 unsigned long reserved1,
                 unsigned long saveFlags,
                 unsigned long reserved2);
```

`reserved1`	Reserved for future expansion. Pass `NULL` in this parameter.
`saveFlags`	Reserved for future expansion. Pass `NULL` in this parameter.
`reserved2`	Reserved for future expansion. Pass `NULL` in this parameter.
function result	

DISCUSSION

Usually when you change screen properties such as pixel depth, the changes will only be temporary and will usually reset after restarting. However, the function `DMSaveScreenPrefs` makes the current screen properties permanent.

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

Changing Display Modes and Display Configurations

With the possible exception of the `DMSetDisplayMode`, `DMMirrorDevices` and `DMUnmirrorDevice` functions, applications should generally never need to use any of the following Display Manager functions that change the user's display configuration. In case you find a compelling need to change the user's display configuration, all Display Manager functions that change display configurations are described here for completeness.

Note that if your application uses Display Manager functions to change the display configuration of the user's video devices, your application should make these changes only with the consent of the user. If your application must have a specific pixel depth, for example, it should display a dialog box that offers the user a choice between changing to that depth or canceling display of the image.

DMBeginConfigureDisplays

Allows your application to configure displays.

```
pascal OSErr DMBeginConfigureDisplays (Handle
                *displayState);
```

`displayState`
: On return, a pointer to a handle to internal Display Manager information about the current display state. The `DMEndConfigureDisplays` (page 65) function and many other functions require this parameter.

function result

DISCUSSION

The `DMBeginConfigureDisplays` function tells the Display Manager to postpone Display Manager configuration checking, the rebuilding of desktop regions, and Apple event notification of Display Manager changes until your application uses the `DMEndConfigureDisplays` function.

You should call the function `DMBeginConfigureDisplays` before calling other Display Manager functions that configure the user's display. When calling functions that configure displays, you should pass the handle obtained by the `DMBeginConfigureDisplays` function. `DMBeginConfigureDisplays` causes system software to wait for your application to complete display changes before managing additional Display Manager events. When your application completes configuring the display environment, call the function `DMEndConfigureDisplays`.

DMSetDisplayMode

Sets the display mode and pixel depth for a video device.

```
pascal OSErr DMSetDisplayMode (
                GDHandle theDevice,
                unsigned long mode,
                unsigned long *depthMode,
                unsigned long reserved,
                Handle displayState);
```

`theDevice`	A handle to the `GDevice` structure for the video device whose display mode and pixel depth you wish to set.
`mode`	The number used by a video device to identify its display mode. If you supply the value 0 in this parameter, `DMSetDisplayMode` uses the current display mode. To specify another display mode, use the function `DMNewDisplayModeList` (page 71).
`depthMode`	A pointer to the desired pixel depth for the video device specified by `theDevice`. If you pass a pointer to 0, `DMSetDisplayMode` attempts to keep the current depth. If you pass a pointer to 1, 2, 4, 8, 16, or 32, `DMSetDisplayMode` attempts to set the device to use your specified pixel depth. If you supply a pointer to a value of 128 or greater, then `DMSetDisplayMode` sets the depth to the depth mode represented by the Video Depth Mode values. See “Video Depth Mode Values” (page 99) for more information.

	On return, this parameter contains a pointer to the new pixel depth. This value represents the depth mode closest to the one you requested when calling `DMSetDisplayMode`.
`reserved`	Reserved for future expansion. Pass `NULL` in this parameter.
`displayState`	If your application called `DMBeginConfigureDisplays` (page 55), you must pass the `displayState` handle obtained. Otherwise pass `NULL` in this parameter.
function result	

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMMoveDisplay

Moves the boundary rectangle for a video device.

```
pascal OSErr DMMoveDisplay (
                  GDHandle moveDevice,
                  short x,
                  short y,
                  Handle displayState);
```

`moveDevice`	A handle to the `GDevice` structure for the video device whose boundary rectangle you wish to move.
`x`	The horizontal coordinate on the QuickDraw global coordinate plane for the point to which you want to move the upper-left corner of the boundary rectangle.
`y`	The vertical coordinate on the QuickDraw global coordinate plane for the point to which you want to move the upper-left corner of the boundary rectangle.

`displayState`
: If your application called `DMBeginConfigureDisplays` (page 55), you must pass the `displayState` handle obtained. Otherwise pass `NULL` in this parameter.

function result

DISCUSSION

The `DMMoveDisplay` function moves the boundary rectangle for the specified video device to the point (`x,y`) in the QuickDraw global coordinate plane. If the video device controls the main screen, which always has the global coordinates (`0,0`), then all other video devices are offset by horizontal distance `x` and vertical distance `y`.

A boundary rectangle is the rectangle that links the local coordinate system of a graphics port to QuickDraw's global coordinate system and defines the area of the pixel image or bit image into which QuickDraw can draw. The boundary rectangle is stored in either the pixel map or the bitmap contained in a `GDevice` structure.

The Display Manager will reposition overlapped or discontiguous boundary rects to create a non-overlapping contiguous desktop space.

DMDisableDisplay

Makes a video device inactive by removing its display area from the desktop.

```
pascal OSErr DMDisableDisplay (
                 GDHandle disableDevice,
                 Handle displayState);
```

`disableDevice`
: A handle to the `GDevice` structure for the video device whose display you wish to disable.

`displayState`

If your application called `DMBeginConfigureDisplays` (page 55), you must pass the `displayState` handle obtained. Otherwise pass `NULL` in this parameter.

function result

DISCUSSION

You are not allowed to disable the last remaining display. Doing so will simply re-enable it. If you want to remove the last remaining display, thereby enabling the `GDevice` structure not associated with any video device, call the function `DMRemoveDisplay` (page 69).

If you specify the device for the main screen in the `disableDevice` parameter, then `DMDisableDisplay` picks another device and makes it the new main screen.

If `DMDisableDisplay` results in setting a new main screen, the handle you pass in the `disableDevice` parameter does not point to the same `GDevice` structure after `DMDisableDisplay` completes; instead, it points to the `GDevice` structure for the new main screen. If you need to recover the `GDevice` structure for the device you disabled, determine its display ID by using the function `DMGetDisplayIDByGDevice` (page 45) before calling `DMDisableDisplay`. Then use the function `DMGetGDeviceByDisplayID` (page 46) to obtain its structure.

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMEnableDisplay

Reactivates a display made inactive with the function `DMDisableDisplay` (page 58).

```
pascal OSErr DMEnableDisplay (
                GDHandle enableDevice,
                Handle displayState);
```

`enableDevice`
: A handle to the `GDevice` structure for the video device whose display you wish to make active.

`displayState`
: If your application called `DMBeginConfigureDisplays` (page 55), you must pass the `displayState` handle obtained. Otherwise pass `NULL` in this parameter.

function result

DISCUSSION

The function `DMEnableDisplay` reactivates the specified video device by adding to the desktop its display area.

If you add a display with the function `DMAddDisplay` (page 67) and there are no active displays, the Display Manager will enable the added display.

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMSetMainDisplay

Sets a display to be the main screen.

```
pascal OSErr DMSetMainDisplay (
                 GDHandle newMainDevice,
                 Handle displayState);
```

`newMainDevice`

A handle to the `GDevice` structure for the video device whose display you wish to make the main screen.

`displayState`

If your application called `DMBeginConfigureDisplays` (page 55), you must pass the `displayState` handle obtained. Otherwise pass `NULL` in this parameter.

function result

DISCUSSION

After a call to the function `DMSetMainDisplay`, the handle specified by the parameter `newMainDevice` will point to the `GDevice` structure for the video device whose display, before calling `DMSetMainDisplay`, was the main screen. To obtain a handle to the main screen, you can use the Color QuickDraw function `GetMainDevice`.

`DMSetMainDisplay` moves the menu bar to the display for the video device specified by `newMainDevice`. QuickDraw maps the (0,0) origin point of the global coordinate system to the main screen’s upper-left corner, and other screens are positioned adjacent to it.

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMMirrorDevices

Turns on video mirroring.

```
pascal OSErr DMMirrorDevices (
                 GDHandle gD1,
                 GDHandle gD2,
                 Handle displayState);
```

`gD1` A handle to the `GDevice` structure for the video device whose pixel image you want duplicated on another device.

`gD2` A handle to the `GDevice` structure for the video device on which you want to duplicate the pixel image specified in the `gD1` parameter.

`displayState`
If your application called `DMBeginConfigureDisplays` (page 55), you must pass the `displayState` handle obtained. Otherwise pass `NULL` in this parameter.

function result

DISCUSSION

Your application should leave control of video mirroring to the user. However, if video mirroring is useful for your application (for example, if your application displays on-screen presentations), you might provide a control so that the user can switch to video mirroring directly from your application. In this case, `DMMirrorDevices` is useful to your application. Your control should also allow the user to turn video mirroring off; the function `DMUnmirrorDevices` (page 63) supports this.

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMUnmirrorDevices

Turns off video mirroring.

```
pascal OSErr DMUnmirrorDevice (
                 GDHandle gDevice,
                 Handle displayState);
```

`gDevice` A handle to the `GDevice` structure for the video device on which you no longer wish to mirror the pixel image of another device.

`displayState`
If your application called `DMBeginConfigureDisplays` (page 55), you must pass the `displayState` handle obtained. Otherwise pass `NULL` in this parameter.

function result

DISCUSSION

When the function `DMUnmirrorDevice` completes, the display controlled by the video device specified in the `gDevice` parameter no longer contains the mirror image of another display.

Your application should leave control of video mirroring to the user. However, if video mirroring is useful for your application (for example, if your application displays on-screen presentations), you might provide a control so that the user can switch to video mirroring directly from your application. In this case, the function `DMMirrorDevices` (page 62) is useful for switching video mirroring on, and `DMUnmirrorDevice` function is useful for switching it off again.

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMBlockMirroring

Disables video mirroring.

```
pascal OSErr DMBlockMirroring (void);
```

function result

DISCUSSION

The function `DMBlockMirroring` disables video mirroring until the user restarts the computer or until an application calls the function `DMUnblockMirroring` (page 64).

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMUnblockMirroring

Reenables video mirroring disabled by the function `DMBlockMirroring` (page 64).

```
pascal OSErr DMUnblockMirroring (void);
```

function result

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMEndConfigureDisplays

Ends configuration begun by `DMBeginConfigureDisplays` (page 55).

```
pascal OSErr DMEndConfigureDisplays (Handle displayState);
```

`displayState`
: Supply this parameter with the handle obtained by the `DMBeginConfigureDisplays` (page 55) function.

function result

DISCUSSION

The function `DMEndConfigureDisplays` resumes Display Manager configuration checking, the rebuilding of desktop regions, and Apple event notification of Display Manager changes, all of which are postponed when you use the function `DMBeginConfigureDisplays` (page 55). Your application will then receive a single Display Notice event notifying your application of Display Manager changes, and your application can manage its windows accordingly.

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

Adding and Removing Video Devices From the Device List

This section describes the Display Manager functions for manipulating the device list. Generally, your application should not use these functions, but should instead allow system software to maintain the device list. These functions are described here for completeness only.

DMNewDisplay

Adds a video device to the device list and makes the device active.

```
pascal OSErr DMNewDisplay (
                 GDHandle *newDevice,
                 short driverRefNum,
                 unsigned long mode,
                 unsigned long reserved,
                 DisplayIDType displayID,
                 Component displayComponent,
                 Handle displayState);
```

`newDevice`	A pointer to a handle to a `GDevice` structure for the video device that you want to add to the device list.
`driverRefNum`	The reference number of the video device which you are adding to the device list. This information is usually set at system startup. The function `DMAddDisplay` (page 67) passes the value supplied here to the `InitGDevice` function in its `gdRefNum` parameter.
`mode`	The depth mode. Used by the video device driver, this value sets the pixel depth and specifies color. The function `DMAddDisplay` (page 67) passes the value supplied here to the function `InitGDevice` in its `mode` parameter.
`reserved`	Reserved for future expansion. Pass `NULL` in this parameter.
`displayID`	A unique identification for the display. For new displays, supply this parameter with the value 0, which causes the Display Manager to generate a unique display ID for this device. If this display was removed, then pass the display ID of the current display in this parameter.
`displayComponent`	Reserved for future expansion. Pass `NULL` in this parameter.

`displayState`
: If your application called `DMBeginConfigureDisplays` (page 55), you must pass the `displayState` handle obtained. Otherwise pass `NULL` in this parameter.

function result

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMAddDisplay

Adds the `GDevice` structure for a video device to the device list.

```
pascal OSErr DMAddDisplay (
                GDHandle newDevice,
                short driver,
                unsigned long mode,
                unsigned long reserved,
                unsigned long displayID,
                ComponentInstance displayComponent,
                Handle displayState);
```

`newDevice`
: A handle to the `GDevice` structure for the video device you want to add to the device list. The function `DMNewDisplay` (page 66) usually initializes this structure.

`driver`
: The reference number of the graphics device which you are adding to the device list. For most video devices, this information is set at system startup. The function `DMAddDisplay` passes the number supplied in this parameter to the `InitGDevice` function in its `gdRefNum` parameter.

`mode`	The depth mode. Used by the video device driver, this value sets the pixel depth and specifies color. The function `DMAddDisplay` (page 67) passes the value supplied here to the function `InitGDevice` in its `mode` parameter.
`reserved`	Reserved for future expansion. Pass `NULL` in this parameter.
`displayID`	A unique identification for the display. For new displays, supply this parameter with the value 0, which causes the Display Manager to generate a unique display ID for this device. If this display was removed, then pass the display ID number of the current display in this parameter.
`displayComponent`	Reserved for future expansion. Pass `NULL` in this parameter.
`displayState`	If your application called `DMBeginConfigureDisplays` (page 55), you must pass the `displayState` handle obtained. Otherwise pass `NULL` in this parameter.
function result	

DISCUSSION

The `DMAddDisplay` function adds the display specified by the `newDevice` parameter as inactive. However, if the specified display is the only display, the Display Manager automatically makes it active. Otherwise, you must call the function `DMEnableDisplay` (page 60) to make the specified display active.

The function `DMNewDisplay` (page 66) automatically calls `DMAddDisplay` and `DMEnableDisplay` (page 60). The only time you ned to call `DMAddDisplay` directly is after the device has been removed by `DMRemoveDisplay` (page 69) but not yet disposed of by `DMDisposeDisplay` (page 70).

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMRemoveDisplay

Removes a video device from the device list.

```
pascal OSErr DMRemoveDisplay (
                 GDHandle removeDevice,
                 Handle displayState);
```

`removeDevice`
: A handle to the `GDevice` structure for the video device you want to remove from the device list. The function `DMRemoveDisplay` does not actually dispose of this structure, but instead removes it from the device list.

`displayState`
: If your application called `DMBeginConfigureDisplays` (page 55), you must pass the `displayState` handle obtained. Otherwise pass `NULL` in this parameter.

function result

DISCUSSION

The function `DMRemoveDisplay` may call the function `DMSetMainDisplay` (page 61), which causes the `removeDevice` parameter to contain a handle to the `GDevice` structure for the new main screen, not the video device whose handle was passed to `DMRemoveDisplay`. To recover the `GDevice` structure for the disabled device, determine its display ID by using the function `DMGetDisplayIDByGDevice` (page 45) before calling `DMRemoveDisplay`. Then use the function `DMGetGDeviceByDisplayID` (page 46) to obtain the `GDevice` structure for the specified device.

You are not allowed to disable the last remaining display. Doing so will simply re-enable it. If you want to remove the last remaining display, thereby enabling the `GDevice` structure not associated with any video device, call `DMRemoveDisplay`.

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMDisposeDisplay

Disposes of the `GDevice` structure for a video device.

```
pascal OSErr DMDisposeDisplay (
                GDHandle disposeDevice,
                Handle displayState);
```

`disposeDevice`
A handle to the `GDevice` structure for a video device you want to delete.

`displayState`
If your application called `DMBeginConfigureDisplays` (page 55), you must pass the `displayState` handle obtained. Otherwise pass `NULL` in this parameter.

DISCUSSION

The `DMDisposeDisplay` function disposes of a `GDevice` structure, releases the space allocated for it, and disposes of all the data structures allocated for it. The Display Manager calls this function when appropriate.

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMNewDisplayModeList

Builds a new display mode list for a specified video device.

```
pascal OSErr DMNewDisplayModeList (
                DisplayIDType displayID,
                unsigned long modeListFlags,
                unsigned long reserved,
                DMListIndexType theListCount,
                DMListType theList);
```

`displayID` The display ID for the video device that will have a new display mode list.

`modeListFlags`
Reserved for future expansion. Pass `NULL` in this parameter.

`reserved` Reserved for future expansion. Pass `NULL` in this parameter.

`theListCount`
The number of entries in the display mode list specified by the `theList` parameter.

`theList`
The display mode list for the specified video device. You can access entries with the function `DMGetIndexedDisplayModeFromList` (page 72)

function result

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMGetIndexedDisplayModeFromList

Obtains a display mode from the display mode list built by `DMNewDisplayModeList` (page 71).

```
pascal OSErr DMGetIndexedDisplayModeFromList (
                DMListType theList,
                DMListIndexType theListCount,
                unsigned long reserved,
                DMDisplayModeListIteratorUPP listIterator,
                void *userData);
```

`theList`	A value that specifies the list from which to obtain information about the display modes created by the function `DMNewDisplayModeList` (page 71).
`theListCount`	A value that specifies the index of the display mode you wish to obtain.
`reserved`	Reserved for future expansion. Pass `NULL` in this parameter.
`listIterator`	A universal procedure pointer. The iterator this pointer specifies supplies the function to be called with the information about the display mode specified by `theListCount`.
`userData`	A pointer you pass for `listIterator` usually used to obtain information about the display mode from the UPP to the caller of `DMGetIndexedDisplayModeFromList`.
function result	

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMDisposeList

Disposes of a display mode list built by `DMNewDisplayModeList` (page 71).

```
pascal OSErr DMDisposeList (
                DMListType theList);
```

`theList` A value that specifies the display mode list you want to delete.

function result

DISCUSSION

You should call the `DMDisposeList` function after you have itterated the mode list.

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

Registering and Unregistering Your Program

This section describes the functions for registering and removing a function that responds to Display Manager changes.

DMRegisterExtendedNotifyProc

Registers a function that responds to a Display Notice event outside of an event loop.

```
pascal OSErr DMRegisterExtendedNotifyProc (
                DMExtendedNotificationUPP notifyProc,
                void *notifyUserData,
                unsigned short notifyOnFlags,
                ProcessSerialNumberPtr whichPSN);
```

`notifyProc` A pointer to your function that handles a Display Notice event.

`notifyUserData`
A pointer to caller-specific information which the Display Manager will return to your application when you request it.

`notifyOnFlags`
Reserved for future expansion. Pass `NULL` in this parameter.

`whichPSN` A pointer to the Process Serial Number associated with your Display Notice event-handling function. If this process terminates, the Display Notice event-handling function is automatically removed. For example, the Monitors control panel supplies the Finder's process number when registering its Display Notice event-handling function.

function result

DISCUSSION

Your Display Notice event-handling function should take one parameter, the Apple event describing the changes made to the display configuration, as shown here:

```
typedef pascal void (*DMNotificationProcPtr) (AppleEvent
                *theEvent);
```

When the Display Manager sends your function the Display Notice event, your application or utility should respond by moving or resizing its windows and update any internally-maintained video device information as appropriate.

When you are finished with your notification function, remove it by calling `DMRemoveExtendedNotifyProc` (page 75).

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMRemoveExtendedNotifyProc

Removes your Display Notice event-handling function registered in `DMRegisterExtendedNotifyProc` (page 74).

```
pascal OSErr DMRemoveExtendedNotifyProc (
                DMExtendedNotificationUPP notifyProc,
                void *notifyUserData,
                unsigned short removeFlags,
                ProcessSerialNumberPtr whichPSN);
```

`notifyProc`	A pointer to your function you want to remove that handles a Display Notice event.
`notifyUserData`	A pointer to caller-specific information which the Display Manager will return to your application when you request it.
`removeFlags`	Reserved for future expansion. Pass `NULL` in this parameter.
`whichPSN`	A pointer to the Process Serial Number associated with your Display Notice event-handling function. If this process terminates, the Display Notice event-handling function is automatically removed. For example, the Monitors control panel supplies the Finder's process number when registering its Display Notice event-handling function.

function result

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

DMSendDependentNotification

Notifies dependent displays of changes in depth mode or configuration.

```
pascal OSErr DMSendDependentNotification (
                ResType notifyType,
                ResType notifyClass,
                AVIDType displayID,
                ComponentInstance notifyComponent);
```

`notifyType`	The resource type that identifies the engine that made the change. Examples might be component engines that control brightness, contrast, or screen size. You may pass zero in this parameter. See `DependentNotifyRec` (page 80) for more information.
`notifyClass`	The resource type that identifies the class of change the user or engine has made, such as color depth, pixel size, or screen size. See `DependentNotifyRec` (page 80) for more information.
`displayID`	The ID number of the dependent display which you want to notify of Display Manager events. On return, the Display Manager sets the `notifyPortID` constant of the `DependentNotifyRec` (page 80) structure. See "Display/Device ID Constants" (page 94) for more information.
`notifyComponent`	A value that notifies the display component what engine, if any, caused a change in a dependent display. You may pass 0 in this parameter.

function result

DISCUSSION

The Display Manager uses the `DMSendDependentNotification` function to send notifications to registered Display Notice event-handling functions. This function uses all its parameters to supply values for the `DependentNotifyRec` (page 80) structure which is sent out to registrants. Generally, your application does not need to use this function.

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

Application-Defined Functions

Display Manager notification functions call the following application-defined functions when your application needs to know when certain events have occurred. The system software may implement these events or follow a user action. When these events occur, the Display Manager will send notification messages to registrants.

MyExtendedNotificationProc

Allows application-defined extended notification procedures.

```
pascal void MyExtendedNotificationProc (
                void *userData,
                short theMessage,
                void *notifyData);
```

`userData`	A pointer you passed into `DMRegisterExtendedNotifyProc` (page 74).

`theMessage`	A message selector. See "Notification Messages" (page 95) for information on specific message selectors.
`notifyData`	A pointer to message-specific information data provided by the the Display Manager, described in "Notification Messages" (page 95).
function result	

DISCUSSION

When you implement this function, the pointer you pass to the `DMRegisterExtendedNotifyProc` (page 74) function should be a universal procedure pointer with the following type definition:

```
typedef (DMExtendedNotificationProcPtr)
DMExtendedNotificationUPP;
```

To create a universal procedure pointer for your application-defined function, you should use the `NewDMExtendedNotificationProc` macro as follows:

```
DMExtendedNotificationUPP  MyExtendedNotificationUPP;

MyExtendedNotificationUPP = NewDMExtendedNotificationProc
(MyExtendedNotificationProc);
```

You can then pass `MyExtendedNotificationUPP` in the `notifyProc` parameter of the `DMRegisterExtendedNotifyProc` (page 74) function. When you no longer need notifications, you should remove it using the `DMRemoveExtendedNotifyProc` (page 75) function.

Using this call ensures that the call is made through a universal procedure pointer.

SPECIAL CONSIDERATIONS

Because this function may move or purge memory blocks or access handles, you cannot call it at interrupt time.

Data Types

This section discussses the general-purpose data types defined by the Display Manager.

AVIDType

The functions `DMGetNameByAVID` (page 50), `DMGetGraphicInfoByAVID` (page 51), `DMGetAVPowerState` (page 52), `DMSetAVPowerState` (page 52), and `DMGetDisplayMode` (page 53) use this type for one of their parameters to indicate the ID number of a particular audio video device. The function `DMSendDependentNotification` (page 76) also contains a parameter of type `AVIDType` which indentifies a display.

```
typedef unsigned longAVIDType;
```

AVLocationRec

The function `DMGetGraphicInfoByAVID` (page 51) uses the `AVLocationRec` structure to get information about graphic displays.

```
struct AVLocationRec {
   unsigned long locationConstant;
};
typedef struct AVLocationRec AVLocationRec;
typedef AVLocationRec *AVLocationPtr;
```

Field descriptions

`locationConstant`Reserved for future expansion. Set this field to zero.

AVPowerStateRec

The functions `DMGetAVPowerState` (page 52) and `DMSetAVPowerState` (page 52) contain a parameter that is a pointer to the `AVPowerStateRec` data type, which in turn corresponds to the `AVPowerStateRec` data type.

```
typedef VDPowerStateRec AVPowerStateRec;
typedef VDPowerStateRec *AVPowerStatePtr;
```

DependentNotifyRec

The function `DMSendDependentNotification` (page 76) uses the `notifyType` and `notifyClass` fields of the `DependentNotifyRec` structure.

```
struct DependentNotifyRec {
   ResType notifyType;
   ResType notifyClass;
   DisplayIDType notifyPortID;
   ComponentInstance notifyComponent;
   unsigned long notifyVersion;
   unsigned long notifyFlags;
   unsigned long notifyReserved;
   unsigned long notifyFuture;
};
typedef struct DependentNotifyRec DependentNotifyRec;
typedef DependentNotifyRec *DependentNotifyPtr;
```

Field descriptions

`notifyType`	A value that specifies the type of engine, if any, that made the change. The Display Manager may set this field to zero.
`notifyClass`	A value specifying the class of change that occurred: for instance, color or screen size. This field uses a value

	supplied by the constant described under “Dependent Notification Constants” (page 93) to specify the class of change that has occurred in a dependent display.
`notifyPortID`	Specifies which device was touched (`kInvalidDisplayID` specifies all or none).
`notifyComponent`	A value that indentifies the engine that made the change. The Display Manager may set this field to zero.
`notifyVersion`	Reserved for future expansion. The Display Manager sets this field to zero.
`notifyFlags`	Reserved for future expansion. The Display Manager sets this field to zero.
`notifyReserved`	Reserved for future expansion. The Display Manager sets this field to zero.
`notifyFuture`	Reserved for future expansion. The Display Manager sets this field to zero.

DisplayIDType

The functions `DMNewDisplayModeList` (page 71), `DMGetDisplayIDByGDevice` (page 45), and `DMGetGDeviceByDisplayID` (page 46) use this data type to specify a video device.

```
typedef unsigned longAVIDType;
typedef AVIDTypeDisplayIDType;
```

DMDepthInfoBlockRec

When you call the function `DMGetIndexedDisplayModeFromList` (page 72), the Display Manager passes the structure `DMDisplayModeListEntryRec` (page 83) to your application. Its field `displayModeDepthBlockInfo` is a pointer to the `DMDepthInfoBlockRec` (page 81) structure.

```
struct DMDepthInfoBlockRec {
   unsigned long   depthBlockCount;
   DMDepthInfoPtr  depthVPBlock;
   unsigned long   depthBlockFlags;
   unsigned long   depthBlockReserved1;
   unsigned long   depthBlockReserved2;
};
typedef struct DMDepthInfoBlockRec DMDepthInfoBlockRec;
typedef DMDepthInfoBlockRec *DMDepthInfoBlockPtr;
```

Field descriptions

`depthBlockCount`	Specifies the number of mode depths available.
`depthVPBlock`	Array of `DMDepthInfoRec` (page 82).
`depthBlockFlags`	Reserved for future expansion.
`depthBlockReserved1`	Reserved for future expansion.
`depthBlockReserved2`	Reserved for future expansion.

DMDepthInfoRec

This structure provides information that the structure `DMDepthInfoBlockRec` (page 81) supplies to the function `DMGetIndexedDisplayModeFromList` (page 72).

```
struct DMDepthInfoRec {
   VDSwitchInfoPtr depthSwitchInfo;
   VPBlockPtr   depthVPBlock;
   UInt32       depthFlags;
   UInt32       depthReserved1;
   UInt32       depthReserved2;
};
typedef struct DMDepthInfoRec DMDepthInfoRec;
typedef DMDepthInfoRec *DMDepthInfoPtr;
```

Field descriptions

`depthSwitchInfo`	A pointer to the structure `VDSwitchInfoRec`, which contains values that specify information on video switch modes and data. You can use this to call the function `DMSetDisplayMode` (page 56)
`depthVPBlock`	A pointer to the structure `VPBlock`, which supplies information about size, depth and format.
`depthFlags`	Values from the video structure `VDVideoParametersInfoRec`, which specify color, size, and depth.
`depthReserved1`	Reserved for future expansion.
`depthReserved2`	Reserved for future expansion.

DMDisplayModeListEntryRec

The `DMDisplayModeListEntryRec` structure contains information about a display mode in a display mode list built by the function `DMNewDisplayModeList` (page 71).

```
struct DMDisplayModeListEntryRec {
   UInt32              displayModeFlags;
   VDSwitchInfoPtr     displayModeSwitchInfo;
   VDResolutionInfoPtr displayModeResolutionInfo;
   VDTimingInfoPtr     displayModeTimingInfo;
   DMDepthInfoBlockPtr displayModeDepthBlockInfo;
   UInt32              displayModeVersion;
   StringPtr           displayModeName;
   DMDisplayTimingInfoPtr   displayModeDisplayInfo;
};
```

```
typedef struct DMDisplayModeListEntryRec
DMDisplayModeListEntryRec;
typedef DMDisplayModeListEntryRec
*DMDisplayModeListEntryPtr;
```

Field descriptions

`displayModeFlags`
: See "Display Mode Flags" (page 94) for a description.

`displayModeSwitchInfo`
: A pointer to video structure `VDSwitchInfoRec`, which provides information you need to tell the driver how to switch into different configurations, bit depths, or resolutions. See the function `DMSetDisplayMode` (page 56) for more information.

`displayModeResolutionInfo`
: A pointer to a pointer to video structure `VDResolutionInfoRec`, which provides information about horizontal pixels, maximum depth modes, and the vertical line of the specified display mode.

`displayModeTimingInfo`
: A pointer to a pointer to video structure `VDTimingInfoRec`, which provides information about timing, format of the specified display mode.

`displayModeDepthBlockInfo`
: A pointer to structure `DMDepthInfoBlockRec` (page 81), which provides information about available pixel formats and the `VPBlock`, including size and depth.

`displayModeVersion`
: The version of this structure. Currently it is version `kDisplayTimingInfoVersionOne`. See "Display Version Values" (page 95) for more information.

`displayModeName`
: A string pointer giving the display mode name. For example, "640 x 480, 67 Hz"

`displayModeDisplayInfo`
: A pointer to the `DMDisplayTimingInfoRec` (page 85) data type. This data type supplies information about the quality and default values of the timing.

DMDisplayTimingInfoRec

This structure supplies information about timing attributes, defaults and values to the structure `DMDisplayModeListEntryRec` (page 83).

```
struct DMDisplayTimingInfoRec {
   UInt32 timingInfoVersion;
   UInt32 timingInfoAttributes;
   SInt32 timingInfoRelativeQuality;
   SInt32 timingInfoRelativeDefault;
   UInt32 timingInfoReserved[16];
};
typedef struct DMDisplayTimingInfoRec
DMDisplayTimingInfoRec;
typedef DMDisplayTimingInfoRec *DMDisplayTimingInfoPtr;
```

Field descriptions

`timingInfoVersion`
: An unsigned 32 bit integer that shows the timing version. See "Display Version Values" (page 95) for timing version values.

`timingInfoAttributes`
: An unsigned 32 bit integer that the Display Manager sets to show timing attributes.

`timingInfoRelativeQuality`
: A signed 32 bit integer whose flags the Display Manager sets to provide information on the quality of the timing.

`timingInfoRelativeDefault`
: A signed 32 bit integer the Display Manager sets that specifies the relative default value of the timing.

`timingInfoReserved[16]`
: Reserved for future expansion.

DMListType

The functions DMDisposeList (page 73) and DMGetIndexedDisplayModeFromList (page 72) use the DMListType data type to help supply a list of display items.

```
typedef void *DMListType;
```

DMListIndexType

The function DMGetIndexedDisplayModeFromList (page 72) uses this data type to supply a list of display modes from which you can obtain information about a specified display mode.

```
typedef unsigned longDMListIndexType;
```

DMMakeAndModelRec

This structure stores information about a specified monitor or display. If you need to keep track of configurations and user preferences, you can store that information in this structure.

```
struct DMMakeAndModelRec {
   ResType manufacturer;
   UInt32  model;
   UInt32  serialNumber;
   UInt32  manufactureDate;
   UInt32  makeReserved[4];
};
typedef struct DMMakeAndModelRec DMMakeAndModelRec;
typedef DMMakeAndModelRec *DMMakeAndModelPtr;
```

`manufacturer`	Represents the manufacturer of the specified display.
`model`	Represents the model name of the specified display.
`serialNumber`	Represents the serial number of the specified display.
`manufactureDate`	Represents the date of manufacture of the specified display.
`makeReserved[4]`	Reserved for future expansion.

DMDisplayModeListIteratorProcPtr

The function `DMGetIndexedDisplayModeFromList` (page 72) has a parameter of type `DMDisplayModeListIteratorUPP`. The Display Manager defines the procedure pointer `DMDisplayModeListIteratorProcPtr` to use for this parameter. This data type is usually used to return information to the caller of `DMGetIndexedDisplayModeFromList.` `DMDisplayModeListIteratorProcPtr` is defined as follows:

```
void DMExtendedNotificationProcPtr (
                  void *userData,
                  DMListIndexType theListCount,
                  DMDisplayModeListEntryPtr
                  displaymodeInfo);
```

`userData`	A pointer to data about mode changes provided by the user. This is data passed into `DMGetIndexedDisplayModeFromList` (page 72)
`theListCount`	Specifies the list entry. See `DMListIndexType` (page 86) for more information. This is the index passed into `DMGetIndexedDisplayModeFromList` (page 72).
`displaymodeInfo`	A pointer to a structure of type `DMDisplayModeListEntryRec` (page 83) that provides display mode information.

DMExtendedNotificationProcPtr

When you call the function `DMRegisterExtendedNotifyProc` (page 74) you designate an application-defined function to handle the extended notification procedure. `DMExtendedNotificationProcPtr` is defined as follows:

```
void DMExtendedNotificationProcPtr (
                 void *userData,
                 short theMessage,
                 void *notifyData);
```

`userData`	A pointer you passed into `DMRegisterExtendedNotifyProc` (page 74).
`theMessage`	A message selector. See “Notification Messages” (page 95) for more information.
`notifyData`	A pointer to message-specific information data provided by the Display Manager described in "Notification Messages" (page 95). This is `NULL` for some messages.

Constants

Your application can use the following constants in Display Manager functions.

Active Device Only Values

The functions `.DMGetFirstScreenDevice` (page 43) and `DMGetNextScreenDevice` (page 44) contain the parameter `activeOnly` which you can specify with an Active Device Constant.

```
dmOnlyActiveDisplays = true,
dmAllDisplays        = false
```

Constant descriptions

`dmOnlyActiveDisplays`	Returns a handle to the `GDevice` structure for an active device only.
`dmAllDisplays`	Returns a handle to the `GDevice` structure for a device, active or not.

Apple Event Notification Keywords

The Display Manager sends an Apple event—the Display Notice event—to notify applications that it has changed the display environment. The keywords that specify the Display Notice event and its descriptor structures are described here.

```
enum {
     kAESystemConfigNotice = FOUR_CHAR_CODE('cnfg'),
     kAEDisplayNotice = FOUR_CHAR_CODE('dspl'),
     kAEDisplaySummary = FOUR_CHAR_CODE('dsum'),
     keyDMConfigVersion = FOUR_CHAR_CODE('dmcv'),
     keyDMConfigFlags = FOUR_CHAR_CODE('dmcf'),
     keyDMConfigReserved = FOUR_CHAR_CODE('dmcr'),
     keyDisplayID = FOUR_CHAR_CODE('dmid'),
     keyDisplayComponent = FOUR_CHAR_CODE('dmdc'),
     keyDisplayDevice = FOUR_CHAR_CODE('dmdd'),
     keyDisplayFlags = FOUR_CHAR_CODE('dmdf'),
     keyDisplayMode = FOUR_CHAR_CODE('dmdm'),
     keyDisplayModeReserved = FOUR_CHAR_CODE('dmmr'),
     keyDisplayReserved = FOUR_CHAR_CODE('dmdr'),
     keyDisplayMirroredId = FOUR_CHAR_CODE('dmmi'),
     keyDeviceFlags = FOUR_CHAR_CODE('dddf'),
     keyDeviceDepthMode = FOUR_CHAR_CODE('dddm'),
     keyDeviceRect = FOUR_CHAR_CODE('dddr'),
     keyPixMapRect = FOUR_CHAR_CODE('dpdr'),
     keyPixMapHResolution = FOUR_CHAR_CODE('dphr'),
     keyPixMapVResolution = FOUR_CHAR_CODE('dpvr'),
     keyPixMapPixelType = FOUR_CHAR_CODE('dppt'),
     keyPixMapPixelSize = FOUR_CHAR_CODE('dpps'),
     keyPixMapCmpCount = FOUR_CHAR_CODE('dpcc'),
     keyPixMapCmpSize = FOUR_CHAR_CODE('dpcs'),
     keyPixMapAlignment = FOUR_CHAR_CODE('dppa'),
     keyPixMapResReserved = FOUR_CHAR_CODE('dprr'),
     keyPixMapReserved = FOUR_CHAR_CODE('dppr'),
     keyPixMapColorTableSeed = FOUR_CHAR_CODE('dpct'),
     keySummaryMenubar = FOUR_CHAR_CODE('dsmb'),
     keySummaryChanges = FOUR_CHAR_CODE('dsch'),
     keyDisplayOldConfig = FOUR_CHAR_CODE('dold'),
     keyDisplayNewConfig = FOUR_CHAR_CODE('dnew')
};
```

Constant descriptions

`kAESystemConfigNotice`
: Keyword for the Event ID for Display Notice event.

`kAEDisplayNotice`
: Keyword for a required parameter to a Display Notice event.

`keyDMConfigVersion`
: Keyword for the descriptor structure describing the version number for this Display Notice event.

`keyDMConfigFlags`
: Reserved for future expansion. Internal use only.

`keyDMConfigReserved`
: Reserved for future expansion. Internal use only.

`keyDisplayID`
: Keyword for the descriptor structure describing the display ID for the video device.

`keyDMDisplayComponent`
: Unless you are disconnecting display components, this is for internal use only.

`keyDisplayDevice`
: Keyword for the descriptor structure containing a handle to the `GDevice` structure for the video device.

`keyDisplayFlags`
: Reserved for future expansion. Internal use only.

`keyDisplayMode`
: Keyword for the descriptor structure containing the `sResource` number from the video device for this display mode.

`keyDisplayModeReserved`
: Reserved for future expansion. Internal use only.

`keyDisplayReserved`
: Reserved for future expansion. Internal use only.

`keyDisplayMirroredID`
: Keyword for the display this device is mirrored to.

`keyDeviceFlags`
: Keyword for the descriptor structure describing the attributes for the video device as maintained in the `gdFlags` field of the `GDevice` structure for the device.

`keyDeviceDepthMode`
: Keyword for the descriptor structure describing the depth mode for the video device; that is, the value of the `gdMode` field in the `GDevice` structure for the device.

`keyDeviceRect`
: Keyword for the descriptor structure describing the boundary rectangle of the video device; that is, the value of the `gdRect` field in the `GDevice` structure for the device.

`keyPixMapRect`
: Keyword for the descriptor structure describing the boundary rectangle into which QuickDraw can draw; that is, the bounds field in the `PixMap` structure for the `GDevice` structure for the video device.

`keyPixMapHResolution`
: Keyword for the descriptor structure describing the horizontal resolution of the pixel image in the `PixMap` structure for the `GDevice` structure for the video device.

`keyPixMapVResolution`

Keyword for the descriptor structure describing the vertical resolution of the pixel image in the `PixMap` structure for the `GDevice` structure for the video device.

`keyPixMapPixelType`

Keyword for the descriptor structure describing the storage format for the pixel image on the device; that is, the value of the `pixelType` field in the `PixMap` structure for the `GDevice` structure for the video device.

`keyPixMapPixelSize`

Keyword for the descriptor structure describing the pixel depth for the device; that is, the value of the `pixelSize` field in the `PixMap` structure for the `GDevice` structure for the video device.

`keyPixMapCmpCount`

Keyword for the descriptor structure containing the number of components used to represent a color for a pixel; that is, the value of the `cmpCount` field in the `PixMap` structure for the `GDevice` structure for the device.

`keyPixMapCmpSize`

Keyword for the descriptor structure describing the size in bits of each component for a pixel; that is, the value of the `cmpSize` field in the `PixMap` structure for the `GDevice` structure for the device.

`keyPixMapAlignment`

Reserved for future expansion. Internal use only.

`keyPixMapResReserved`

Reserved for future expansion. Internal use only.

`keyPixMapReserved`

Reserved for future expansion. Internal use only.

`keyPixMapColorTableSeed`

Keyword for the descriptor structure containing the value of the `ctSeed` field of the `ColorTable` structure for the `PixMap` structure for the `GDevice` structure for the video device.

`keySummaryMenubar`

Reserved for future expansion. Internal use only.

`keySummaryChanges`
: Reserved for future expansion. Internal use only.

`keyDisplayOldConfig`
: Keyword for the descriptor structure describing the video device's previous state.

`keyDisplayNewConfig`
: Keyword for the descriptor structure describing the video device's new state.

Dependent Notification Constants

The function `DMSendDependentNotification` (page 76) contains the parameter `notifyClass` which you can specify with a Dependent Notification Constant.

```
enum {
kDependentNotifyClassShowCursor = FOUR_CHAR_CODE('shcr'),
kDependentNotifyClassDriverOverride =
FOUR_CHAR_CODE('ndrv'),
kDependentNotifyClassDisplayMgrOverride =
FOUR_CHAR_CODE('dmgr'),
kDependentNotifyClassProfileChanged =
FOUR_CHAR_CODE('prof'),
};
```

Constant descriptions

`kDependentNotifyClassShowCursor`
: The Display Manager sends an extended notification when a hidden cursor shows during a display unmirror.

`kDependentNotifyClassDriverOverride`
: The Display Manager sends notification that a video driver has been overridden with a newer revision.

`kDependentNotifyClassDisplayMgrOverride`
: The Display Manager sends notification that it has been upgraded with a newer revision.

`kDependentNotifyClassProfileChanged`
The Display Manager sends notification when the profile associated with a display changes.

Display / Device ID Constants

The Display Manager uses these values to help with the configuration of the display.

```
enum {
     kDummyDeviceID = 0x00FF,
     kInvalidDisplayID = 0x0000,
     kFirstDisplayID = 0x0100
};
```

Constant descriptions

`kDummyDeviceID`	This is the ID of the dummy display, used when the last "real" display is removed.
`kInvalidDisplayID`	This is the ID of the invalid display, which has been removed from the active display list.
`kFirstDisplayID`	When your application sets this bit it asks the Display Manager to return the ID of the first display device on the active display list.

Display Mode Flags

The structure `DMDisplayModeListEntryRec` (page 83) uses these values for its `displayModeFlags` field.

```
   kDisplayModeListNotPreferredBit  = 0,
   kDisplayModeListNotPreferredMask = 1
```

Constant descriptions

`kDisplayModeListNotPreferredBit`
: Indicates there is a better timing available and that this timing should be shown only if the user wants to see all options.

`kDisplayModeListNotPreferredMask`
: `(1 << kDisplayModeListNotPreferredBit)`

Display Version Values

These values supply information to the structure `DMDisplayModeListEntryRec` (page 83).

```
enum {
     kDisplayTimingInfoVersionZero = 1,
     kDisplayTimingInfoReservedCountVersionZero = 16,
     kDisplayModeEntryVersionZero = 0,
     kDisplayModeEntryVersionOne = 1
};
```

Constant descriptions

`kDisplayTimingInfoVersionZero`
: This relative information is always `NULL` in this version.

`kDisplayTimingInfoReservedCountVersionZero`
: This relative information is always `NULL` in this version.

`kDisplayModeEntryVersionZero`
: This relative information is always `NULL` in this version.

`kDisplayModeEntryVersionOne`
: This relative information is always `NULL` in this version.

Notification Messages

Display Manager functions needed for dependency notification and event processing use the following notification message selectors in extended application-defined functions. `DMRegisterExtendedNotifyProc` (page 74) gets all these messages. Applications should update all information about the display configurations at this point.

```
enum {
     kDMNotifyInstalled = 1,
     kDMNotifyEvent = 2,
     kDMNotifyRemoved = 3,
     kDMNotifyPrep = 4,
     kDMNotifyExtendEvent = 5,
     kDMNotifyDependents = 6,
     kDMNotifySuspendConfigure = 7,
     kDMNotifyResumeConfigure = 8

};
```

Constant descriptions

`kDMNotifyInstalled`
: The Display Manager provides this message during a callback function to if your application has installed an extended notification procedure pointer for the first time. The Display Manager provides this message in the `notifyData` parameter of `MyExtendedNotificationProc` (page 77).

`kDMNotifyEvent`
: The Display Manager provides this message when an Apple event update occurs, after a display configuration change is made. This is the only time non-extended notifications are called.

`kDMNotifyRemoved`
: The Display Manager provides this message when the function `DMRemoveExtendedNotifyProc` (page 75) is called on your function.

`kDMNotifyPrep`
: Before passing `kDMSNotifyRemoved`, the Display Manager provides this message to indicate that it is about

to begin to configure. Calling `DMBeginConfigureDisplays` (page 55) tells the Display Manager to send this message.

`kDMNotifyDependents`

The Display Manager provides this message to `DMSendDependentNotification` (page 76).

`kDMNotifySuspendConfigure`

The Display Manager passes this selector to notify your upp that configuration is temporarily suspended. For instance, if a video game makes a temporary change to the display configuration, the game is expected to resume configuration and restore video before allowing other applications to access the screen.

`kDMNotifyResumeConfigure`

The Display Manager passes this selector to notify your application when previously suspended configuration is resumed. Your application can then replace windows and icons, and change depth mode if necessary.

Notification Types

The function `DMSendDependentNotification` (page 76) uses these values in the `notifyType` parameter.

```
enum {
     kFullNotify = 0,
     kFullDependencyNotify = 1
};
```

Constant descriptions

`kFullNotify`

The Display Manager sets this bit to provide the major Apple notification event.

`kFullDependencyNotify`

The Display Manager sets this bit to provide notification only to those applications that need to know about interrelated functionality. It is used for updating the user interface.

Switch Flags

In its `switchFlags` parameter, the function `DMCheckDisplayMode` (page 47) returns a pointer to a long integer that specifies flags in two of its bits. The following constants represent bits that are set to 1. These bits are set by the Display Manager, not your application

```
enum {
     kNoSwitchConfirmBit = 0,
     kDepthNotAvailableBit = 1,
     kShowModeBit = 3,
     kModeNotResizeBit = 4,
     kNeverShowModeBit = 5
};
```

Constant descriptions

`kNoSwitchConfirmBit`
: If the Display Manager sets this bit the display mode is required to function correctly. Your application does not need to provide confirmation if the user switches to this mode.

`kDepthNotAvailableBit`
: If the Display Manager sets this bit the pixel depth of the specified device is not available for the specified display mode.

`kShowModeBit`
: If the Display Manager sets this bit your application should display this mode to the user, even though it may require confirmation.

`kModeNotResizeBit`
: If the Display Manager sets this bit you should not use this mode to resize a display; this mode drives a different connector in cards than in a built-in display.

`kNeverShowModeBit`

If the Display Manager sets this bit you should not show the mode in the user interface.

Video Depth Mode Values

The functions `DMSetDisplayMode` (page 56) and `DMCheckDisplayMode` (page 47) use these values in the `depthMode` parameter, to set or check pixel depth. A depth mode specified by the `firstVidMode` constant represents the lowest supported pixel depth—minimally, 1 bit per pixel. A depth mode specified by the `secondVidMode` constant represents the next highest supported pixel depth—often, but not necessarily, 2 bits per pixel. If a video device supports 4 bits per pixel instead of 2 as its next highest pixel depth, then its driver uses the `secondVidMode` constant to represent 4 bits per pixel. In this manner, the remaining constants signifying depth modes specify an ordered set of increasingly higher pixel depths.

```
enum {
     firstVidMode = 128,
     secondVidMode = 129,
     thirdVidMode = 130,
     fourthVidMode = 131,
     fifthVidMode = 132,
     sixthVidMode = 133

};
```

Constant descriptions

`firstVidMode`	Represents lowest supported pixel depth.
`secondVidMode`	Represents next highest supported pixel depth.
`thirdVidMode`	Represents next highest supported pixel depth.
`fourthVidMode`	Represents next highest supported pixel depth.
`fifthVidMode`	Represents next highest supported pixel depth.
`sixthVidMode`	Represents next highest supported pixel depth.

Result Codes

noErr	0	No error
paramErr	–50	Invalid value passed in a parameter
kDMGenErr	–6220	Indeterminate error
kDMMirroringOnAlready	–6221	Video mirroring is already enabled
kDMWrongNumberOfDisplays	–6222	Wrong number of displays
kDMMirroringBlocked	–6223	Video blocked
KDMCantBlock	–6224	Video mirroring already enabled and can't be blocked; use `DMUnMirrorDevice`, then call `DMBlockMIrroring` again.
kDMMirroringNotOn	–6225	Video mirroring is not currently enabled.
kSysSWTooOld	–6226	Some piece of system software is too old for the Display Manager to operate.
kDMSWNotInitializedErr	–6227	Required pieces of system software are not initialized.
kDMDriverNotDisplayMgrAwareErr	–6228	The video driver for the display does not support the Display Manager.
kDMDisplayNotFoundErr	–6229	There are no `GDevice` structures for displays in the device list.
kDMDisplayAlreadyInstalledErr	–6230	The display is already in the device list and can't be added.
	–6231	No device–main display cannot move
	–6232	Item found